GPU Efficient Texture Based Bezier Curve Evaluation

Alan Wolfe
Blizzard Entertainment

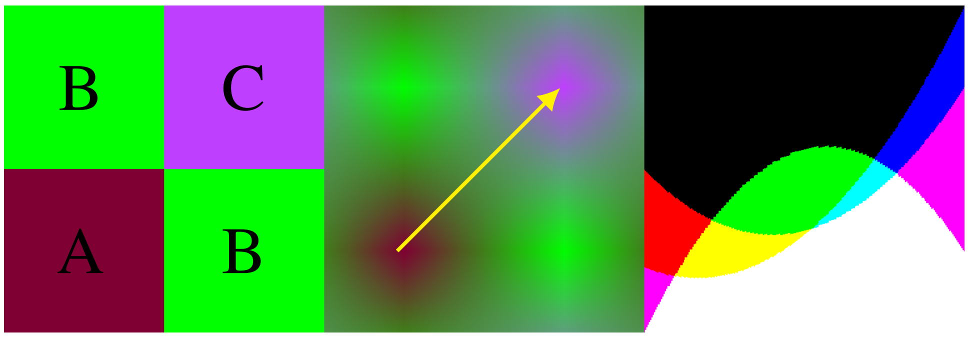

Figure 1. *Left:* 2x2 texture containing control points for a quadratic Bezier curve in each color channel. *Middle:* The texture as viewed with bilinear sampling. *Right:* The resulting quadratic Bezier curves when sampling along the yellow line.

Abstract

Modern graphics techniques expose internal parameters to allow re-use and to help separate the technical implementation from artistic usage cases. A popular choice is to expose parameters as customizable curves and to quantize the curves into textures. Quantization leads to either lower quality results, or more texture memory being used to accommodate higher sampling frequencies. The technique presented in this paper leverages the capabilities of GPU texture samplers to allow more efficient storage and evaluation of both integral and rational Bezier curves of any degree, resulting in higher fidelity for similar costs. Piecewise curves, B-Splines and NURBS are addressed, and there are also limited applications towards vector graphics.

1. Introduction

There are two basic ways for a shader program to access customized curve data. One way is to evaluate the curve on the CPU at discrete intervals and put those points into a texture that can be used by the GPU. The other way is to pass curve control point

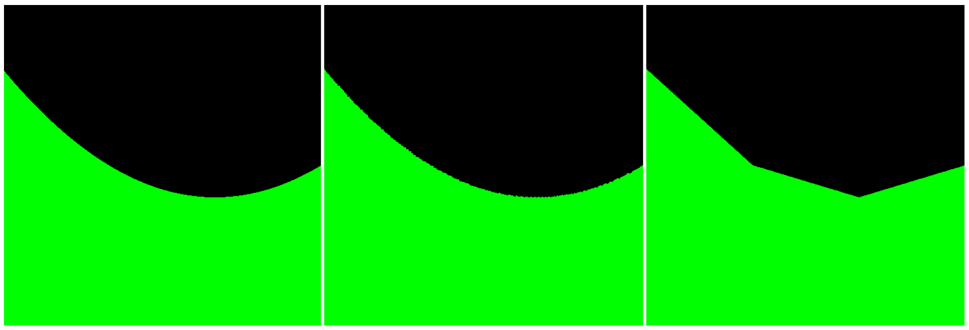

Figure 2. Quadratic Curve. *Left:* Calculated from shader program constants. *Middle:* The technique in this paper, using a 2x2 texture and a single texture sample per pixel. *Right:* Baked out curve, using a 4x1 texture and a single texture sample per pixel.

data from the CPU to the shader program as shader constants, allowing the shader program to evaluate the curve at a given t.

Baking curve points into a texture means that the shader program doesn't need to know what type of curve it was that made the data, nor how many control points it had, but comes at the loss of quality since there are a finite number of curve points sampled, and the data between points is just a linear interpolation between the samples. If higher quality is desired, you can increase the resolution of the texture to trade accuracy for texture memory.

Passing control point data to a shader program as shader constants allows you to get high quality curve calculations, but comes at the cost of the shader program being written specifically for the type of curve you want to use, and also takes more shader program instructions to calculate specific curve points.

