Computing exact integrals of interpolated images within polygons and their derivatives

Martin de La Gorce

April 28, 2019

Abstract

This document describes a numerical method to compute exactly the integral on an interpolated image inside a polygon and the derivatives of this integral with respect to the polygon's vertices locations. The text and formulas are mostly from my PhD Thesis [7]

1 Analytic formulas

1.1 Integal formulation

Suppose we are given

- a discretized function f with domain $D = \{0, W-1\} \times \{0, H-1\} \subset \mathbb{R}^2$ taking values in $\mathbb{R}$. We formulate the interpolation of f in the continuous domain $[0, W] \times [0, H]$ as a convolution by a kernel k:

$$f_c(x, y) = \sum_{(i,j) \in D} f(i, j)\ k(x - i, y - j) \tag{1}$$

- a non self-intersecting polygon defined through a sequence of N vertices

$$V = (v_1, \ldots, v_N)$$

with $v_i = (x_i, y_i)$ circulating the polygon boundary counter-clockwise.

We denote $P(V) \subset \mathbb{R}^2$ the inside of the polygon and χ_P its characteristic function i.e. the function that takes value 1 inside the plygon and 0 ouside.

The integral of the function f_c inside the polygon $P(V)$ writes:

$$S(V) = \iint_{P(V)} f_c(x, y) dx dy \tag{2}$$

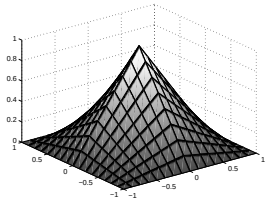

Figure 1: kernel for bilinear interpolation

1.2 interpolation kernels

The kernel depends on the type of interpolation we use:

- The nearest neighbor (shifted by 0.5) interpolation is obtained using the kernel that is constant and equal to 1 within the square $[0,1]^2$ and zeros outside the square:

$$k_n(x,y) = 1 \text{ if } (x,y) \in [0,1]^2,\ 0 \text{ else} \tag{3}$$

  Some basic variables manipulation will lead to:

$$f_c(x,y) = \sum_{(i,j)\in L_d} f(i,j)k_n(x-i,y-j) = f(\lfloor x \rfloor, \lfloor y \rfloor) \tag{4}$$

  with $\lfloor x \rfloor$ the greatest integer that is smaller or equal to x.

- The bilinear interpolation is obtained using the kernel k_b defined as follows:

$$k_b(x,y) = (1-|x|)(1-|y|) \text{ if } (x,y) \in [-1,1]^2,\ 0 \text{ else} \tag{5}$$

  The kernel k_b is shown in [Figure (1)]. We define the function $\epsilon(x) = x - \lfloor x \rfloor$. With some rewriting we get:

$$\begin{aligned} f_c(x,y) =& (1-\epsilon(x))(1-\epsilon(y))f(\lfloor x \rfloor, \lfloor y \rfloor) \\ &+ (\epsilon(x))(1-\epsilon(y))f(\lfloor x \rfloor + 1, \lfloor y \rfloor) \\ &+ (1-\epsilon(x))(\epsilon(y))f(\lfloor x \rfloor, \lfloor y \rfloor + 1) \\ &+ (\epsilon(x))(\epsilon(y))f(\lfloor x \rfloor + 1, \lfloor y \rfloor + 1) \end{aligned} \tag{6}$$

1.3 First order derivates with respect to vertices

We aime at computing $\partial S(V)/\partial v_j$. Given a vertex of the polygon v_k we use v_{k-1} and v_{k+1} to denote the preceding and following vertices in the polygon The derivative of the functional $S(V)$ with respect to a vertex v_k of the polygon writes :

$$\begin{aligned}\frac{\partial S}{\partial v_k} =& J(v_k - v_{k-1}) \int_0^1 f_c(tv_k + (1-t)v_{k-1})tdt \\ &+ J(v_{k+1} - v_k) \int_0^1 f_c(tv_k + (1-t)v_{k+1})tdt\end{aligned} \tag{7}$$

with J the roation matrix defined by

$$J = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Let us introduce l_k and $\hat{n}_k$ being the length and the normal of the segment $\overline{v_{k-1}v_k}$. We have:

$$J(v_k - v_{k-1}) = \hat{n}_k l_k \tag{8}$$

One should point out that, for each vertex v_k, the information of the functional f_c is being integrated along its adjacent edges. Using a mechanical metaphor, the resulting vector $\partial S/\partial v_k$ can be interpreted as a 2D data force acting on the polygon vertex v_k.

Note that f_c does not have to be continuous everywhere for the integral $S(P)$ to be differentiable. When using shifted-nearest-neighbor interpolation, f_c is discontinuous for any point that lies on a vertical or an horizontal line of the pixel grid. However the first order derivative $\frac{\partial S}{\partial v_k}$ of the integral $S(V)$ is defined and continuous as long as none of the two adjacent polygon edge matches a vertical line or an horizontal line of the pixel grid.

1.4 Second order derivatives

In scenerios where we use the integral $S(V)$ as function we aim to minimize we may want to get access to the hessian of $S(V)$ in oder to be able to use a newton method.

The second order derivatives of the functional $S(V)$ with respect to two vertices of the polygon is derived by differentiating the equation eqn.7. For any pair of vertices (v_i, v_j) of the polygon the order derivative $\partial^2 S/\partial v_i v_j$ is a 2 by 2 matrix. If the two vertices are not connected by an edge then $\partial^2 S/\partial v_i v_j = 0_{2\times 2}$. As a consequence, if we gather all the derivatives $\frac{\partial^2 S}{\partial^2 v_j}$

into a single matrix M composed of 2×2 sub-matrices $M_{2i:2i+1,2j:2j+1}$ corresponding to the 2×2 matrices $\partial^2 S / \partial v_i v_j$, then this matrix M is expected to be sparse and symmetric.

