Infinite Resolution Textures

Conference Paper · June 2016

CITATIONS
0

READS
1,086

2 authors, including:

Alexander Reshetov
NVIDIA
19 PUBLICATIONS **410** CITATIONS

SEE PROFILE

High Performance Graphics (2016)
Ulf Assarsson and Warren Hunt (Editors)

Infinite Resolution Textures

Alexander Reshetov and David Luebke

NVIDIA

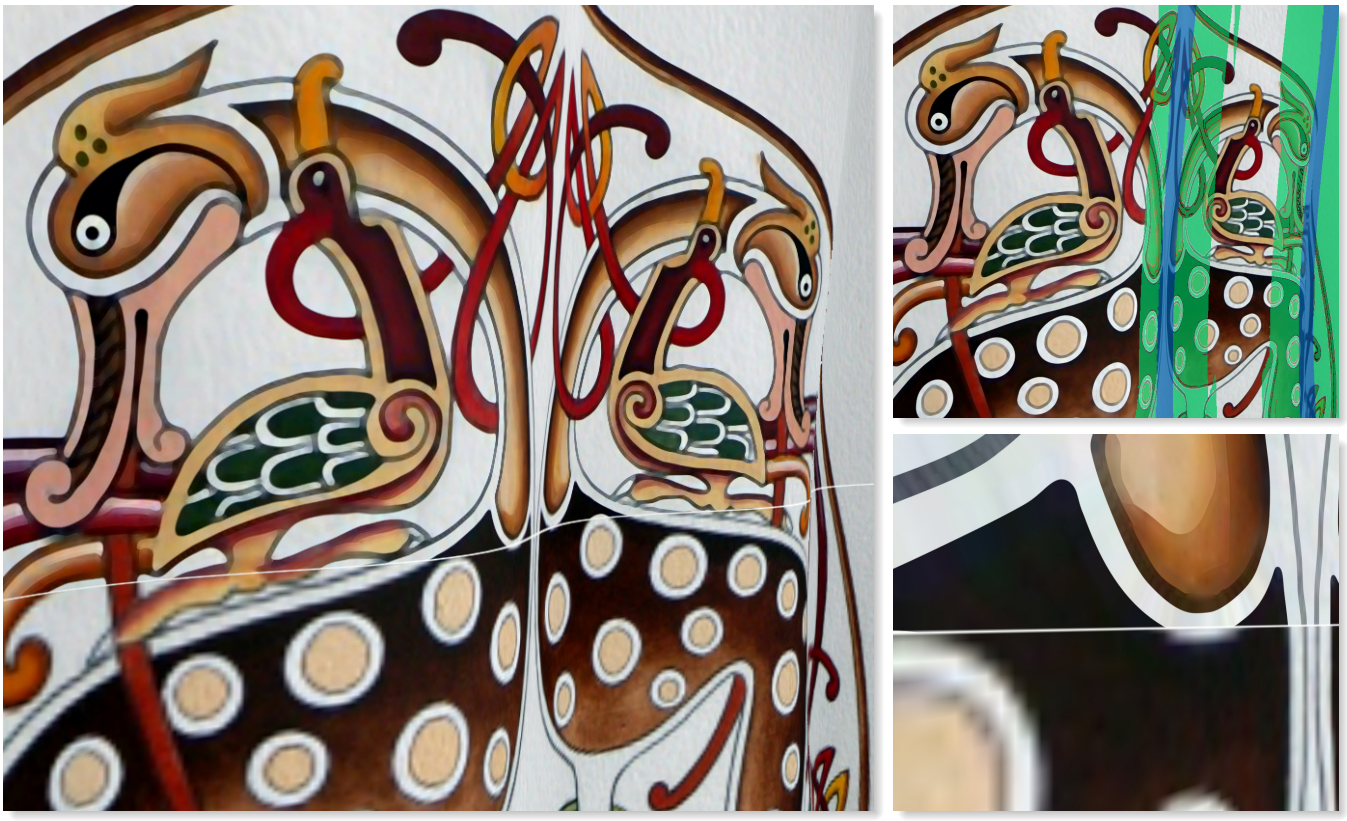


Figure 1: *left: the new IRT sampler (top) and a traditional raster sampler (bottom). The texture is mapped onto a waving flag. Right top: green color indicates areas where the IRT sampler is effectively blended with the raster sampler; in blue areas only the raster sampler is used. Right bottom: close-up. IRT chooses the sampler dynamically by analyzing texture coordinate differentials. In all cases, only a single texel from the raster image is fetched per pixel. The original texture resolution is 533×606. The IRT sampling rate is about 6 billion texels per second on an NVIDIA GeForce GTX 980 graphics card. The image is a photograph of the airbrush painting "Celtic Deer" © CelticArt.*

Abstract

We propose a new texture sampling approach that preserves crisp silhouette edges when magnifying during close-up viewing, and benefits from image pre-filtering when minifying for viewing at farther distances. During a pre-processing step, we extract curved silhouette edges from the underlying images. These edges are used to adjust the texture coordinates of the requested samples during magnification. The original image is then sampled — only once! — with the modified coordinates. The new technique provides a resolution-independent image representation capable of billions of texels per second on a mid-range graphics card.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.3]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; Computer Graphics [I.3.3]: Picture/Image Generation—Antialiasing; Image Processing and Computer Vision [I.4.3]: Enhancement—Sharpening and deblurring

1. Introduction

Graphics applications such as games combine 3D geometric data and 2D textures. These assets behave differently under scaling. The 3D data represents the shape of the objects in a scene and as such can be sampled at any resolution. 2D textures are typically used to represent material properties and minute geometric details, and have a limited resolution defined by the underlying image. Unlike 3D models, textures can be easily pre-filtered at multiple coarse levels and stored in a mipmapped format. This is a significant advantage, as it allows integrating over all subsamples in a pixel by issuing a single texel fetch. Using trilinear mipmapping [Wil83], the level of detail can vary smoothly, so when a computer graphics object moves away from the viewpoint, its texturing will change gradually from the fine to the coarse levels. Unfortunately, when we zoom in on such an object, the pixels in even the highest-detail texture image will eventually become larger than the pixels on the screen. When this happens we will see a familiar blotchy structure of the texture and the image will become over-blurred.

Ironically, this wasn't always the case. The earliest 2D computer graphics were all based on such geometric primitives as straight and curved line segments, directly rendered on vector displays. The wide use of discrete raster images came later, supported by the rapid development of raster displays and hardware texture samplers.

Vector graphics continue to proliferate in areas in which quality approximation is not acceptable, such as professional graphics. This includes, in particular, illustration and computer-aided design. Vector graphics allow storing data in a resolution-independent format which can then be rendered on any device or printed as a hard copy. Most vector graphics formats (such as PostScript or SVG) can be treated as programs prescribing the process of rendering an image composed of (potentially overlapping) geometric primitives. For this reason, computing a color at a single position might necessitate executing the whole program. This is not an issue in professional applications when the whole image emerges as a result of executing such a program, but it makes vector graphics a less efficient substitute for texture assets in games, where often only a portion of an image is accessed every frame and samples are irregularly distributed.

Vector graphics formats, considered as programs, tend to be sequential in nature. This hindered their hardware optimization, until the groundbreaking work of Loop and Blinn [LB05], Kilgard and Bolz [KB12], and Ganacim et al. [GLdFN14]. Kilgard and Bolz introduced a two-step "Stencil, then Cover" approach, allowing efficient GPU rendering of vector textures as a whole. Ganacim et al. went further, employing an acceleration structure whose traversal enabled rendering parts of the image.

Human visual perception relies on the ability to detect edges [Sha73] and most vector formats store and process silhouette edges natively. Vector graphics is also well-suited for close-ups, providing a theoretically infinite resolution. Yet rendering such images at a distance is superfluous as multiple primitives overlap the same pixel. In principle, it is possible to pre-render vector graphics into a sequence of raster images at decreasing resolution and then blend vector and raster samples together, as suggested by Ray et al. [RCL05]. However, this is still rather wasteful and can also exhibit ghosting.

Instead, our approach (named "Infinite Resolution Textures" — IRT) unifies vector and raster representations by always computing the resulting color through a single texel fetch as

```
float4 c = tex.SampleLevel(s, uv+duv, lod);
```
(1)

This HLSL example shows how an application would ordinarily fetch the color `c` from the texture `tex` using the sampler `s`, except for the texture coordinate adjustment `duv`. IRT computes `duv` to produce crisp silhouette edges at close distances. When moving away from an object, this adjustment is scaled back until it completely vanishes, decreasing the `duv` vector to zero magnitude. At this point, the IRT color is simply the conventional mipmapped raster color, taking advantage of the minification at the level of detail `lod`.

Our goal is to sample a color at a position that is *a*) close to the sample `uv` but *b*) farther away from any silhouette edge than a given distance (of a few pixels). In other words, we want to move the sample outside of the blurred area around the curves, but do it conservatively. This will not create any new image details, but, hopefully, sidestep the limitations of a fixed resolution of the original image. It is similar to the existing image deblurring techniques but executed on demand at run-time with just a small performance overhead.

We compute `duv` by accessing the curved silhouette data in the neighborhood of the sample point `uv`. These records are typically shared among multiple subsamples in the neighborhood of the curve, allowing a good memory cache utilization (section 6.2).