This paper shows a third method where:

- The control points of the curves are encoded in a texture.
- The texture sampler calculates points on the curve before the data reaches the shader program.
- It gives accuracy results closer to that of shader constant curves, while having less calculation overhead.
- The technique can support both integral and rational Bezier curves of any degree and can also be used for piecewise curves.
- The curve type must be decided on in advance, like when using the shader constant method.
- There are limited applications towards vector graphics.

A quick comparison of the visual quality of these three techniques can be seen in Figure 2.

Besides being useful for exposing internal parameters, the technique presented in this paper is also useful as an alternative to traditional shader look up tables, since this technique allows non linear interpolation between data points, where traditional lookup table textures only offer linear interpolation between data points.

2. The Technique

The core of this technique is that the N-dimensional linear texture interpolation capabilities on the GPU can be mathematically equivalent to De Casteljeau's algorithm for evaluating Bezier curves. We can use linear texture sampling on textures of increasing dimensionalities to have the texture sampler calculate points on curves of increasing degree.

2.1. Intuition

One dimensional linear texture sampling is just the linear interpolation between two data points. This is equivalent to the De Casteljeau algorithm when evaluating a linear curve of degree 1. In both cases, it is just linear interpolation between two values.

Two dimensional linear texture sampling (bilinear texture sampling) can be thought of as just the linear interpolation across the y axis, of two linear interpolations across the x axis (or an interpolation across the x axis of two linear interpolations across the y axis - order doesn't matter). In other words, bilinear texture sampling is just the linear interpolation of two linear interpolations.

The De Casteljeau algorithm for evaluating a quadratic curve (degree 2) is also just a linear interpolation between two linear interpolations. If the control points of the quadratic curve are P_0, P_1, P_2, let's call the first interpolated point $Q_{0,1}$ which is the interpolation between P_0 and P_1 at t. The second linear interpolation is $Q_{1,2}$ and is the interpolation between P_1 and P_2 at t. The final point on the curve will be the interpolation at t between $Q_{0,1}$ and $Q_{1,2}$, which we can call $R_{0,1,2}$.

We could set up a texture such that the two x axis interpolations would give us the values $Q_{0,1}$ and $Q_{1,2}$ respectively, and then the y axis interpolation between those values would give us $R_{0,1,2}$. When sampling, we would also use texture coordinates such it results in the same t value for interpolation on each axis.

To further see how bilinear interpolation can be equivalent to evaluating a quadratic Bezier curve with the De Casteljeau algorithm, Figure 3 shows it visually, and you can also compare the GLSL source code in Listing 1.

```
float QuadraticBezier (
  in float t,
  in float P0, in float P1, in float P2
) {
```

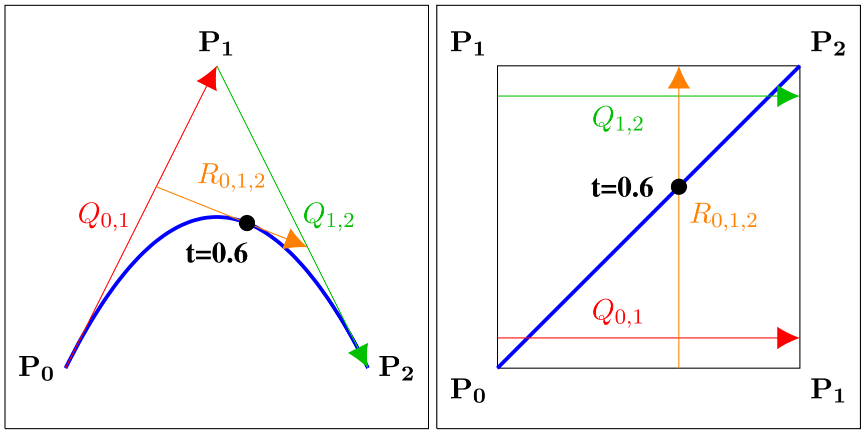

Figure 3. *Left:* A quadratic Bezier curve evaluated at **t**=0.6 using the De Casteljeau algorithm. *Right:* The De Casteljeau algorithm using bilinear interpolation.

```
    return mix(mix(P0,P1,t), mix(P1,P2,t), t);
}

// This function is equivelant to the function above when:
// u = t, v = t
// A = P0, B = P1, C = P1, D = P2
float BilinearInterpolate (
  in float u, in float v,
  in float A, in float B, in float C, in float D
) {
    return mix(mix(A,B,u), mix(C,D,u), v);
}
```

Listing 1. GLSL implementation of bilinear interpolation and the De Casteljeau algorithm for a quadratic Bezier curve.