If two vertices are connected ($i = j + 1$), then by setting $k = j$ in the equation eqn.7 and differentiating, we obtain:

$$\begin{aligned} \frac{\partial^2 S}{\partial v_{j+1} \partial v_j} = &+ J \int_0^1 f_c((1-t)v_{j+1} + tv_j) t dt \\ &+ J(v_{j+1} - v_j) \int_0^1 \nabla f_c((1-t)v_{j+1} + tv_j)(1-t) t dt \end{aligned} \tag{9}$$

Because of the symmetry of derivatives we also get $\frac{\partial^2 S}{\partial v_j \partial v_{j+1}} = \frac{\partial^2 S}{\partial v_{j+1} \partial v_j}^T$. Note that the function f_c should be differentiable almost everywhere along the two adjacent edges in order to have the second order derivative being properly defined. One should recall the bilinear interpolation f_c is not differentiable for any point that lies on a vertical or an horizontal line of the pixel grid. Therefore the second order derivative $\partial^2 S / \partial v_{j+1} \partial v_j$ is defined for the bilinear interpolation if and only if the edge $\overline{v_j v_{j+1}}$ does not match a vertical or an horizontal line of the image grid.

The second order derivative when $i = j$ writes

$$\begin{aligned} \frac{\partial^2 S}{\partial^2 v_j} = &J \int_0^1 f_c((1-t)v_{j-1} + tv_j) t dt - J \int_0^1 f_c((1-t)v_{j+1} + tv_j) t dt \\ &+ J(v_j - v_{j-1}) \int_0^1 \nabla f_c((1-t)v_{j-1} + tv_j) t^2 dt \\ &+ J(v_{j+1} - v_j) \int_0^1 \nabla f_c((1-t)v_{j+1} + tv_j) t^2 dt \end{aligned} \tag{10}$$

The symmetry of the 2×2 matrix $\partial^2 L / \partial^2 v_j$ is not obvious when considering this equation but can be proved using integration by parts. The second order derivative $\partial^2 S / \partial^2 v_j$ is defined for bilinear interpolation and if none of the two adjacent edge $\overline{v_{j-1} v_j}$ $\overline{v_j v_{j+1}}$ matches a vertical or an horizontal line of the image grid.

Note that in the context of active contour, several authors [11, 4, 12, 2] proposed the use of second order derivatives to accelerate the convergence. Their formulations are done for general continuous and differentiable curves, generally require Gâteaux derivatives in the derivation and are implemented using level set methods. In our case we have an explicit formulation of the region boundary as a polygon and, as a consequence, the second order derivatives are simpler to obtain.

2 Relationship with antialiasing

There is a strong relationship between the problem of computing *exactly* the integral $S(V)$ and performing exact anti-aliased image of the polygon. In the context of computer graphics and signal processing in general the aliasing problem appears when a signal with high frequencies (spacial frequencies for an image) is sampled at a frequency smaller than $f_e/2$ (f_e being the highest frequencies with no negligible energy in the signal). In computer graphics this appears when rendering geometric primitives with sharp boundaries on a discrete pixel grid. The abrupt intensity discontinuities at the boundaries cause high amplitude spectral components at extremely high frequencies. In practice if one discretizes an image composed of object with sharp boundaries, this result in "jaggies" or stair-like artifacts. In real cameras the optics creates some low-pass filtering (blurring) and pixel have spacial extension, which removes the aliasing effect.

Three techniques are now commonly used to reduce aliasing: pre-filtering, uniform super-sampling, and stochastic sampling. Pre-filtering consists in filtering the image in order to remove high spatial frequencies before sampling at pixel rates. For each pixel, the combination of the filtering and sampling operations can be interpreted as the computation of the scalar product of the filter kernel centered at the pixel with the continuous image. The super-sampling methods consist in increasing the sampling rates (and hence the Nyquist rates) to some small multiple of pixel rates and then form each pixel intensity of the discrete image from a weighted average of neighboring samples. The super-sampling methods do not remove the discontinuous intensity changes, but reduce their magnitude, when the objects being drawn move continuously. Stochastic super-sampling methods randomly displace the super-sampling positions so that any aliased components appear as uncorrelated noise in the discrete image. The noise induces temporal discontinuities in the discrete image when the objects being drawn displace continuously. Only the pre-filtering method can provide a discrete image that varies continuously when the rendered object displace continuously.

When the filter is the characteristic function of a unit square and the object being drawn are polygons with uniform colors, the antialiasing process can be re-interpreted as follows: Each polygon is clipped to the extend of the pixel and the contribution of each polygon to the pixel is weighted by area of the clipped region. This method is referred as the unweighted area sampling in [9](p133-134) and exact algorithms have been proposed in [5] and [8] to compute the weights. In [5], each polygon is clipped to each

pixel using the Weiler-Atherton method which is quite expensive. In [8] the approach is extended by replacing the area computation with a contour integral. Several optimizations that exploit the coherence of scan-conversion are also proposed. A method to compute exact antialiased triangles with any *prisme spline* filter (which include the 2D box filter and the filter k_b that is used for bilinear interpolation) has been proposed in [15]. Recent methods have been proposed to compute exact prefiltered antialiasing of polygon and bezier curves using various kernels [1, 14]. In order to see the relationship between the problem of computing the integral $S(P)$ and performing anti aliasing we rewrite the integral defining $S(P)$ as follows:

$$\begin{aligned} S(P) &= \iint f_c(x,y)\chi_P(x,y)dxdy \\ &= \iint \left[\sum_{(i,j)} f(i,j)k(x-i,y-j))\right] \chi_P(x,y)dxdy \\ &= \sum_{(i,j)} f(i,j) \underbrace{\iint k(x-i,y-i))\chi_P(x,y)dxdy}_{\tilde{M}(i,j)} \end{aligned} \tag{11}$$