The stored curved silhouettes are either given, if the underlying image is provided in vector format, or have to be computed from the underlying raster image. There are many approaches to edge detection in raster images; we will describe one that is well suited for our purposes. Our ultimate goal is an efficient way to increase the resolution of a broad class of available texture assets, suitable for a drop-in replacement in games and 3D applications.

The texture resampling was first used in the *pinchmaps* proposed by Tarini and Cignoni [TC05]. A similar approach was latter exploited for antialiasing [Res12] and super-resolution [JP15]. We compare our implementation with pinchmaps in section 2.1.

2. Related Work

The research community has long recognized the need for a resolution-independent texture representation that allows real-time sampling. This need can be directly addressed by devising ways to efficiently sample the existing vector formats such as SVG or PostScript. A significant corpus of work exists in this area, for a comprehensive review refer to the specialized publications [KL11, SXD*12, KB12, GLdFN14, BKKL15].

Approaches aspiring to represent a broader class of genuine raster images include *bixels* [TC04], *silmaps* [Sen04], *pinchmaps* [TC05], and *Vector Texture Maps (VTM)* [RCL05]. A VTM decomposes texture space into different regions delineated by a set of implicit cubic polynomials. Each region can be sampled by a different fragment shading function. Antialiased filtering is done for pixels straddling the borders of such regions by computing blending coefficients for the two colors returned by the shaders at the each side of the discontinuity. Bixels use the similar strategy by decomposing the texture plane into addressable tiles with straight boundary segments and using supersampling for antialiasing.

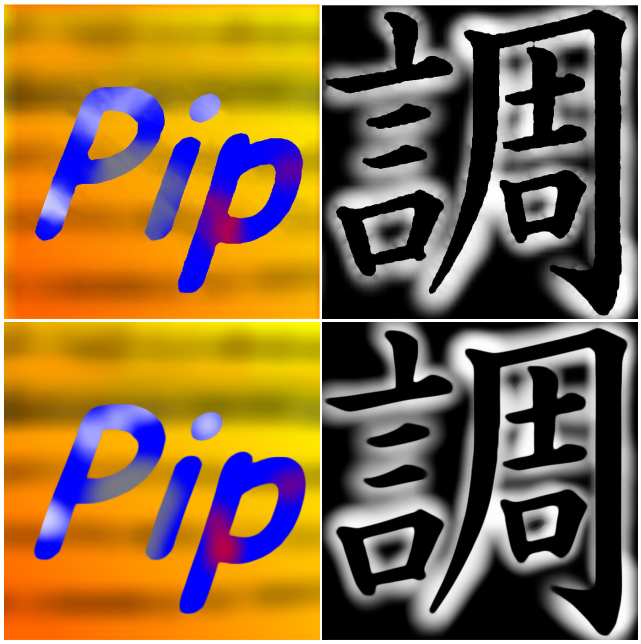


Figure 2: *top: two pinchmap images [TC05]; bottom: IRT rendering of these images retrieved from the pinchmaps paper at 256×256 resolution.*

Silmaps were originally proposed to fight undersampling artifacts in shadow maps by constructing a piecewise-linear approximation to the true shadow [SCH03]. This work was latter extended to general textures [Sen04] and volumetric data [KHPS07]. Silmaps eliminate blurring during texture magnification by custom-filtering colors on the same side of the discontinuity. The algorithm supports six possible configurations of the linear silhouette edges inside a pixel.

Diffusion curve textures [OBW*08] employ a different way to achieve a resolution-independent texture representation by construing an image as a solution of radiative heat transport equation. Sun et al. [SXD*12] proposed to use an explicit form of such solution for closed diffusion curves as a sum of Green's functions. Such reduction allows a random access to the color of each texel (a few milliseconds per 1M texels in a typical image).

Observing that infinite resolution of vector textures is rarely required, Song et al. [SWWW15] introduced a variable resolution scheme in which an image is first partitioned into a kd-tree. Each leaf node is then compressed with an acyclic feed-forward neural network. Such representation by *vector regression functions (VRF)* allows very fast random access and filtering. VRF approximation can cause artifacts, in particular sharp features will appear as smooth gradients in close-ups. This behavior might still be more visually appealing than traditional pixelization artifacts.

2.1. Comparison of IRT with Prior Art

Pinchmaps [TC05] employ the same approach as IRT by resampling the original raster image near the edges. A single quadratic silhouette edge per pinchmap texel is reconstructed by bilinearly interpolating the four corner values that describe the edge orientation and position. These values are then used to compute the texture coordinate offsets that are thence limited by a pinchmap texel size.

This method is attractive in its simplicity, always executing a single auxiliary fetch (from a 4-channel pinchmap) for each texel. However, it does not allow edge intersections and reconstructs only a single silhouette per pinchmap texel. Even more limiting, the pixels that are not intersected by edges will have `uv` offset equal to 0 by design — even if there are edges in close proximity passing through the neighboring pixels. Accordingly, the texture coordinate offsets across the pixels with zero and non-zero pinches will be discontinuous. This creates artifacts that cannot be easily avoided (Figure 2).

We address these problems by storing multiple contributing silhouette edges per texel and not limiting the magnitude of the texture coordinate adjustment. This allows better image reconstruction and smoother silhouettes (third order) at the cost of the more complicated data structures (section 5.6).

In comparison with pinchmaps, silmaps [Sen04] can represent silhouette intersections. This is achieved by considering piecewise-linear segments with some mild restrictions on the edge topology (one silhouette edge per pixel side). Silmaps produce crisp edges by always interpolating colors on the same side of the edge. To avoid fetching four corner colors that are called for by a straightforward implementation of this custom interpolation scheme, a clever strategy is proposed that uses a single bilinear texel lookup for 1, 2, and 4-corner cases and needs three lookups only for the 3-corner case. This compares favorably with bixels [TC04], in which a sample is evaluated by a one of the proposed ten *patch functions* that might necessitate fetching all colors at a patch boundary.

However, these custom-made interpolations make mipmapping more complicated, requiring an explicit color blend in a shader. An edge antialiasing would also require an extra work, making multiple texel lookups unavoidable.

In contrast, pinchmaps allow natural antialiasing and mipmapping by scaling the texture coordinate offset and we also employ a similar strategy (section 4.2). The difference between pinchmaps and IRT in this respect is that the expressive power of pinchmaps is somewhat diluted by limiting the maximum magnitude of this adjustment; this also introduces additional frequencies pertinent to the resolution of the pinchmap.

All these algorithms require a pre-processing step. Pinchmaps are extracted from either a vector-based representation or a high-resolution raster image. At run-time, the appropriately downsampled raster image is used as well. The same scheme would work for silmaps, except that for the better results the accompanying raster image would require special processing to recover distinct non-blurred colors on both sides of any stored silhouettes (to facilitate custom-made interpolation at run-time). This makes this algorithm better suited for man-made images with pronounced edges.

Due to its less restrictive format (multiple curves per pixel, user-controlled offsets), IRT might also be suitable for some natural images as well (see Figures 5, 14, 12, 13). To facilitate this functionality, we have designed an edge detection and smoothing scheme that converts a single raster image to a coordinated vector format (sec-

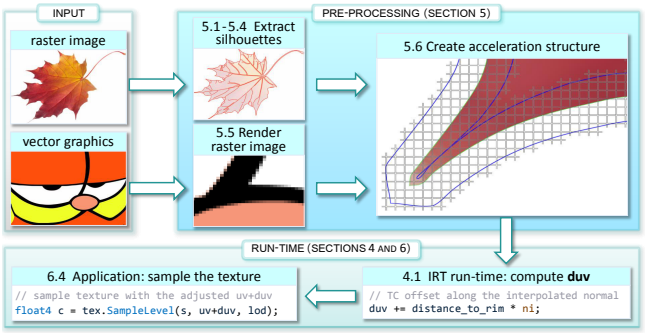


Figure 3: *IRT flowchart.*

tion 5). We conjecture that other texture magnification approaches would benefit from this functionality as well.

3. Algorithm Outline

Figure 3 shows the IRT data flow, with corresponding paper sections.

To compute the texture coordinate adjustment `duv` in (1), we need both a raster image and its silhouette edges. When starting from a raster image, those silhouettes have to be determined from the image; our approach to this task is described in sections 5.1 – 5.4. Conversely, vector images do not have paired *raster* images as such, so one has to be rendered from the vector format at the appropriate resolution (section 5.5).

We use a grid acceleration structure to store the relevant curved silhouettes in the neighborhood of a given `uv` query (section 5.6). Note that a grid might not be an ideal structure, especially when different parts of an image have vastly contrasting scales (a 2D equivalent of the "teapot in a stadium" problem). We plan to continue exploring possible alternative approaches in this regard.

We begin by explaining our chosen way of computing the `uv` $\mapsto$ `duv` mapping, since this is at the core of IRT (section 4.1). This mapping is continuous everywhere except on edges. To avoid aliasing near the edges, we must handle such samples differently; two pertinent approaches are described in section 4.2.

Finally, we discuss performance (section 6.1) and limitations of the technique (6.3), then conclude with potential application areas and future work (6.4).