The pattern continues for higher texture dimensions and curve degrees as well. Linear texture sampling of textures of dimension N is just the linear interpolation between two linear texture samples of dimension $N-1$. Using the De Casteljeau algorithm to evaluate a curve of degree N just means that you take a linear interpolation between curves of degree $N-1$.

Compared to a quadratic curve evaluated with De Casteljeau's algorithm, bilinear texture sampling has two blend weights instead of one, and has four values to interpolate between instead of three, but you can set a texture up such that when you sample it a specific way, you get as output a quadratic Bezier curve. Because of this, the De Casteljeau algorithm is a subset of the N-dimensional linear texture sampling available on the GPU.

It can be seen that the actual pixel value in a bilinear sampled texture is in the middle of a pixel. That means that when calculating our texture coordinates for eg.

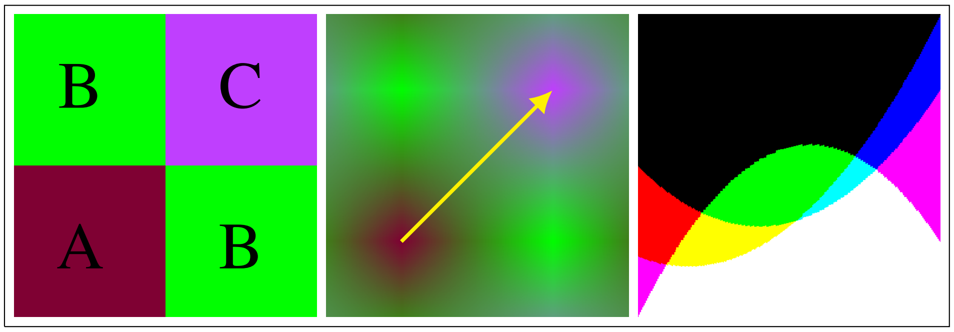

Figure 4. *Left:* a 2x2 texture storing control points for a quadratic Bezier curve in each color channel. *Middle:* The same texture as viewed when using bilinear texture sampling. The yellow line indicates where texture samples are taken from to evaluate the quadratic Bezier curve. *Right:* The curves resulting from sampling along the yellow line.

a 2x2 texture, we must sample between pixel location $(0.5, 0.5)$ and pixel location $(1.5, 1.5)$ to get the correct results.

The common pixel formats contain four color channels – Red, Green, Blue and Alpha – which allows us to be able to evaluate four curves with a single texture read (Figure 4).

It is also possible to get points on curves of higher degree by taking multiple texture reads and combining them either using the De Casteljeau algorithm, or by using the Bernstein form equation of Bezier curves.

2.2. Mathematical Basis

The De Casteljeau algorithm is equivalent to the Bernstein form of Bezier curves (Equation 1).

$$\mathbf{P(t)} = \sum_{\mathbf{i=0}}^{\mathbf{n}} \binom{\mathbf{n}}{\mathbf{i}} (\mathbf{1-t})^{\mathbf{n-i}} \mathbf{t^i P_i} \tag{1}$$

This means that the Bernstein form for a given n must also be able to be evaluated by n-Linear interpolation. A linear curve should be able to be evaluated by linear interpolation, a quadratic curve should be able to be evaluated by bilinear interpolation, a cubic curve should be able to be evaluated by trilinear interpolation, and so on.

When n is 1, to make a linear Bezier curve, the equation we get out is in fact just linear interpolation, so we can see that it's true in that case.

$$\mathbf{P(t)} = \mathbf{P_0(1-t)} + \mathbf{P_1 t} \tag{2}$$

When n is 2, to make a quadratic Bezier curve with bilinear interpolation, the relationship is not as obvious.