The integral $S(V)$ is rewritten as a finite sum over the image pixels where the contribution of each pixel is weighted by $\tilde{M}(i,j)$. The image $\tilde{M}$ can be interpreted as an anti-aliased version of the polygon binary image M that is the pixel sampling of the characteristic function $\chi_P(x,y)$. Indeed $\tilde{M}(i,j)$ corresponds to the evaluation at the integer location (i,j) of χ_P convolved with the kernel k. This convolution filters out (or reduce) high spatial frequencies.

If we use the nearest neighbor interpolation shifted by 0.5 (eqn.4) one can easily interpret $\tilde{M}$ as the antialiased version to the polygon characteristic function χ_P using the *unweighted area sampling* method. We rewrite the equation using the corresponding kernel k:

$$\begin{aligned} \tilde{M}(i,j) &= \iint k(x-i,y-j)\chi_P(x,y)dxdy \\ &= \iint_{(x,y)\in[i,i+1]\times[j,j+1]} \chi_P(x,y)dxdy \\ &= \iint_{\chi_P\cap[i,i+1]\times[j,j+1]} 1dxdy \end{aligned} \tag{12}$$

For each pixel (i,j) the value $\tilde{M}(i,j)$ is the measure of the area of inter-

section of the polygon $P(V)$ with the pixel (i, j) that is spatially extended into a unit square $[i, i+1] \times [j, j+1]$. The integral $S(V)$ is a finite sum over the image pixels where the contribution of each pixel is weighted according to the surface of the intersection between the pixel and the polygon.

In order to compute the integral $S(V)$ we could use the antialiasing algorithms proposed in [5] and in [8]. However, as shown in the next section, it is not actually needed to compute the antialiased image M when computing $S(P)$. Our method uses the Green theorem to reduce the computation complexity by replacing the integral within the polygon interior by an integral along it boundary.

3 Numerical integration

3.1 using green's theorem

The integral $S(V)$ of f_c inside the polygon $P(V)$ can be rewritten as an integral on its boundary $\Gamma(V) \equiv \partial P(V)$ using the green's theorem:

$$\iint_{P(V)} f_c(x, y)dxdy = \oint_{\Gamma(V)} < F(s), \hat{n}(s) > ds \tag{13}$$

where $\hat{n}$ denotes the outward unit normal to $\Gamma(V)$, ds the Euclidean arc length element and the circle in the right-hand integral indicates that the curve is closed and $F(s)$ is any function such that $div(F) = f_c$ We choose $F_y = 0$ and

$$F_x(x, y) = \int_{t=0}^{x} f_c(t, y)dt \tag{14}$$

We have $F(s) = F_x(s).\hat{x}$. Using this function integral $S(V)$ becomes:

$$S(V) = \iint_{P(V)} f_c(x, y)dxdy = \oint_{\Gamma(V)} F_x(s) < \hat{n}(s).\vec{x} > ds \tag{15}$$

The integral over $P(V)$ is reduced to an integral over the polygon boundary $\Gamma(V)$. If we use the shifted nearest neighbor interpolation then F_x can be rewritten as:

$$F_x(x, y) = \sum_{u=0}^{\lfloor x \rfloor - 1} f(u, \lfloor y \rfloor) + (x - \lfloor x \rfloor) f(\lfloor x \rfloor, \lfloor y \rfloor) \tag{16}$$

3.2 Line segments

Let consider a segment $\overline{v_k v_{k+1}}$ whose extremities are $v_k \equiv (x_k, y_k)$ and $v_{k+1} \equiv (x_{k+1}, y_{k+1})$ (like the segment $\overline{ab}$ in [Figure (2.a)]) The contribution of the segment $\overline{v_k v_{k+1}}$ to the integral is:

$$C_k \equiv \int_{\overline{v_k v_{k+1}}} F_x(s) < \hat{x}, \hat{n_k} > ds \tag{17}$$

The normal vector of the segment $\hat{n_k}$ writes $\hat{n_k} = J(v_{k+1} - v_k)/l_k$ with $J = \left[\begin{smallmatrix} 0 & 1 \\ -1 & 0 \end{smallmatrix}\right]$ and $l_k \equiv |v_k - v_{k+1}|$ the length of the segment.

$$< \hat{x}, \hat{n_k} >= (y_{k+1} - y_k)/l_k$$

We can parameterize each segment linearly using a parameter $t \in [0, 1]$:

$$\overline{v_k v_{k+1}} = \{(1-t)v_k + tv_{k+1} \mid t \in [0, 1]\} \tag{18}$$

Using this parameterization for each segment, the integral rewrites:

$$C_k = (y_{k+1} - y_k) \int_0^1 F_x((1-t)v_k + tv_{k+1})dt \tag{19}$$

Supposing $y_k < y_{k+1}$, the segment can also be parameterized using the following curve

$$\begin{array}{rccc} r: & [y_k, y_{k+1}] & \rightarrow & \mathbb{R}^2 \\ & v & \mapsto & (\alpha v + \beta, v) \end{array} \tag{20}$$

with $\alpha = (x_k - x_{k+1})/(y_k - y_{k+1})$ and $\beta = x_k - \alpha y_k$. We get $\overline{v_k v_{k+1}} = r([y_k, y_{k+1}])$. Using this parameterization of the segment, its contribution rewrites:

$$C_k = \int_{v=y_k}^{y_{k+1}} F_x(r(v)) < \hat{n}_k, \vec{x} > |r'(v)|dv \tag{21}$$