4. Run-Time Computations

All algorithms in this paper use the following two parameters:

- σ — the maximum distance in the RGB color space between similar pixels; we use $\sigma = 25$ for all images in this paper (for 8-bit RGB colors).
- h — an assumed size of a convolution kernel that distorts the colors around the edges; we use $2\sqrt{2}$, which is the length of the two pixel diagonals. (2)

A sample IRT implementation is provided in Appendix A. IRT uses three types of values to compute the texture coordinate adjustment `duv` in the statement (1):

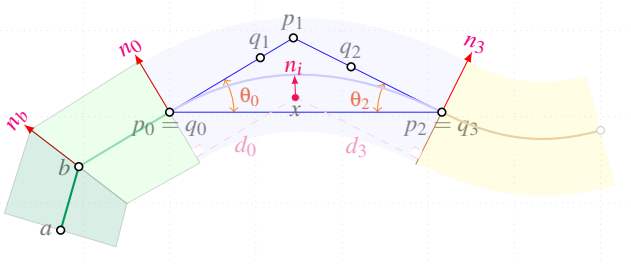


Figure 4: *a continuous silhouette consisting of two straight and two curved segments. Color shaded regions demarcate areas of influence of each segment. At shared vertices (b, q_0, q_3), bisectors split the angle between two adjacent tangent lines, separating the corresponding truncated Voronoi regions.*

- level of detail `lod` for the given texture coordinates `uv`.
- distances `dc` to all the silhouette edges closer to the sample than the supplied distance h above.
- vectors `ni` that point away from the found silhouettes. (3)

To handle a broad class of images, we use smooth curves as silhouettes. Each curve is split into curved segments and we create truncated Voronoi regions for such segments delineated by curve's normals at its endpoints (Figure 4).

Our algorithm will work with any curve representation (including a piecewise linear format) for which there is an efficient way of computing the distance `dc`. Since IRT can use any curve format and it is not essential for the algorithm itself, we describe our implementation later in section 5.

For mipmapped textures, `lod` is usually computed as `log2(pixratio)`, using the ratio of a pixel size to a texel size, though IRT does not depend on it. We also use `reduce` factor to smoothly fuse vector and raster textures by scaling the value of `duv` (line 15 in Appendix A). For `pixratio` ≥ 1, i.e., when pixels become bigger than texels, we immediately return the mipmapped color, so there is a very little overhead during minification.

Distance to the curve `dc` is a continuous function of `uv` coordinates, but the curve's normal at the closest point is not (and there could be multiple such points). Since we approximate the distance `dc` (section 5.3.2) — and we want continuously sampled colors outside the curves — we can use an approximate "normal" as a value of `ni`. The simplest solution is to employ a 2D version of Phong normal [Pho75] by interpolating two normals defined by the side edges of the curve's Voronoi region. In Figure 4, these are normals n_0 and n_3 for the curve $q_{0,1,2,3}$. For the interpolation weights, we use distances to these edges since we already compute them during a clipping step (lines 36 and 39). By design, such an interpolated normal will vary smoothly when moving across the side edges (i.e., between the two adjacent Voronoi regions).

4.1. Computing the Adjustment for Texture Coordinates

We want to conservatively move the sampling position `uv` (in the statement 2) away from the edges. In a simplest possible implementation, we just move by `duv = (h - dc) * ni` along the interpolated normal `ni` for the closest curve. If the sample is overlapped by multiple areas of influence (x in Figure 11), we sum all such adjustments,

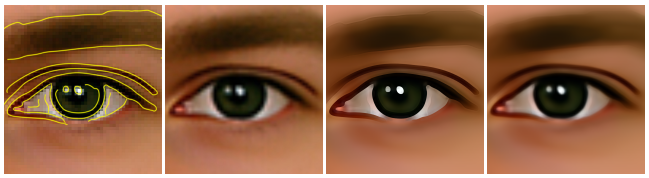


Figure 5: *from left to right: raster image with the detected edges; bilinearly interpolated image; crisp edges; smoothed out edges.*

given that those overlaps could only happen for samples at sizable distances from curves where any adjustment is small anyway.

Only curves that are closer to the sample than `h` — and not occluded by the other curves — influence `duv` computations. This assures a continuous function of `uv` coordinates, i.e., small changes in `uv` result in small changes in `duv`, unless `uv` passes through the curve. If edges of a Voronoi region were reduced to avoid overlaps with other such regions (as in Figure 7g), we scale down the value of `h` accordingly. This ensures that the new sampling position will be inside the corresponding Voronoi region.

This will create crisp edges. Depending on an application area or an image at hand, this might not be a desirable outcome. Instead of a strict linear `dc` ↦ `duv` mapping, we could employ a different scheme, e.g., smoothing out areas near the edge similar to Optical Low Pass Filter (OLPF). A few such possible alterations are provided in comments to the code in Appendix A, see lines 61–65, 89, and 104. In Figure 5, the result of such OLPF-style filtering is shown on the right (the only change to the shader is to uncomment line 65).

4.2. Antialiasing Options

Due to the discontinuous nature of `uv` ↦ `uv+duv` mapping at edges, antialiasing is a requisite part of IRT.

The most straightforward approach is to reproduce the behavior of an analytical box filter for all pixels closer to any curve than the screen pixel size. To detect this, we convert `dc` — that is computed in texture space — to screen space using screen-space derivatives of the texture coordinates as shown in Appendix A, line 6. We then blend two colors, each one fetched with its own `duv`. One vector is computed by the algorithm directly (lines 50 or 55), another by negating the signed distance to the closest curve (i.e., considering the mirrored sample with respect to the edge). Two blending weights at line 80 approximate areas of the pixel split by the edge.

This approach causes just a small overhead for the affected pixels by requiring two color fetches. Still, despite being at the core of the hardware texture filtering, box filters are subpar in removing aliasing, especially in a temporal domain. A detailed discussion is carried out by Ganacim et al. [GLdFN14].

A better solution would require convolving resampled colors with a data-dependent smoothing kernel, a feat that might look unattainable with our chosen mode of operations. Yet, we can approximate the *result* of such convolution with just a single color fetch. The key idea is to use a higher LOD mipmap that already contains pre-filtered colors.

A dashed green line in Figure 6e shows the length of the vector `duv = (h - dc) * ni` as a function of the distance to the curve. It is

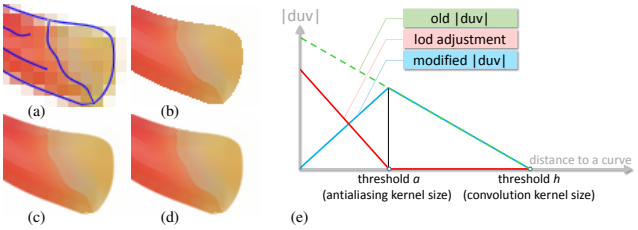


Figure 6: *(a) original image (as in Figure 3) with the detected curved edges; (b) IRT without antialiasing; antialising with two (c) and one (d) fetches; (e) the modified ‖duv‖ profile used for the antialiasing with a single fetch (section 4.2).*

a monotonically decreasing function with a maximum at `dc = 0`. Vectors `duv` for the two close samples on the different sides of an edge will have the same magnitude but opposite directions — this is why we actually need antialiasing for such samples.

One way to dampen such discontinuity is to impose a ∧-shaped profile for ‖`duv`‖ function (blue line), but this will not solve aliasing problems since color gradients in the immediate neighborhood of an edge are at local maximum as a matter of choice. This can be mitigated though by increasing `lod` for samples that are close to an edge (red line). By adjusting the position of the threshold at which this adjustment is effected, we can create edges with different levels of blurriness (line 72).

In Figure 6(cd) these two antialiasing modes are shown side-by-side.

5. Preprocessing

IRT accepts both vector and raster images as an input. After the preprocessing stage, the distinction between these two categories disappears and at run-time we use a uniform acceleration structure, as shown in Figure 3.

To extract curved silhouettes from a raster image, we pursue a three-prong approach to

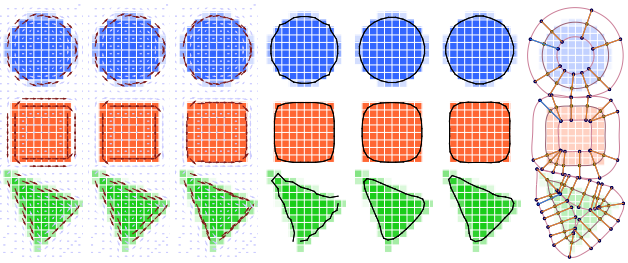
1. find pixels that lie on an edge (section 5.1),
2. connect such pixels (5.2), and
3. convert the connected sequences to either Bézier curves (5.3) or rational polynomials (5.4).

(4)

These goals can be achieved by using existing software, such as Inkscape [Bah07] or several online tools including Adobe Creative Cloud® [SOT13]. Some of these systems aim at an artistic image depiction with exaggerated edges; others target a faithful image conversion, albeit in a restricted context (Computer Aided Design). Since we seek a representation of a wide range of images used in computer graphics, we opted for our own silhouette detector.