$$\mathbf{P(t) = P_0(1-t)^2 + P_1 2(1-t)t + P_2 t^2} \tag{3}$$

To see how that equation can be the same as bilinear interpolation, let's start with the equation for bilinear interpolation, which is just a linear interpolation between two other linear interpolations. We'll use t and u as the interpolation weights of each interpolation. We will linearly interpolate between linear interpolations $E(u)$ and $F(u)$ by t to get out a result of $P(t, u)$

$$\begin{aligned}\mathbf{E(u)} &= \mathbf{P_0 * (1-u) + P_1 * u}\\ \mathbf{F(u)} &= \mathbf{P_2 * (1-u) + P_3 * u}\\ \mathbf{P(t,u)} &= \mathbf{E(u) * (1-t) + F(u) * t}\\ \mathbf{P(t,u)} &= \mathbf{(P_0 * (1-u) + P_1 * u) * (1-t) + (P_2 * (1-u) + P_3 * u) * t}\end{aligned} \tag{4}$$

In our usage case of bilinear interpolation to evaluate Bernstein polynomials, P_0 will be A, P_1 and P_2 will both be B, and P_3 will be C. Also, we set t and u equal in our usage case so we'll replace u with t. We will also replace $(1-t)$ with s for readability.

$$\begin{aligned}\mathbf{P(t)} &= \mathbf{(A * s + B * t) * s + (B * s + C * t) * t}\\ \mathbf{P(t)} &= \mathbf{A * s^2 + B * st + B * st + C * t^2}\\ \mathbf{P(t)} &= \mathbf{A * s^2 + B * 2st + C * t^2}\end{aligned} \tag{5}$$

If we replace A,B,C with P_0, P_1, P_2, and s with $(1-t)$ we then get the familiar equation below, which is the Bernstein form of a quadratic Bezier curve.

$$\mathbf{P(t) = P_0(1-t)^2 + P_1 2(1-t)t + P_2 t^2} \tag{6}$$

This pattern continues for trilinear interpolation mapping to a cubic curve, and beyond. N dimensional linear texture sampling is just the linear interpolation between two linear texture samples of dimension $N-1$. Bezier curves of degree N are just the linear interpolation between two Bezier curves of degree $N-1$. These operations are equivalent.

2.3. Accuracy

A limitation with this technique is that control points can only take values that can be stored and recalled from the specific texture format being used, which puts a limitation on the range of values able to be used, as well as the precision of the values stored.

Furthermore, the fractional position of a texel used in the interpolation formulas is limited to 256 values due to it being calculated with a 9 bit (1.8) fixed point format [NVIDIA 2015].

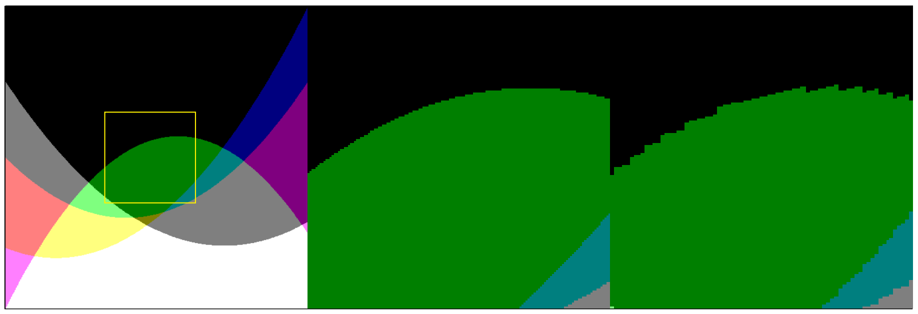

Figure 5. *Left:* Sampled curves. *Middle:* Zooming in on a curve calculated with shader program constants. *Right:* Zooming in on a curve calculated using the method in this paper