Assuming that we are walking counterclockwise around the polygon, the outside normal $\hat{n}$ points towards the right while going from v_k to v_{k+1} (i.e. the inside of the polygon is on the left when going from v_k to v_{k+1}). Because $y_k < y_{k+1}$ one can show that the outside normal points toward the right and we have $< \hat{n}_k, \vec{x} >$ positive. With some calculus one can show that we have $< \hat{n}_k, \vec{x} > |r'(v)| = 1$ and we get:

$$\begin{aligned} C_k &= \int_{v=y_k}^{y_{k+1}} F_x(r(v))dv \\ &= \int_{v=y_k}^{y_{k+1}} F_x(\alpha v + \beta, u)du \\ &= \int_{v=y_k}^{y_{k+1}} \int_{u=0}^{\alpha v+\beta} f_c(u, v)dudv \end{aligned} \tag{22}$$

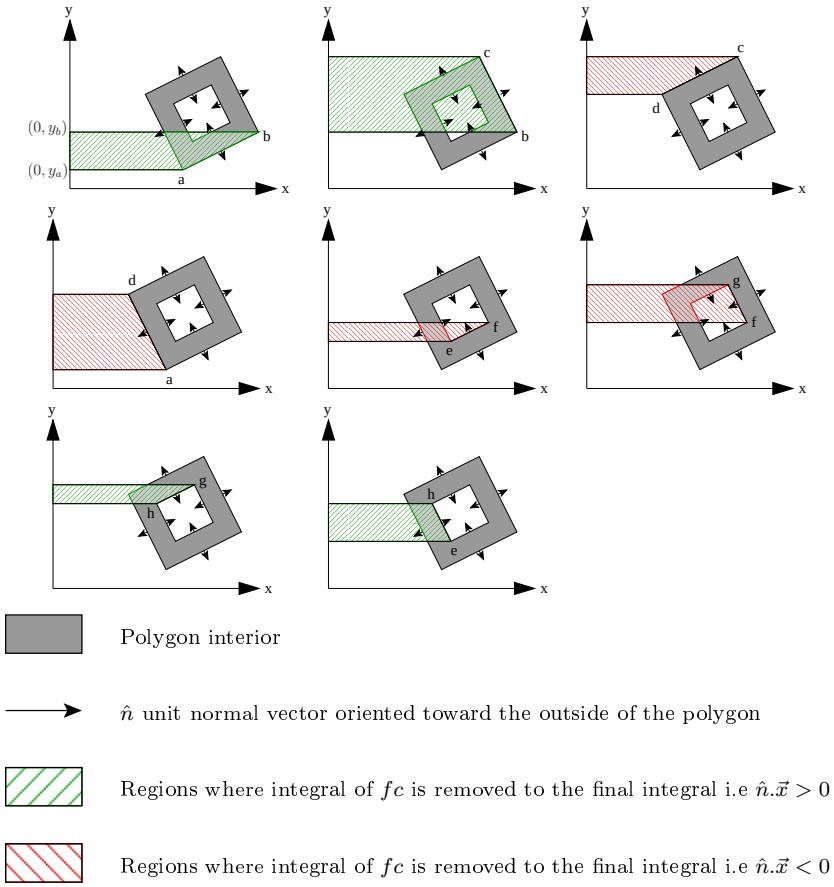

Figure 2: trapezoid integrals

The last equation allows us to re-interpret easily the contribution of the segment to the contour integral. It corresponds to the integral of f_c in the trapezoidal region on the left of the segment [Figure (2)] i.e with corners of coordinates (x_k, y_k),(x_{k+1}, y_{k+1}),$(0, y_{k+1})$, and $(0, y_k)$. If the edge is on a hole boundary, the contribution of the segment to the contour integral is the integral of $-f_c$ in the trapezoidal region on the left of the segment.

Depending on the interpolation scheme (shifted-nearest neighbor or bi-linear) the function f_c is either constant or quadric on each pixel. In order to compute exactly the integral of F_x along the segment $\overline{v_k v_{k+1}}$ one needs to split this line segment into sub-pixel fragments. An example of such a fragmentation in shown in [Figure (3)]. Each subpixel fragment is contained

in an unit square pixel $[x, x+1] \times [y, y+1]$, with $(x, y) \in D$. This is done by clipping the segment against each pixel, which can be done efficiently using the algorithm whose pseudo code is provided in 1. The segment $\overline{v_k v_{k+1}}$ is clipped by computing its intersections with the pixel grid that is composed of vertical lines with integer x coordinates and horizontal lines with integer y coordinates. The intersections are detected in an order such that their distance from v_k is increasing. Each segment whose extremities are two successive intersections with the pixel grid is contained in a pixel square.