5.1. Finding Edge Pixels

Edge detection is one of the core problems in computer science. A multitude of approaches exist, targeting different applications with disparate requirements. Several researchers have reviewed the state of the art [MA09, PP11, SC12, MSH12].



(a) (b) (c) (d) (e) (f) (g)

Figure 7: *From left to right: improving the quality of the found edges. (a) tangent lines (orthogonal to gradients) with chosen edges in dark color; (b) tie breaking; (c) increasing sub-pixel accuracy through parabola fitting; (d) it still will result in wavering silhouettes, unless we apply (e) 3-tap or (f) 5-tap filter; (g) curved edges with the corresponding Voronoi regions.*

Maini and Aggarwali [MA09] show that the Canny edge detector [Can86] performs better than most other straightforward techniques under almost all scenarios. IRT generally follows the steps of the Canny edge detector. We will briefly describe our implementation, emphasizing the additional steps we introduced to smooth out the found continuous edges.

Gradient detection. We use RGB colors, following the ideas of Novak and Shafer as described by Kanade in [Kan87]. This technique requires computing the 3×2 Jacobian J of the partial derivatives along x and y directions for each RGB component and then finding the largest eigenvalue of 2×2 matrix J^TJ. The square root of this eigenvalue gives the edge strength and the corresponding eigenvector — its normal (with $gradient = strength * normal$). To compute partial derivatives, we use the Scharr operator [JSK99] due to its improved rotational symmetry.

Non-maximum suppression. We elected to use a continuous version of the edge thinning step by invalidating all pixels for which the edge strength is less than either of the two values sampled in the positive or negative gradient direction.

These steps are illustrated in Figure 7. Initially, we compute the gradient at all pixels, but only some of them survive non-maximum suppression (a). In rare cases, such as the red box in the middle, colors will have an axial symmetry in some local neighborhood. This would result in detecting edges on both sides of the axis of symmetry, or (which is more likely due to numerical errors) randomly choosing edges on both sides. This ambiguity can be resolved by slightly decreasing gradients for brighter pixels (b).

Edge positions can be tuned by fitting a parabola to the three strength values that were used for the non-maximum suppression (c). Once we connect the found edge pixels (section 5.2), we can further smooth out these curves by applying (e) 3-tap or (f) 5-tap filter. We found that the simplest 5-tap filter given by $weights = [-1,3,6,3,-1]/10$ is quite adequate for our purposes. This is comparable with $weights = [-2,5,10,5,-2]/16$ proposed by Dyn et al. [DLG87] except for somewhat reduced weights for the two farthermost points. More evolved topics are discussed by Nehab and Hoppe [NH12].

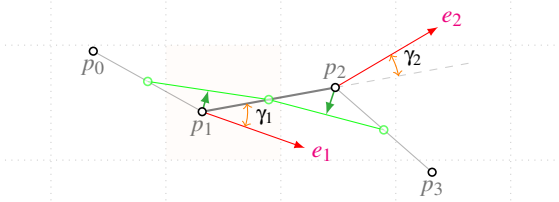


Figure 8: *choosing the best connections by maximizing the similarity measure (5). The result of a 3-tap filterig is shown as a green curve.*

5.2. Connecting Edge Pixels

So far, we found points that we believe should lie on a silhouette, and the plausible silhouette orientation. We will refer to these values as *expected* to differentiate them from the *adjusted* positions and directions. Now we have to connect such points.

Our goal is to determine two neighbors for each edge pixel in such a way that directions to these neighbors will not differ too much from the expected edge orientation. To facilitate a possible parallel execution, we adopted the following procedure (which is somewhat different from the traditional hysteresis thresholding).

Using notation in Figure 8, for each edge pixel p_1 and edge pixel p_2 in its immediate neighborhood, we compute the similarity measure

$$m = \frac{(\cos\gamma_1 + |\cos\gamma_2|)}{\|p_2 - p_1\|}\sqrt{s_2} \tag{5}$$

where γ_1 and γ_2 are angles between $p_2 - p_1$ and the expected tangent vectors e_1 and e_2; s_2 is the strength of the edge at the pixel p_2.

We prefer smaller angles γ_1 and γ_2 and also prefer the adjusted silhouette to go through the stronger pixels that are closer to p_1. We compute the maximum of m (ignoring negative values). It will yield us the best candidate pixel to continue the silhouette in the positive direction of e_1. Together with a connection in the negative direction $-e1$, we will have 2 candidate connections for pixel p_1. If p_1 is in turn among the candidate connections for p_2, we join these two pixels. This will give us sequences of the connected pixels.

Once all the sequences are found, we eliminate those for which the average magnitude of the gradient is less than the predefined parameter σ in (2). This is a more relaxed form of the hysteresis thresholding, aimed at finding longer sequences and avoiding the sequential nature of the traditional approach (start at pixels with the gradient magnitude $> \sigma$ and keep connecting the points until the magnitude drops below $q\,\sigma$ where $q < 1$).

5.3. Extracting Bézier Curves

Cubic Bézier curves are used in most vector graphics formats. It makes sense to approximate the found sequences of edge points with such curves in like manner. Our goal is to reduce the overall number of the used curves while keeping the approximation error below the given threshold. We also enforce an additional linearization constraint to allow using implicit representation of a Bézier curve as a proxy to a distance to such curve (see section 5.3.2).

5.3.1. Bézier Curve Implicitization

We compute the texture coordinate adjustment `duv` in Equation (1), using distances to all curves in the neighborhood of sample `uv`. A direct approach requires solving an equation of fifth degree. Approximate algorithms [LB05, NH08] compute the distance with an error vanishing at the curve. We could use either technique, but we took another approach pursuing the lowest possible execution complexity.

Following Floater [Flo95], for each planar cubic Bézier curve with control points $q_{0,1,2,3}$ (Figure 4) we define λ_1 and λ_2 such as $q_1 = (1-\lambda_1)p_0 + \lambda_1 p_1$ and $q_2 = (1-\lambda_2)p_2 + \lambda_2 p_1$ and

$$\begin{aligned} &\alpha_i = 3(1-\lambda_i),\ \beta_i = 3\lambda_i,\ \phi_i = \beta_i - \alpha_k\beta_k,\ i = 1,2,\ k = 2,1 \\ &A = -\beta_1^2\phi_1,\ C = -3\beta_1\beta_2 + 2\beta_1^2\alpha_1 + 2\beta_2^2\alpha_2 - \beta_1\beta_2\alpha_1\alpha_2 \\ &B = -\beta_2^2\phi_2,\ D = \alpha_2\phi_1,\ E = \alpha_1\phi_2,\ F = 1 - \alpha_1\alpha_2 \\ &f(x,y) = A\tau_0^2\tau_2 + B\tau_0\tau_2^2 + C\tau_0\tau_1\tau_2 + D\tau_0\tau_1^2 + E\tau_1^2\tau_3 + F\tau_1^3 \end{aligned} \tag{6}$$

For any point $[x,y]$, we compute its barycentric coordinates $\tau_{0,1,2}$ for the Bézier triangle $p_{0,1,2}$. The function f is an implicit representation of the Bézier curve ($f \equiv 0$ at the curve). At run-time, computing f is just slightly more expensive than computing two dot products, see the fragment shader example in Appendix A (line 50).

5.3.2. Distance to Bézier Curve

Floater proves that f in equation (6) is unique inside the Bézier triangle $p_{0,1,2}$ (i.e., equal to 0 only on the curve) if and only if $\phi_1\phi_2 > 0$. We want to be able to handle texels inside (the bigger) curve's Voronoi region, so this result is not directly applicable. We scale coefficients $A-F$ so that f approximate the distance to the curve for all such texels. Using $f/|\nabla f|$ would result in a better approximation, but is more expensive.

To analyze typical errors of such approximation, we consider a plurality of Bézier curves with equispaced control points. Up to a uniform scaling, rotation, and translation, such curves are defined by two angles $\theta_0 = \angle p_1p_0p_2$ and $\theta_2 = \angle p_1p_2p_0$ in Figure 4, reducing a search space. We found that if such angles are smaller than 0.6, the error will not exceed 0.1 inside the Voronoi region with sideway edges equal to $2\|q_0 - q_3\|$. In turn, when we fetch the color with the adjusted `duv`, it will result in color differences not exceeding 10% of 255 in agreement with the supplied σ in (2).

Each straight segment can be treated as a cubic Bézier curve with q_1 and q_2 lying on $[q_0, q_3]$ and we intend to do so to minimize the code divergence. Yet, it would result in infinite barycentric coordinates, requiring a division by zero (area of the triangle $p_{0,1,2}$). Since f is a uniform polynomial, we can avoid this problem by dropping the normalization requirement ($\tau_0 + \tau_1 + \tau_2 \equiv 1$). We can also directly bake-in the distance scaling into $A-F$ coefficients in equation (6) arriving at a very simple way of computing $\tau_{0,1,2}$ using the two dot products with the Bézier triangle edges (line 46 in Appendix A), followed by the distance-to-the-curve computation in line 50.

5.3.3. Fitting Bézier Curves To Point Sequences

The results of the previous section give us a simple way to fit sequences of points to Bézier curves. First, at each point we find a direction of a tangent line by averaging unit vectors to two adjacent points. Then, starting with the first point in a sequence, we traverse the connected points until either of $\theta_{0,2}$ exceeds $0.1 radian \approx 34°$ or $\phi_1\phi_2 \le 0$.