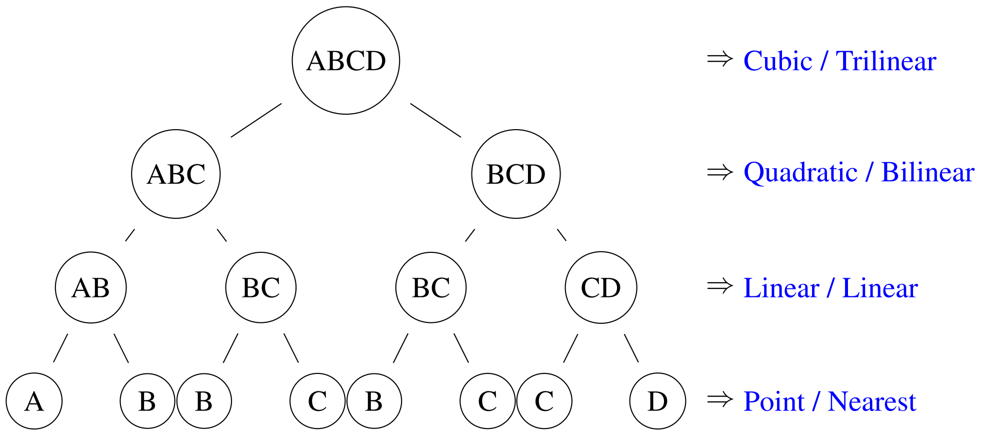

Figure 6. A tree showing the De Casteljeau algorithm for a cubic curve. The labels on the right show what type of curves are evaluated at that level, as well as the N dimensional sampling that is required to evaluate nodes at that level with a single texture read.

The accuracy issues can be seen in Figure 5, although the accuracy can still be significantly higher than a baked out curve using the same number of pixels as seen in Figure 2.

3. Texture Dimensionality

While this technique is able to be used with any texture dimensionality, there are different characteristics for each choice. Those characteristics will be explored here.

3.1. One Dimensional Textures

Using one dimensional textures, the built in texture interpolation is only able to perform linear interpolation, which allows calculations of only 1st degree (linear) Bezier curves (Figure 6).

Since the De Casteljeau algorithm just interpolates between linear interpolations, we can take more texture samples and combine them in the shader program to get higher degree curves. For a degree N Bezier curve, the De Casteljeau algorithm requires 2^{N-1} linear interpolations for the linear level of the tree. You need two pixels to perform a linear interpolation, so it seems that we would need 2^N pixels to store a curve of degree N in a 1 dimensional textures. For a cubic curve, that would look like this: $[P_0, P_1, P_1, P_2, P_1, P_2, P_2, P_3]$.

We can do better than that though. Firstly, there are redundant lerps in the list for lerping between P_1 and P_2. As the degree of the curve increases, the number of redundancies increase as well. We don't want to have to pay the cost of pixel storage and texture samples for redundancies, so because we have $N + 1$ control points and we only ever sample between P_i and P_{i+1}, we only need to store N lerps in the 1 dimensional texture, which would mean we would need $N * 2$ pixels. A cubic curve would then look like this: $[P_0, P_1, P_1, P_2, P_2, P_3]$.

After eliminating redundant lerps, we can see that there are redundant data values in our texture. P_1 and P_2 each show up twice. As the degree of the curve increases, we would notice the pattern that all control points show up twice except the end points. We can remove these redundancies and still be able to get the same information out. The result of removing these redundancies is that our texture only needs to contain each control point once, which means we only need $N + 1$ pixels to encode a curve of degree N. The cubic curve now looks like this: $[P_0, P_1, P_2, P_3]$.

If M piecewise curves are desired, we multiply the number of pixels used for one curve by M. The size of the one dimensional texture needed for M piecewise curves, each of degree N is $M * (N + 1)$. Two piecewise cubic curves encoded into a texture would be 8 pixels in size and look like this: $[P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7]$

It's often the case that you want $C0$ continuity for piecewise curves though, which in the example would give P_4 the same value as P_3. If your piecewise curves always had $C0$ continuity, you could remove the redundant control point for every curve after the first, which would make it so the size of the texture required became $N * M + 1$ (Figure 7).

Taking N linearly interpolated texture samples from a one dimensional texture of size $N + 1$ gives us the linear interpolated values we would need to continue to perform the De Casteljeau algorithm. We can compute the final curve point either by doing that, or by plugging the values into the Bernstein form equation of Bezier curves.