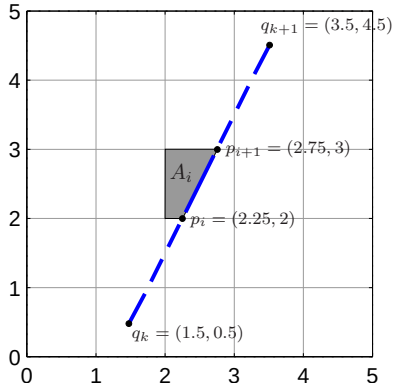

Figure 3: Sub-pixel fragmentation of a segment. Example with $v_k = (1.5, 0)$ and $v_{k+1} = (3.5, 4.5)$

Once the segment is partitioned into sub-pixel fragments we can easily perform integration on each fragment. Let $p_i \equiv (x_i, y_i)$ and $p_{i+1} \equiv (x_{i+1}, y_{i+1})$ be the extremities of a subpixel segment of $\overline{v_k v_{k+1}}$. We have $\overline{p_i p_{i+1}} \subset \overline{v_k v_{k+1}}$. The two extremities p_i and p_{i+1} belong to the same pixel which means that we have

$$\left\{ \begin{array}{lcl} (x_i, y_i) & \in & [k, k+1] \times [l, l+1] \\ (x_{i+1}, y_{i+1}) & \in & [k, k+1] \times [l, l+1] \end{array} \right. \tag{23}$$

with $k = \lfloor (x_i + x_{i+1})/2 \rfloor$ and $l = \lfloor (y_i + y_{i+1})/2 \rfloor$. Note that we may have $\lceil x_i \rceil \neq \lceil x_{i+1} \rceil$ of $\lceil y_i \rceil \neq \lceil y_{i+1} \rceil$ if x_i or y_i are integers. We denote $c^i \equiv (x_c^i, y_{ci})$ the center of the i^{th} subpixel segment. we have $x_{ci} = (x_i + x_{i+1})/2$ and $y_{ci} = (y_i + y_{i+1})/2$. We introduce the two variables $\Delta_{yi} \equiv y_{i+1} - y_i$ and $\Delta_{xi} \equiv x_{i+1} - x_i$.

3.2.1 Nearest neighbor interpolation

If we use the shifted nearest neighbor interpolation with the kernel k_n, then the function f_c is constant within each unit square corresponding to a pixel. Furthermore, the function F_x is linear within each of these squares. Because p_i and p_i+1 belong to the same pixel and we use the shifted nearest neighbor interpolation, the function $u \mapsto F_x(r(u))$ is linear on the interval $[y_i, y_{i+1}]$ and we can evaluate the integral by simply evaluating the center point c^i of the interval:

$$\begin{aligned} C_i &= (y_{i+1} - y_i) F_x\big(r((y_i + y_{i+1})/2)\big) \\ &= \Delta_{yi} F_x(x_{ci}, y_{ci}) \end{aligned} \tag{24}$$

We get from eqn 16:

$$C_i = (y_{i+1} - y_i) \sum_{u=0}^{\lfloor x_c \rfloor - 1} f(u, \lfloor y_c \rfloor) + \underbrace{(y_{i+1} - y_i)(x_c - \lfloor x_c \rfloor)}_{A_i} f(\lfloor x_c \rfloor, \lfloor y_c \rfloor) \tag{25}$$

The value of A_i is the area of the intersection of the trapezoidal region on the left of the fragment $\overline{p_i p_{i+1}}$ and the pixel as shown in [Figure (3)].

Consequently, our algorithm could be easily adapted to implement the antialiasing method we referred as the "unweighted area sampling" in the previous section. After meticulous search of similar algorithm in the literature, it appeared that our method bears similarities with the one proposed in [6] that adapts the Bresenham [3] line rasterization method using two error variables. However, our method that clips segment on pixel grids appears to be simpler. The source code can be found on the Graphic Gems Repository [1]. Our method also bears similarities with antialiasing methods proposed in [16] [2],[21] and [10]. These algorithms adapt the Bresenham [3] rasterization line algorithm to perform antialiasing. These algorithms differ from our contribution because they consists in clipping the line segment only against vertical lines of the pixel grid or only against horizontal lines, depending on the direction of the line segment. The resulting approximate coverage agrees with the exact coverage only when the edge does not cross a line on the pixel grid that is orthogonal to the set of lines chosen for the clipping. Furthermore these algorithms do not handle line ends properly.

[1] http://tog.acm.org/resources/GraphicsGems/gemsiii/edgeCalc.c

[2] A pseudo code can be found here http://www710.univ-lyon1.fr/~jciehl/Public/educ/ENS/2003/antialiassage.pdf

3.2.2 Bilinear interpolation

From the equation eqn.6 that defines the bilinear interpolation and the equation eqn.14 that defines F_x we get:

$$\begin{aligned} F_x(x,y) =& \epsilon(y) \left(\sum_{u=0}^{\lfloor x \rfloor} f(u, \lceil y \rceil) - \frac{1}{2} f(0, \lceil y \rceil) - \frac{1}{2} f(\lfloor x \rfloor, \lceil y \rceil) \right) \\ &+ (1 - \epsilon(y)) \left(\sum_{u=0}^{\lfloor x \rfloor} f(u, \lfloor y \rfloor) - \frac{1}{2} f(0, \lfloor y \rfloor) - \frac{1}{2} f(\lfloor x \rfloor, \lfloor y \rfloor) \right) \\ &+ \epsilon(x) f_c(\lfloor x \rfloor + \frac{\epsilon(x)}{2}, y) \end{aligned} \tag{26}$$