The angles $\theta_{0,2}$ are uniquely identified by the tangent lines and $q_{0,3}$ endpoints which we acquire from the sequence; the exact position of $q_{1,2}$ on the tangent lines is computed by requiring a uniform split of the interval $[q_0, q_3]$.

This approach creates smoothly connected curve segments (with geometric G1 continuity) that are neither too big nor too small. However, situations could arise when an angle between the two vectors toward the neighbors is smaller than $180° - 2*34° = 112°$. In that case we will create a straight segment (see two left segments in Figure 4). We also modify a tangent line to ensure smoothly connected *straight-curved* segments, so the only non-smooth (G0) connections are at *straight-straight* joints.

5.4. Using Rational Curves

A cubic polynomial is a bad choice for approximating a linear distance far off the curve. To remedy this situation, we will consider a quotient of two multivariate polynomials. Floater theory holds for rational Bézier curves, but we instead derive a suitable representation from first principles using variables that are natural to the problem at hand and easy to compute. We consider (Figure 4)

$$\begin{aligned} &d_{0,3} \text{ – signed distances to the edges of curve's Voronoi region} \\ &d_n \text{ – distance from a sample } x \text{ to the edge } q_{0,3} \\ &\quad \text{along the interpolated direction } n_i \end{aligned} \tag{7}$$

We compute $n_i = t_3 n_0 + t_0 n_3$ by interpolating normals to the curve at endpoints $q_{0,3}$ with weights $t_{0,3} = d_{0,3}/(d_0 + d_3)$. Variables $d_{0,3}$ can also be used to verify that a sample is inside the region (if and only if $d_{0,3} \ge 0$). We intend to use (non-normalized) interpolated vector n_i for `duv` adjustment since the modified texture coordinates always stay inside the Voronoi region delineated by $q_0 + t\,n_0$ and $q_3 + t\,n_3$ lines.

The sought-after implicit function f should be 0 at $q_{0,3}$ and its gradient at these points should be collinear with normals $n_{0,3}$. These Hermite conditions guarantee a smooth stitching of silhouette segments by restricting numerical values of coefficients of given polynomials.

For a variety of rational polynomials over $d_{0,3,n}$ (and a few other feasible variables) we eliminated those that cannot satisfy Hermite conditions. We then sorted out the remaining polynomials by how well they approximate random Bézier curves. One representation stands out. For better insight, we will deduce this representation by examining — and exploiting — its desired properties.

We want f to behave like $O(d_n)$ afar from the curve. Without loss of generality, $f = d_n + g(d_0, d_n, d_3)\,t_0 t_3$. Both these terms equal to 0 at $q_{0,3}$, so we only need f to satisfy the second Hermite condition. We will be interested in the simplest form of g that also has nice asymptotic properties at a distance from the curve (vary like $O(d)$).

The simplest such function that is also symmetric with respect to its variables is $g = a_3 d_0 + a_0 d_3$. The second Hermite condition leads to $a_0 = tan(\theta_0),\ a_3 = tan(-\theta_3)$.

We have used all degrees of freedom (available coefficients) to satisfy Hermite conditions. In Figure 9a, a yellow unit circle is approximated by four such curves with the Hausdorff distance between

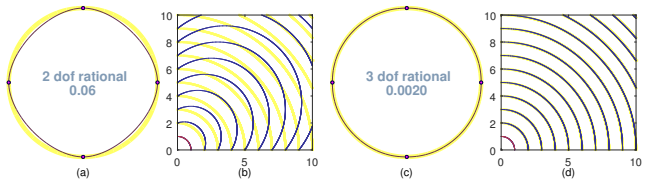


Figure 9: *(a) a circle approximation by four implicit rational curves ($f = x+y-1-xy/(x+y)$ in the first quadrant); (b) contour lines for $f = (x^2+y^2-1)/(x+y+1)$ that approximates a unit arc exactly; (cd) approximation by expression (8).*

the two sets equal to 0.06. We can improve the approximation by adding $f_2 = c\,(d_0+d_3)\,(t_0\,t_3)^2$ to f. This quadratic term automatically satisfies both Hermite conditions and asymptotically behaves as $O(d_{0,3})$. We can use the constant c to fit the given points between $q_{0,3}$. For a unit arc in the first quadrant, the resulting function

$$f = x+y-1-xy\,(1+0.664\,xy/(x+y)^2)/(x+y) \qquad (8)$$

deviates from the true distance to the arc by no more than 0.2% at any distance from it, as shown in Figure 9(cd). For comparison, the Hausdorff distance between a unit quarter arc and its cubic Bézier approximation is $\sqrt{71/6-2\sqrt{2}}/3-1 \approx 0.03\%$ [AKS04].

We could have approximated the arc exactly, e.g., by choosing $f_2 = c\,d_0\,d_3\,d_1/(d_0+d_3)/(d_0+d_3+1)$, where d_1 is a distance from x to a line passing through $q_{0,3}$. Yet, the resulting function, $f = (x^2+y^2-1)/(x+y+1)$, significantly deviates from the true distance to the arc, as can be seen in Figure 9b.

In situations when we want a sharp edge (like the two green regions in Figure 4 or one of the triangle vertices in Figure 7), tangent lines will be different on the two sides (G0 connectivity). Consequently, edges of the corresponding Voronoi regions will not be orthogonal to the tangent lines. Values of $a_{0,3}$ can be then computed as (using $(\cdot)$ notation for a dot product and $v^\perp$ for an orthogonal vector) :

$$a_0 = \frac{(q_{10}^\perp \cdot q_{30})}{(n_0^\perp \cdot q_{30})\,(n_0^\perp \cdot q_{10})}, \quad a_3 = \frac{-(q_{23}^\perp \cdot q_{30})}{(n_3^\perp \cdot q_{30})\,(n_3^\perp \cdot q_{23})} \qquad (9)$$

where q_{10} and q_{23} are two tangential vectors and $q_{30} = q_3 - q_0$.

To find the coefficient c that minimizes curve deviation from a given set of points, we solve the system of over-defined linear equations

$$d_n + (a_0 d_3 + a_3 d_0)\,t_0\,t_3 + c\,(d_0+d_3)\,(t_0\,t_3)^2 = 0$$

where equation coefficients are computed at each given point (i.e., perform a linear regression). Run-time evaluation of the distance to the curve `dc` from sample `uv` is given in Appendix A, lines 52 – 55.

5.5. Choosing Rendering Resolution for Vector Images

When a vector image is rasterized and antialiased, a color of each pixel is obtained by averaging all relevant subsamples. A color of pixel p_3 in Figure 11 is influenced by colors on both sides of the blue curve that intersects it. The intent of the resampling offset `duv` in the statement (1) is to move away from such pixels since they do not faithfully reproduce the colors of the original vector image.

Figure 10: *(a-c) increasing a resolution of a companion raster image improves fidelity near very thin features (right part of the image); (d) reconstructing edges from 1542^2 raster image (not using SVG data at all); (e) using stored colors together with raster image improves quality near sharp corners (bottom right).*

We choose a resolution of a raster image so that most of `uv+duv` samples land in pixels that are not intersected by any curve; there are 96% such samples for the Garfield image in Figure 12 with the resolution set to 1542×1542. Figure 10 illustrates effects of varying the resolution of a companion raster image.

For some vector images with a significant amount of intersected curves or ones with a very thin strokes, the required resolution might be too high. For such images, we could store colors on both sides of curve segments in an acceleration structure. This will work though only for images with a piecewise flat colors (Figure 10e).

5.6. Acceleration Structure Layout and Optimization

Formulae (6) and (9) provide us with an opportunity to trade bandwidth for computations, given that (smoothly) connected segments require very little storage as such. Analyzing such trade-offs on different architectures is an interesting problem in itself, but for the sake of clarity in this paper we precompute and store all coefficients for each curve and its Voronoi region separately.

Following Qin et al. [QMK08], we use a grid acceleration structure with the dimensions of the original raster image. Since each pixel can be overlapped by multiple regions, this would have required a double indirection: pixel index $\mapsto$ list of indices for overlapped regions $\mapsto$ region data (similar to indexed meshes in Computer Graphics).

We opted for a single indirection with pixel index directly pointing into a sequence of the curves influencing this pixel. Referring to Figure 11, pixel p_0 has three Voronoi regions covering some parts

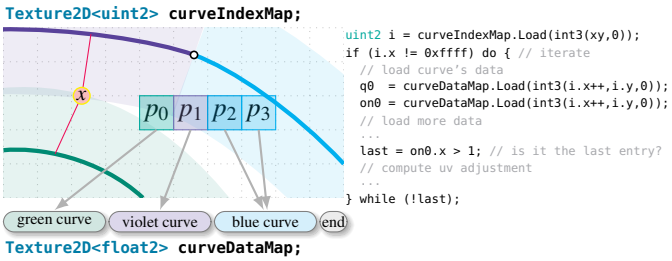


Figure 11: *acceleration structure layout using two texture maps. For a given sample x with scaled texture coordinates `xy`, we use `curveIndexMap` to access a list of curve segments (`curveDataMap`) overlapping a texel that contains x.*

of it and its index points into the start of the sequence of three such curves. Pixel p_1 is overlapped by only violet and blue regions and it points into the second entry in the same sequence, while $p_{2,3}$ share the blue region at the end of the sequence. At run time, we iterate over all corresponding curves until the end-of-the-sequence flag is read (see `do…while` loop at line 21 in Appendix A).