The contribution of the i^{th} sub-pixel fragment can be shown to be:

$$\begin{aligned} C_i = \Delta_{yi} F_x(\lfloor x_{ci} \rfloor, y_{ci}) &+ c_{00} f(\lfloor x_{ci} \rfloor, \lfloor y_{ci} \rfloor) + c_{10} f(\lfloor x_{ci} \rfloor + 1, \lfloor y_{ci} \rfloor) \\ &+ c_{01} f(\lfloor x_{ci} \rfloor, \lfloor y_{ci} \rfloor + 1) + c_{11} f(\lfloor x_{ci} \rfloor + 1, \lfloor y_{ci} \rfloor + 1) \end{aligned} \tag{27}$$

with c_{00},c_{10},c_{01} and c_{11} defined as:

$$c_{11} \equiv \frac{\Delta_{xi} \Delta_{yi}^2}{12} \epsilon(x_{ci}) + \frac{\Delta_{yi} \Delta_{xi}^2}{24} \epsilon(y_{ci}) \tag{28}$$

$$c_{10} \equiv -c_{11} + \frac{\Delta_{yi} \Delta_{xi}^2}{24} \tag{29}$$

$$c_{01} \equiv -c_{11} + \frac{\Delta_{xi} \Delta_{yi}^2}{12} \tag{30}$$

$$c_{00} \equiv -c_{10} - \frac{\Delta_{xi} \Delta_{yi}^2}{12} \tag{31}$$

These equations are quite tedious to derive and we used the Maple symbolic computation tool. Note that methods to integrate 2D polynomials into polygons have been proposed in [20, 18, 13, 17, 19] and their result might be useful to extend the integration in the polygon for other polynomial interpolation schemes such as bi-cubic interpolation The method proposed in [15] to render antialiased triangles is also of interest.

3.3 Derivatives

In order to compute the *exact* derivative given by equation 7, we need to clip each segment onto the pixel grid and to integrate on each subpixel fragment.

By stating that the gradient is *exact* we mean that, given a definition of f_c as nearest neighbor or bilinear interpolation of f, we are able to compute exactly the equation 7. The method to compute exactly the gradient is very similar to the method that is used to compute exactly the integral $S(V)$. The pseudo code for shifted nearest neighbor interpolation is at alg.2 and the pseudo code for the bilinear interpolation at alg.3 (this second algorithm also compute second order derivatives whose derivation is explained in the section 1.4).

The pseudo-code to compute first and second order derivative for bilinear interpolation of f is provided in alg.3

Note that we could approximate the integrals in the gradient and the hessian by finite sums through uniform sampling of points along each segment and evaluation of f_c and ∇f_c at these locations. However the equation 10 does not exhibit obvious symmetry and the approximate Hessian we would obtain is not symmetric. Therefore we would have to impose the symmetry of the Hessian by averaging the computed Hessian matrix and its transpose.

The Hessian we obtain using the exact integration along the edges using the algorithm alg.3 is symmetric up to round-off errors.

3.4 Improvements

The symmetry of the hessian is not obvious from our equations, maybe we could obtain an expression where the symmetry is obvious and hopefully that would lead to a slightly simplier expression that would require a bit less computation for its exact computation.

Algorithm 1: clipSegmentOnPixelGrid

Data: Two segment extremities $a \equiv (x_a, y_a)$ and $b \equiv (x_b, y_b)$
Result:a list of Point $p_1, \ldots, p_N$ and their normalized curvilinear coordinate $t_1, \ldots t_N \in [0, 1]$ on the segment such that $p_1 = (x_a, y_a)$, $p_N = (x_b, y_b)$, $\overline{ab} = \bigcup_{i=1}^{N} \overline{p_i, p_{i+1}}$ and for each i the two segments extremities p_i and p_{i+1} lies inside the same pixel

// get the first intersection with a vertical line

if $x_b > x_a$ **then**
  $\delta_v^x \leftarrow 1$; $x_v \leftarrow \lfloor x_a \rfloor + 1$;
else
  $\delta_v^x \leftarrow -1$; $x_v \leftarrow \lceil x_a \rceil - 1$;
if $\delta_x x_v < \delta_x x_b$ **then**
  $N_v \leftarrow \lceil \delta_x(x_b - x_v) \rceil$; $\delta_v^T \leftarrow \delta_v^x/(x_b - x_a)$; $\delta_v^y \leftarrow (y_b - y_a)\delta_v^T$;
  $y_v \leftarrow \delta_v^x \delta_v^y (x_v - x_a) + y_a$; $t_v \leftarrow \delta_v^x \delta_v^T (x_v - x_a)$;
else
  $N_v = 0$;$t_v = 1$;

// get the first intersection with an horizontal line

if $y_b > y_a$ **then**
  $\delta_h^y \leftarrow 1$; $y_h \leftarrow \lfloor y_a \rfloor + 1$;
else
  $\delta_h^y \leftarrow -1$; $y_h \leftarrow \lceil y_a \rceil - 1$;
if $\delta_y y_h < \delta_y y_b$ **then**
  $N_h \leftarrow \lceil \delta_y(y_b - y_h) \rceil$; $\delta_h^T \leftarrow \delta_h^y/(y_b - y_a)$; $\delta_h^x \leftarrow (x_b - x_a)\delta_h^T$;
  $x_h \leftarrow \delta_h^x \delta_h^y (y_h - y_a) + x_a$; $t_h \leftarrow \delta_v^y \delta_v^T (y_v - y_a)$;
else
  $N_h \leftarrow 0$;$t_h = 1$;

$p_1 = (x_a, y_a)$; $t_1 = 0$;

// Loop until it reaches the extremity b

$N \leftarrow N_h + N_v + 2$;
for $k \leftarrow 2$ **to** N **do**
  if $t_v < t_h$ **then**
    // the next intersection is with a vertical line
    $p_k \leftarrow (x_v, y_v)$; $t_k \leftarrow t_v$;
    // get the next intersection with an vertical line
    $(x_v, y_v, t_v) \leftarrow (x_v + \delta_v^x, y_v + \delta_v^y, t_v + \delta_v^T)$;
  else
    // The next intersection is with an horizontal line
    $p_k \leftarrow (x_h, y_h)$; $t_k \leftarrow t_h$;
    // get the next intersection with an horizontal line
    $(x_h, y_h, t_h) \leftarrow (x_h + \delta_h^x, y_h + \delta_h^y, t_h + \delta_h^T)$;

$p_N \leftarrow (x_b, y_b)$; $t_N = 1$;

Algorithm 2: integrateSegmentDerivativeNearestNeighbor

Data: The polygon vertices $(v_i)_{i=1}^N$ and a discretized function $f(x, y)$
Result: The derivatives $\dot{v}_i \equiv \frac{\partial I}{\partial v_i}$ of the integral I of f_c inside the polygon with f_c the shifted nearest neighbor interpolation of f

for $i \leftarrow 1$ **to** N **do**
  $\dot{v}_i \leftarrow [0, 0]^T$;
// loop over edges of the polygon
for $i \leftarrow 1$ **to** N **do**
  $j = mod(i, N) + 1$;
  // clip the segment $\overline{v_i, v_j}$ *on the pixel grid*
  $(p_k, t_k)_{k=1}^K \leftarrow$ intersectSegment(v_i, v_j);
  // loop over subpixel fragments
  for $k \leftarrow 1$ **to** $K - 1$ **do**
    // compute the middle point of the subpixel fragment
    $x_c = (x_k + x_{k+1})/2$;
    $y_c = (y_k + y_{k+1})/2$;
    $t_c = (t_k + t_{k+1})/2$;
    $\Delta_t = (t_{k+1} - t_k)$;
    // add contribution of fragment to the derivative
    $\dot{v}_i \leftarrow \dot{v}_i + \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} (v_j - v_i)(t_{k+1} - t_k)t_c f(\lfloor x_c \rfloor, \lfloor y_c \rfloor)$;
    $\dot{v}_j \leftarrow \dot{v}_j + \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} (v_j - v_i)(1 - t_c) f(\lfloor x_c \rfloor, \lfloor y_c \rfloor)$;

In order to make the pseudo-code 3 comprehensive, we explain some of the intermediary variables:

$$\begin{aligned} x(t) &= (1-t)x_i + tx_j = A_x[1,t]^T \\ y(t) &= (1-t)y_i + ty_j = A_x[1,t]^T \\ x(t)y(t) &= A_x y[1,t,t^2]^T \end{aligned} \tag{32}$$

$$v_d = \int_{t_k}^{t_{k+1}} t^{d-1} dt \tag{33}$$

$$\begin{aligned} c_0 &= \int_{t_k}^{t_{k+1}} t^{d-1} dt \\ c_x &= \int_{t_k}^{t_{k+1}} \epsilon(x(t)) t^{d-1} dt \\ c_y &= \int_{t_k}^{t_{k+1}} \epsilon(y(t)) t^{d-1} dt \\ c_{xy} &= \int_{t_k}^{t_{k+1}} \epsilon(x(t))\epsilon(y(t)) t^{d-1} dt \end{aligned} \tag{34}$$

After summation over all subpixel fragments of a segment we have:

$$\begin{aligned} \beta &= \int_0^1 [1,t,t^2]^T f_c(x(t),y(t)))dt \\ \gamma &= \int_0^1 [1,t,t^2]^T \nabla f c(x(t),y(t)))dt \end{aligned} \tag{35}$$

And we finally obtain.

$$\begin{aligned} \alpha_1 &= \int_0^1 f_c((1-t)v_i + tv_j)t dt \\ \alpha_2 &= \int_0^1 f_c((1-t)v_i + tv_j)(1-t) dt \\ \alpha_3 &= \int_0^1 \nabla f_c((1-t)v_i + tv_j)\ t^2 dt \\ \alpha_4 &= \int_0^1 \nabla f_c((1-t)v_i + tv_j)(1-t)^2 dt \\ \alpha_5 &= \int_0^1 \nabla f_c((1-t)v_i + tv_j)(1-t)\ t\ dt \end{aligned} \tag{36}$$

References

[1] Thomas Auzinger. *Sampled and Prefiltered Anti-Aliasing on Parallel Hardware*. PhD thesis, Technischen Universität Wien, 2015.

[2] L. Bar and G. Sapiro. Generalized newton-type methods for energy formulations in image processing. *SIIMS*, 2(2):508–531, 2009.

[3] J.E. Bresenham. Algorithm for computer control of a digital plotter. *IBM System Journal*, 4(1):25–30, 1965.

[4] M. Burger. Levenberg-marquardt level set methods for inverse obstacle problems. *Inverse Problems*, 20:20–259, 2003.

[5] E. Catmull. A hidden-surface algorithm with anti-aliasing. In *SIGGRAPH*, pages 6–11, 1978.

[6] R.C.H Cheng. Fast linear color rendering. *Graphics gems*, 3:343–354, 1992.

[7] Martin De La Gorce. *Model-based 3D Hand Pose Estimation from Monocular Video*. PhD thesis, Ecole centrale Paris, 2009.

[8] T. Duff. Polygon scan conversion by exact convolution. *Raster Imaging & Digital Typography*, 1989.

[9] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer graphics (2nd ed. in C): principles and practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[10] A. Fujimoto and K. Iwata. Jag-free images on raster displays. *Computer Graphics and Applications*, 3(9):26–34, 1983.

[11] M. Hintermüller. A combined shape Newton and topology optimization technique in real time image segmentation. In *Real-Time PDE-Constrained Optimization*, volume 3, pages 253–274. 2007.

[12] M. Hintermüller and W. Ring. A second order shape optimization approach for image segmentation. *SIAM Journal of Applied Mathematics*, 64(2):442–467, 2004.

[13] J.A. Liggett. Exact formulae for areas, volumes and moments of polygons and polyhedra. *Communications in applied numerical methods*, 4:815–820, 1988.