To minimize the overlap of areas of influence for each curve (as for sample x in Figure 11), we construct a Voronoi diagram for super-sampled edges and then consider intersections of normals at curves' endpoints with such a diagram to produce side edges for truncated Voronoi regions. We clip such vectors if their length exceeds the supplied parameter h in (2) and (optionally) reduce them for weak edges as defined by the gradient magnitude during the edge detection step (section 5.1). If the intersection of these two vectors for each region is closer to $q_{0,3}$ than their assigned length, we reduce this length to exclude the intersection from the area of influence and avoid potential division by zero during the run-time computation (line 37 in Appendix A). By linearly interpolating the found edge vectors — considering both positive and negative directions — we will get an area of influence for a particular curve (line 58). Such areas will contain points that are closer to the given curve than to any other (as in Figure 7g). Small overlaps are still possible and we handle them by summing the corresponding `duv` offsets so not to impair `duv` continuity (line 86).

6. Discussion

IRT combines the best traits of vector and raster textures in a unified arrangement. It smoothly blends vector and raster representations to exploit the pre-filtered accommodating properties of the traditional textures at farther distances. This blending does not incur any additional penalty, besides computing the mipmap level. Thus, the infinite resolution textures are only slightly more expensive than traditional mipmapped textures for distant objects and exhibit reasonable performance for close-ups (section 6.1).

IRT textures differ from traditional textures at closer distances, revealing crisp edges that hopefully agree with human intuition. There are clearly limits to such detail hallucination (section 6.3).

6.1. Performance

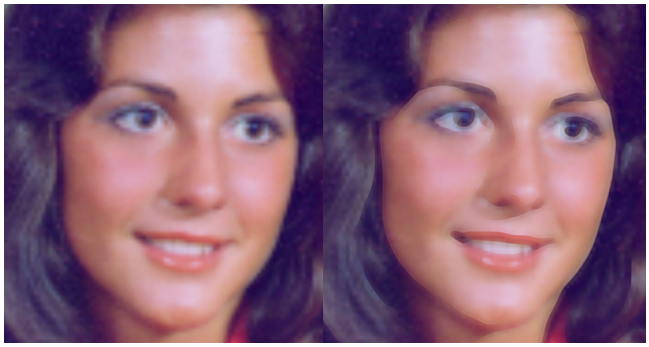
By creatively using hardware not originally designed for vector graphics per se, GPU-accelerated methods for vector textures [KB12, GLdFN14] achieve rendering times of 20+ milliseconds per 1M texels. This works well when the whole image has to be rendered, since certain operations are amortized over the whole image.

Our goal is to complement such methods, targeting resource consumption by games with random texture sampling, rather than resource creation. We measured the performance of our algorithm on a discrete GeForce™ GTX 980 using a Direct3D 11 viewer.

For 24 images in the Kodak Image Suite, a single frame with 1M pixels runs from 140 to 412 microseconds. The slowest performance was observed for image №13, for which almost all pixels are in close proximity to one or more curves. A fragment of this image is shown in Figure 12c.

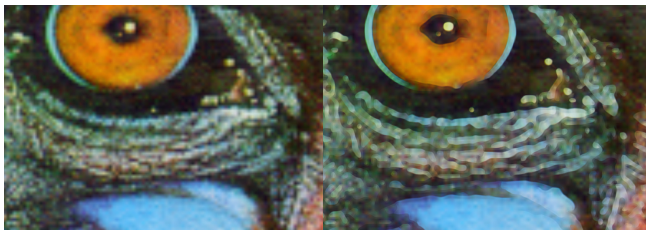
The performance numbers for the USC-SIPI Image Database are

(a) Bézier = [166,2021,2305], rational = [136,1655,1803]

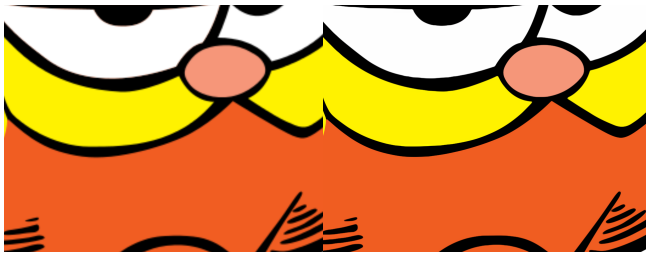


(b) Bézier = [161,2008,2065], rational = [138,1656,1728]

(c) Bézier = [616,10275,8990], rational = [412,5560,5837]; edges shown on the left



(d) Bézier = [456,7193,6979], rational = [305,4481,4653]



(e) Bézier = [178,2587,1692], rational = [177,2603,1693]

Figure 12: *Details of various images rendered with bilinear (left column) and IRT sampling (right). Performance for the whole image is given in µs per 1M texels for* [discrete NVIDIA GeForce™ GTX 980, mobile GT 720, integrated Intel® HD Graphics 4400 (performance mode)] *GPUs.*

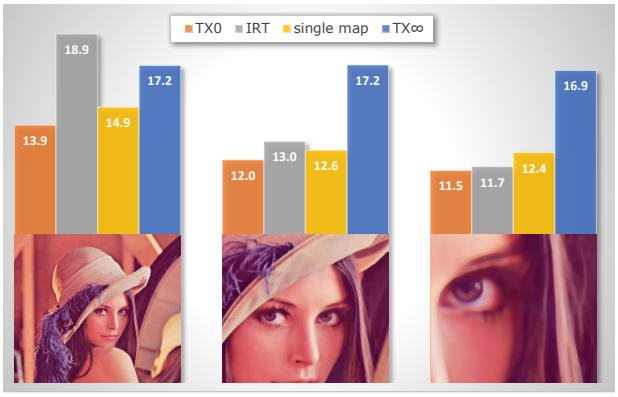


Figure 13: *GeForce™ GTX 680 memory controller traffic in MB per a single 1024×1024 frame at three different viewpoints. Four different texture sampling approaches were used: TX0 — sampling the original 512×512 raster image with a hardware texture unit; IRT — the method in this paper; single map — IRT with a single record per pixel; TX∞ — mipmaped texture that achieves the same quality as IRT (shown at the bottom of the chart).*

in the range of 136 – 305 μs; the slowest image is, not surprisingly, the mandrill (a fragment of which is shown in Figure 12d).

Just sampling a raw raster texture incurs a performance overhead of about 90 μs per 1M texels for any image, so for simpler images the IRT cost roughly doubles texture unit timing. On the tested mobile platforms, this observation also holds true with the overall performance constrained by the texture sampler throughput.

These measurements are for the rational polynomials (5.4); Bézier curves (5.1) are about 30% slower. For each image in Figure 12, we give measurements for one desktop and two mobile cards.

We believe the low cost of IRT during magnification and negligible cost during minification should allow it to be turned on all the time for all textures the artist deems appropriate.

6.2. Bandwidth

IRT uses two precomputed data structures (section 5.6) supplementing a given raster image:

- an array of the curved segments whose size is proportional to the number of the segments and
- per-pixel indices into these segments.

These two structures exhibit different memory utilization patterns. The index data is a constant overhead for all texels with LOD that is less than the IRT threshold (see line 19 in Appendix A). The segment data, however, is amortized over visible pixels, benefiting from the texture cache at closeups. This is illustrated in Figure 13, for which we evaluated the GPU memory controller traffic by rendering a quad with Lenna image at three different viewpoints. This data was measured by GPU-Z — Video card GPU Information Utility [Tec16]. The third bar in the chart shows a hypothetical situation when there is only one 32-bit record per pixel that describes the sole segment overlapping the pixel. We did not implement this version of IRT per se, but simulated it to understand the bandwidth requirements of the methods with the constant per-pixel overhead, such as pinchmaps [TC05] or silmaps [Sen04]. Pinchmaps always fetch a single record per texel, while for silmaps accessing neighboring records is necessary for texels with more than one border segment, but there could be only up to 4 boundary edges in each cell.

The IRT segment data might cause a noticeable memory traffic increase, especially at midrange distances for images that are more complex than Lenna. On the positive side, this extra data provides a flexibility to describe multiple curved segments intersecting a given pixel.

An obvious alternative to edge preserving techniques is a traditional texture format that is mipmapped from a coarse to a very detailed representation (if such hi-res representation is available at all). The bandwidth data for such implementation is approximated in the fourth bar in Figure 13 by capturing IRT-rendered image and then reusing it as a bilinearly interpolated texture. This creates the same image that is rendered faster at the expense of increasing memory traffic at close-ups, even though only a single mipmapped level is used (trilinear interpolation is not necessary in this scenario due to 1:1 correspondence between pixels and texels).

Note that there are (hardware-specific) limitations on the maximum texture size. The right image in Figure 13 requires a 4096×4096 texture to achieve the same level of quality as IRT with 512×512 base raster image. We did not use a 4096×4096 texture as such, instead creating a 1024×1024 snapshot by capturing the IRT-rendered framebuffer.