[14] Josiah Manson and Scott Schaefer. Analytic rasterization of curves with polynomial filters. *Computer Graphics Forum*, 32(2 PART4):499–507, 2013.

[15] M.D. McCool. Analytic antialiasing with prism splines. In *SIGGRAPH*, pages 429–436, New York, NY, USA, 1995. ACM.

[16] M.L.V. Pitteway and D.J. Watkinson. Bresenham's algorithm with grey scale. *Commun. ACM*, 23(11):625–626, 1980.

[17] M.H. Singer. A general approach to moment calculation for polygons and line segments. *PR*, 26(7):1019–1028, 1993.

[18] C. Steger. On the calculation of moments of polygons. Technical Report FGBV–96–04, Forschungsgruppe Bildverstehen (FG BV), Informatik IX, Technische Universität München, August 1996.

[19] N.J.C. Strachan, P. Nesvadba, and A.R. Allen. A method for working out the moments of a polygon. *PRL*, 11:351–354, 1990.

[20] J. Tumblin. Exact two-dimensional integration inside quadrilateral boundaries. *Journal of Graphics, GPU and Game Tools*, 11(1):61–71, 2006.

[21] X. Wu. An efficient antialiasing technique. In *SIGGRAPH*, volume 25, pages 143–152, 1991.

Algorithm 3: integrateSegmentDerivativesBilinear

Data: The polygon vertices $(v_i)_{i=1}^N$ and a discretized function $f(x, y)$

Result: The derivatives $\dot{v}_i \equiv \frac{\partial I}{\partial v_i}$ and $H_{ij} \equiv \frac{\partial^2 I}{\partial v_i v_j}$of the integral I of f_c inside the polygon with f_c the bilinear interpolation of f

// loop over edges of the polygon

for $i \leftarrow 1$ **to** N **do**

$j = mod(i, N) + 1$;

$\beta \leftarrow 0_{3\times 1}$; $\gamma \leftarrow 0_{3\times 2}$

$(p_k, t_k)_{k=1}^K \leftarrow$ intersectSegment(v_i, v_j);*// clip the segment* $\overline{v_i, v_j}$ *on the pixel grid*

// loop over subpixel fragments

for $k \leftarrow 1$ **to** $K - 1$ **do**

$x_c = (x_k + x_{k+1})/2$; $y_c = (y_k + y_{k+1})/2$; $t_c = (t_k + t_{k+1})/2$;

$\Delta_x = x_{k+1} - x_k$; $\Delta_y = y_{k+1} - y_k$; $\Delta_t = t_{k+1} - t_k$;

$A_x \leftarrow [\epsilon(x_c) - (\Delta_x t_c/\Delta_t),\ \Delta_x/\Delta_t]$; *// see eqn.32*

$A_y \leftarrow [\epsilon(y_c) - (\Delta_y t_c/\Delta_t),\ \Delta_y/\Delta_t]$;

$A_{xy} = [A_x(1)A_y(1),\ A_x(1)A_y(2) + A_x(2)A_y(1),\ A_x(2)A_y(2)]$;

for $d \leftarrow 1$ **to** 5 **do**

$v_d = ((t_{k+1})^d - (t_k)^d)/d$; *// see eqn.33*

$f_{00} \leftarrow f(\lfloor x_c \rfloor, \lfloor y_c \rfloor)$; $f_{01} \leftarrow f(\lfloor x_c \rfloor, \lfloor y_c \rfloor + 1)$;

$f_{10} \leftarrow f(\lfloor x_c \rfloor + 1, \lfloor y_c \rfloor)$; $f_{11} \leftarrow f(\lfloor x_c \rfloor + 1, \lfloor y_c \rfloor + 1)$;

$b_0 \leftarrow f_{00}$; $b_x \leftarrow f_{10} - f_{00}$; $b_y \leftarrow f_{01} - f_{00}$;

$bxy \leftarrow f_{11} - f_{10} + f_{00} - f_{01}$;

for $d \leftarrow 1$ **to** 3 **do**

$c_0 \leftarrow v_d$;

$c_x \leftarrow A_x[v_d, v_{d+1}]^T$; *// see eqn.34*

$c_y \leftarrow A_y[v_d, v_{d+1}]^T$;

$c_{xy} \leftarrow A_{xy}[v_d, v_{d+1}, v_{d+2}]^T$;

$\beta_d \leftarrow \beta_d + b_0 c_0 + b_x c_x + b_y c_y + b_{xy} c_{xy}$; *// see eqn.35*

$\gamma_{[d,:]} \leftarrow \gamma_{[d,:]} + b_{xy} c_y + [b_x, b_y]^T c_0$;

$\alpha_1 \leftarrow [0, 1, 0]\beta$; *// see eqn.36*

$\alpha_2 \leftarrow [1, -1, 0]\beta$; $\alpha_3 \leftarrow [0, 0, 1]\gamma$; $\alpha_4 \leftarrow [1, -2, 1]\gamma$;

$\alpha_5 \leftarrow [0, 1, -1]\gamma$; $J \leftarrow \left[\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right]$;

$\dot{v}_j \leftarrow \dot{v}_j + J(v_j - v_i)\alpha_1$;

$\dot{v}_i \leftarrow \dot{v}_i + J(v_j - v_i)\alpha_2$;

$H_{ii} \leftarrow H_{ii} - J\alpha_2 + J(v_j - v_i)\alpha_4$;

$H_{jj} \leftarrow H_{jj} + J\alpha_1 + J(v_j - v_i)\alpha_3$;

$H_{ji} \leftarrow J\alpha_2 + J(v_j - v_i)\alpha_5$;

$H_{ji} \leftarrow H_{ji}^T$;