Raster images, used in Figure 13, were not compressed. Real applications, such as games, tend to use compressed formats to save memory and bandwidth. Rendering a single quad is not a good way to evaluate all possible trade-offs. Still, to get some insight into potential issues, we redid all measurements using DXT1 compression. This did not change the frame rate in any measurable way (since it is already near the limit of the graphics pipeline throughput), but reduced the memory controller load from 68% to 51% for the last image in the TX∞ mode. Even though this reduction is significant, the memory controller can be a limiting resource by itself, accentuating the complex nature of the modern GPUs.

The most efficient compression techniques introduce artifacts, especially near high-frequency edges. In this respect, splitting the image into a low-resolution raster and a high-resolution edge representation might allow deeper compression levels for the low-resolution parts. We plan to explore all such trade-offs, also considering a potential curve data compression as well. Another bandwidth saving approach would require stealing a bit from the RGBA representation to indicate whether the particular pixel is influenced by any curve. This would significantly reduce the memory traffic due to the curve indexing but it is application-specific as those extra bits might not be readily available.

6.3. Limitations

IRT is only as good as the edge detection algorithm it uses. Underdetected edges can lead to incorrect conclusions about object affinities: similarly-colored candies in Figure 12a look melted. In a digital era, viewers tend to expect a correlation between an image acuity and visible details. IRT may break this correlation by slumping into

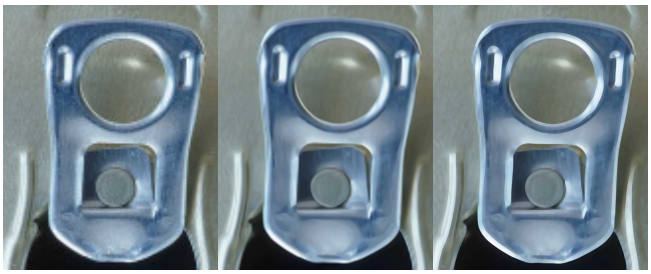


Figure 14: *Both IRT (right) and 9X superresolution (middle) reduce blurring and aliasing near edges; such artifacts are symptomatic of the bilinear interpolation (left).*

an uncanny valley: the IRT-rendered eye in Figure 5 looks better than the bilinearly interpolated one, but individual eyelashes are not visible and neither is a reflection in the iris. To a certain degree, IRT creates an impression of a painted human figure at close proximity, rather than a high fidelity photograph (Figure 12b). If needed, we could even exaggerate this effect by filtering a raster image that is used in tandem with IRT — high frequency details will be preserved in IRT curves anyway.

It is interesting to note that for some objects, even widely used, there are no a priori expectations of the discernible details; IRT-based image magnification of a can in Figure 14 looks roughly equivalent to one obtained with a super-resolution technique — nonlocal autoregressive modeling [DZLS13] — but significantly faster to render. Notably, the recently proposed super-resolution method based on subpixel shifting model [JP15], which is similar in spirit to IRT, achieves remarkable reconstruction results.

When used for sampling genuine vector graphics, IRT images are somewhat different from ones obtained with the more evolved techniques [KB12, GLdFN14]. Away from edges, we rely on a trilinear mipmapping to compute a sample color, while SVG colors are analytically calculated for all samples. IRT colors are always derived from the corresponding raster image. A low resolution of this image could be a problem for samples that are close to multiple curves, see Figures 10 and 12e.

Our current implementation uses Matlab code for the edge detection 5.1, and requires a few minutes per image; we expect this to be drastically reduced by switching to a GPU version. We also have not yet explored any benefits of data compression; rational polynomial approximation (section 5.4) looks more appealing in this regard as it operates on naturally compressible entities such as distances and angles.

6.4. Future Work

Texture sampling is ubiquitous in modern computer graphics. We anticipate that IRT can be used for a wide range of effects, including:

- Material properties (colors).
- Normal maps, which are typically used to represent minute geometric detail. The prospect of providing infinite resolution and crisp edges for such maps is appealing.
- Vegetation rendering. Traditionally, foliage is rendered by representing individual leaves or branches as textures with an alpha channel for transparency. This works well at a distance but breaks at close-ups. A vector representation should help with this, see the leaf image in Figure 3.
- 1D curve rendering (hair).
- Video game decals (like bullet holes or char marks) and pellucid text on arbitrary surfaces.
- User interface elements for games and web pages.
- Shadow maps and light maps.

Acknowledgements

We would like to thank the anonymous reviewers for their detailed and helpful comments and suggestions. We gratefully acknowledge invaluable discussions with and assistance from Henry Moreton.

We appreciate the opportunity to use the Kodak Lossless True Color Image Suite, the USC-SIPI Image Database, and Computer Vision Lab of the Weizmann Institute of Science repository. Averaged male face (Figure 5) is taken from the "Beautycheck" homepage of the Universität Regensburg. Garfield is licensed under a Creative Commons license from Brands of the World site.

References

[AKS04] AHN Y., KIM Y., SHIN Y.: Approximation of circular arcs and offset curves by Bézier curves of high degree. *Journal of Computational and Applied Mathematics 167*, 2 (2004), 405–416. 8

[Bah07] BAH T.: *Inkscape: Guide to a Vector Drawing Program*, first ed. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007. 5

[BKKL15] BATRA V., KILGARD M. J., KUMAR H., LORACH T.: Accelerating vector graphics rendering using the graphics hardware pipeline. *ACM Trans. Graph. 34*, 4 (July 2015), 146:1–146:15. 2

[Can86] CANNY J.: A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell. 8*, 6 (June 1986), 679–698. 6

[DLG87] DYN N., LEVIN D., GREGORY J. A.: A 4-point interpolatory subdivision scheme for curve design. *Comput. Aided Geom. Des. 4*, 4 (Dec. 1987), 257–268. 6

[DZLS13] DONG W., ZHANG L., LUKAC R., SHI G.: Sparse representation based image interpolation with nonlocal autoregressive modeling. *IEEE Trans. on Image Processing 22*, 4 (2013), 1382–1394. 11

[Flo95] FLOATER M. S.: Rational cubic implicitization. In *Mathematical Methods for Curves and Surfaces* (1995), Lyche, Schumaker, (Eds.), Vanderbilt University Press 1995, pp. 151–159. 7

[GLdFN14] GANACIM F., LIMA R. S., DE FIGUEIREDO L. H., NEHAB D.: Massively-parallel vector graphics. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 229:1–229:14. 2, 5, 9, 11

[JP15] JEON J., PAIK J.: Single image super-resolution based on subpixel shifting model. *Optik - International Journal for Light and Electron Optics 126*, 24 (2015), 4954 – 4959. 2, 11

[JSK99] JÄHNE B., SCHARR H., KÖRKEL S.: *Principles of Filter Design*, vol. 2. Academic Press, 1999, p. 125–151. 6

[Kan87] KANADE T.: Image understanding research at CMU. In *Proc. of the Image Understanding Workshop* (Los Angeles, CA, 1987), vol. 2, pp. 32–40. 6

[KB12] KILGARD M. J., BOLZ J.: GPU-accelerated path rendering. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 172:1–172:10. 2, 9, 11

[KHPS07] KNISS J., HUNT W., POTTER K., SEN P.: IStar: A raster representation for scalable image and volume data. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (nov 2007), 1424–1431. 3

[KL11] KOPF J., LISCHINSKI D.: Depixelizing pixel art. *ACM Trans. Graph. 30*, 4 (July 2011), 99:1–99:8. 2

[LB05] LOOP C., BLINN J.: Resolution independent curve rendering using programmable graphics hardware. In *ACM SIGGRAPH 2005 Papers* (2005), SIGGRAPH '05, pp. 1000–1009. 2, 7

[MA09] MAINI R., AGGARWALI H.: Study and comparison of various image edge detection techniques. *International Journal of Image Processing (IJIP) 3*, 1 (Feb. 2009), 1–11. 5, 6

[MSH12] MITTAL A., SOFAT S., HANCOCK E. R.: Detection of edges in color images: A review and evaluative comparison of state-of-the-art techniques. In *Proc. Autonomous and Intelligent Systems (AIS) 2012* (2012), pp. 250–259. 5

[NH08] NEHAB D., HOPPE H.: Random-access rendering of general vector graphics. *ACM Trans. Graph. 27*, 5 (Dec. 2008), 135:1–135:10. 7

[NH12] NEHAB D., HOPPE H.: A fresh look at generalized sampling. *Foundations and Trends in Computer Graphics and Vision 8*, 1 (2012), 1–84. 6

[OBW*08] ORZAN A., BOUSSEAU A., WINNEMÖLLER H., BARLA P., THOLLOT J., SALESIN D.: Diffusion curves: A vector representation for smooth-shaded images. *ACM Trans. Graph. 27*, 3 (2008), 1–8. Special Issue: SIGGRAPH 2008. 3

[Pho75] PHONG B. T.: Illumination for computer generated pictures. *Commun. ACM 18*, 6 (June 1975), 311–317. 4

[PP11] PAPARI G., PETKOV N.: Review article: Edge and line oriented contour detection: State of the art. *Image Vision Comput. 29*, 2-3 (Feb. 2011), 79–103. 5

[QMK08] QIN Z., MCCOOL M. D., KAPLAN C.: Precise vector textures for real-time 3D rendering. In *Proc. of the 2008 Symposium on Interactive 3D Graphics and Games* (2008), I3D '08, pp. 199–206. 8

[RCL05] RAY N., CAVIN X., LÉVY B.: *Vector texture maps on the GPU*. Tech. rep., Laboratoire Lorrain de Recherche en Informatique et ses Applications, 2005. 2

[Res12] RESHETOV A.: Reducing aliasing artifacts through resampling. In *Proc. of the Fourth ACM SIGGRAPH/Eurographics Conf. on High-Performance Graphics* (2012), EGGH-HPG '12, pp. 77–86. 2

[SC12] SHRIVAKSHAN G. T., CHANDRASEKAR C.: A comparison of various edge detection techniques used in image processing. IJCSI Press. 5

[SCH03] SEN P., CAMMARANO M., HANRAHAN P.: Shadow silhouette maps. In *ACM SIGGRAPH 2003 Papers* (2003), SIGGRAPH '03, pp. 521–526. 3

[Sen04] SEN P.: Silhouette maps for improved texture magnification. In *Proc. of the ACM SIGGRAPH/Eurographics Conf. on Graphics Hardware* (2004), HWWS '04, pp. 65–73. 2, 3, 10

[Sha73] SHAPLEY R. M.TOLHURST D. J.: Edge detectors in human vision. *The Journal of Physiology 229*, 1 (1973), 165–183. 2

[SOT13] SMITH J., OSBORN J., TEAM A. C.: *Adobe Creative Cloud Design Tools Digital Classroom*, 1st ed. Wiley Publishing, 2013. 5

[SWWW15] SONG Y., WANG J., WEI L.-Y., WANG W.: Vector regression functions for texture compression. *ACM TOG 35*, 1 (2015). 3

[SXD*12] SUN X., XIE G., DONG Y., LIN S., XU W., WANG W., TONG X., GUO B.: Diffusion curve textures for resolution independent texture mapping. *ACM Trans. Graph. 31*, 4 (July 2012), 74:1–74:9. 2, 3

[TC04] TUMBLIN J., CHOUDHURY P.: Bixels: Picture Samples with Sharp Embedded Boundaries. *Eurographics Workshop on Rendering* (2004). 2, 3

[TC05] TARINI M., CIGNONI P.: Pinchmaps: textures with customizable discontinuities. *Comput. Graph. Forum 24*, 3 (2005), 557–568. 2, 3, 10

[Tec16] TECHPOWERUP: *GPU-Z Video card GPU Information Utility*, 2016. URL: https://www.techpowerup.com/gpuz. 10

[Wil83] WILLIAMS L.: Pyramidal parametrics. *SIGGRAPH Comput. Graph. 17*, 3 (July 1983), 1–11. 2

Appendix A: HLSL implementation of IRT.

```
Texture2D<uint2>  curveIndexMap : register(t2); // see Figure 11
Texture2D<float2> curveDataMap  : register(t3); // in section 5.6

float4 Resample(float2 uv) {
  float2 xy = uv * texdim; // texdim = texture dimensions
  float pixratio = 0.5f * length(fwidth(xy)); // pixel/texel size
  float adc, weight = 1, lod = max(0,log2(pixratio));
  // lod = tex.CalculateLevelOfDetailUnclamped(s,uv); // alternative
  // pixratio = exp(lod); lod = max(0, lod);          // version
  // see https://www.opengl.org/discussion_boards/showthread.php/171485
  //     (Texture LOD calculation useful for atlasing)

  // We start reducing duv when pixratio >0.5 and completely stop
  // offsetting uv when pixratio >= 1.
  float2 reduce = min(1, 2 * (1 - pixratio)) / texdim;
  float2 q0s, duv = {0, 0};

  uint2 i = curveIndexMap.Load(int3(xy, 0)); // curve index

  if (i.x != 0xffff && pixratio < 1) {        // ∃ curves near uv
    bool last; // flag for the last curve in the list
    do {       // sample-dependent number of iterations
      float2   // read curve data
        q0  = curveDataMap.Load(int3(i.x++, i.y, 0)),// endpoint
        on0 = curveDataMap.Load(int3(i.x++, i.y, 0)),// ort(n0)
        on3 = curveDataMap.Load(int3(i.x++, i.y, 0)),// ort(n3)
        aux = curveDataMap.Load(int3(i.x++, i.y, 0)),// aux data
        n30 = curveDataMap.Load(int3(i.x++, i.y, 0)),// ort(q3-q0)
        a03 = curveDataMap.Load(int3(i.x++, i.y, 0)),// p1-q0 or tan
        n10 = curveDataMap.Load(int3(i.x++, i.y, 0)),// Voronoi
        n13 = curveDataMap.Load(int3(i.x++, i.y, 0));// sizes
      last = on0.x > 1;                   // is it a last curve?
      if (last) on0.x -= 3;               // restore original value
      q0s = q0 - xy;                      // from sample to q0
      float d0 = dot(on0, q0s);           // barycentric coordinates
      float d3 = dot(on3, xy) + aux.x;    // in Voronoi region (VR)
      bool outside = d0 < 0 || d3 < 0;    // if outside, duv += 0;
      float  den = 1/(d0 + d3);           // we avoid 1/0 by design
      float    t = d0*den;                // d{03} > 0 inside VR
      float2 oni = lerp(on0, on3, t);     // ort(ni)
      float2  ni = float2(-oni.y, oni.x); // interpolated normal

      #ifdef EXACT_BEZIER // implementation of section 5.3; a03 = p1-q0; (Figure 4)
      float2 ab = curveDataMap.Load(int3(i.x++, i.y, 0));// six
      float2 cd = curveDataMap.Load(int3(i.x++, i.y, 0));// poly
      float2 ef = curveDataMap.Load(int3(i.x++, i.y, 0));// coeffs
      float   w = (xy.x-q0.x)*n30.x + (xy.y-q0.y)*n30.y; // Bary-
      float   v = (xy.x-q0.x)*a03.y - (xy.y-q0.y)*a03.x; // centric
      float   u = aux.y - w - v; // aux.y = 2 * area of Bezier tri
      float  dc = u*v*(ab.x*u + ab.y*v + cd.x*w) + // eval Floater
                  w*w*(cd.y*u + ef.x*v + ef.y*w);  // polynomial
      #else // implementation of section 5.4; a03 = equation 9 (≈ tangents); aux.y = c;
      float  dc = dot(q0s, n30)/dot(ni, n30); // xy+dc*ni ∩ [q0,q3]
      float  polq = t*d3;                     // polynomial ratio
      float  ai = lerp(a03.x, a03.y, t);      // interpolated tan
      dc +=  (ai + aux.y * polq*den) * polq;  // |xy - curve|
      #endif

      float2 hs = lerp(n10, n13, t);   // scaled down h
      float rim = dc > 0? hs.x : hs.y; // choose side of curve[i]
      float d2r = rim - dc;            // distance to the rim
      // smoother alternatives
      // float d2r = rim - (dc + rim/2) * 2/(2+1);  // soft landing
      // float d2r = sqrt(abs(rim))*sign(rim) - dc; // another way
      // edge smoothing = 1/q; no need for antialiasing in this case
      // float q = 1; d2r *= min(1, q*abs(dc/rim)); pixratio = -1;

      adc = outside * 1e6 + abs(dc); // force duv += 0 outside VR
      pixratio -= vismode & (1<<4);  // ignore next block in !AA mode
      if (adc < pixratio) {      // antialias the sample (2 modes)
        if (vismode & (1<<7)) { // a single sample
          static const float h = 2*sqrt(2.0);
          float fix = min(0.5f, pixratio * falloff); // edge width
          float morph = fix * h;
          if (morph - adc > 0) {
            float loda = (morph - adc)/morph; // [0,1]
            d2r *= 1 - loda;     // modified distance
            lod += loda;         // increased L.O.D.
          }
        } else { // blend 2 samples with
          weight = (adc + pixratio)/(2*pixratio);
        }
        duv = 0;      // only use the
        last = true; // current curve
      }
      d2r *= adc < abs(rim); // 0 for afar samples
      duv += d2r * ni; // uv offset along the interpolated normal
    } while (!last);

    if (vismode & (1<<2)) {
      float r = min(1,length(q0s)/(5*pixratio));
      if (adc < pixratio) {  // show curve
        float4 c = tex.SampleLevel(s, uv, lod);
        return lerp(float4(r,1-r,1-r,1), c, adc/pixratio);
      }
      float lo = abs(0.5*length(duv)/pixratio - 1);
      if (lo < 1) {          // show curve's influence
        float4 c = tex.SampleLevel(s, uv, lod);
        return lerp(float4(1, 1, 0, 1), c, lo);
      }
    }
  }

  float4 c = tex.SampleLevel(s, uv - reduce*duv, lod); // eq (1)
  // return 1-float4(abs(duv),0,0); // VR ↦ pseudocolors
  if (weight == 1) return c; // no need in second sample
  // blend with the second sample for edge antialiasing
  return lerp(tex.SampleLevel(s, uv + reduce*duv, lod), c, weight);
}
```