In the case of a cubic curve with four control points, the one dimensional texture will be 4 pixels in size, and three texture reads will need to be done to get the interpolations along the line segments for $\overline{P_0P_1}, \overline{P_1P_2}, \overline{P_2P_3}$. Those will then need to be combined using either the De Casteljeau algorithm, or by using the Bernstein form of a quadratic Bezier curve ($P = P_0 * s^2 + P_1 * 2st + P_2 * t^2$) to elevate those three points from degree 1 to degree 3, to get the final point on the curve. Sample code of

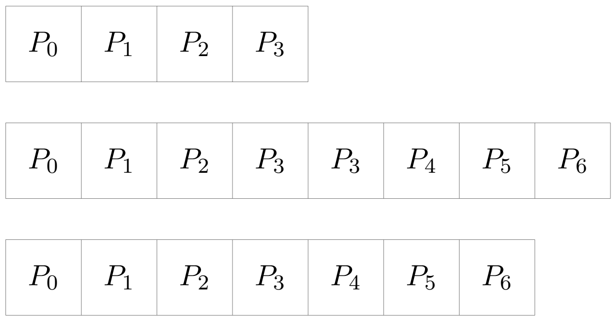
P_0 P_1 P_2 P_3
P_0 P_1 P_2 P_3 P_3 P_4 P_5 P_6
P_0 P_1 P_2 P_3 P_4 P_5 P_6

Figure 7. *Top:* A cubic curve encoded in a 1d texture with a size of 4 pixels. *Middle:* Two piecewise cubic curves encoded in a 1d texture with $C0$ continuity. *Bottom:* With $C0$ continuity, a redundant P_3 can be removed to store these curves in 7 pixels instead of 8.

this example is provided in Listing 2.

```
uniform sampler1D uSampler;
const float c_textureSize = 4.0; // $P_0, P_1, P_2, P_3$

vec4 CubicCurveFromTexture1D_DeCasteljeau(in float t) {
    vec4 P0P1 = texture(uSampler, (t + 0.5) / c_textureSize);
    vec4 P1P2 = texture(uSampler, (t + 1.5) / c_textureSize);
    vec4 P2P3 = texture(uSampler, (t + 2.5) / c_textureSize);
    vec4 P0P1P2 = mix(P0P1, P1P2, t);
    vec4 P1P2P3 = mix(P1P2, P2P3, t);
    return mix(P0P1P2, P1P2P3, t);
}

vec4 CubicCurveFromTexture1D_Bernstein(in float t) {
    vec4 P0P1 = texture(uSampler, (t + 0.5) / c_textureSize);
    vec4 P1P2 = texture(uSampler, (t + 1.5) / c_textureSize);
    vec4 P2P3 = texture(uSampler, (t + 2.5) / c_textureSize);
    float s = (1 - t);
    float s2 = s * s;
    float t2 = t * t;
    // Quadratic Bezier Curve = A*s*s + B*2*s*t + C*t*t
    return P0P1*s2 + P1P2*2.0*s*t + P2P3*t2;
}
```

Listing 2. GLSL for evaluating a cubic curve encoded in a 4 pixel 1d texture. Linear texture sampling is used to evaluate the linear level of the De Casteljeau algorithm, then the process is continued both with the De Casteljeau algorithm, as well as the Bernstein form of a quadratic Bezier curve.

3.2. Two Dimensional Textures

Using two dimensional textures allows the texture interpolator to perform bilinear interpolation, which allows calculations of 2nd degree (quadratic) Bezier curves (Fig-

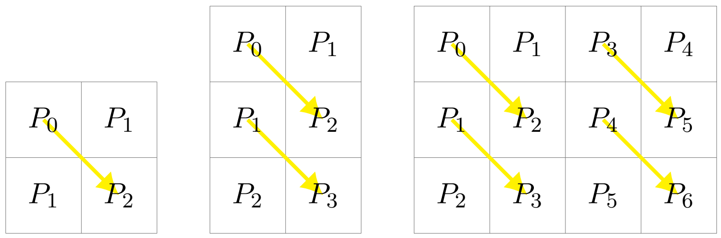

Figure 8. *Left:* A quadratic curve encoded in a 2d texture that is $(2, 2)$ in size. *Middle:* A cubic curve encoded in a 2d texture that is $(2, 3)$ in size. *Right:* Two piecewise cubic curves encoded in a 2d texture with $C0$ continuity that is $(4, 3)$ in size. The yellow lines show how the curves are sampled from the texture over t.

ure 6).

This allows us to either calculate a quadratic Bezier curve with a single texture read, or allows us to get the quadratic level values needed to make a higher degree curve.

Knowing that a quadratic curve is just the linear interpolation at t between the points on two linear curves at t, we have to find a way to make a texture that will cause this to happen with bilinear interpolation. The result is that if we have a 2x2 texture, the top two pixels should be P_0 and P_1 and the bottom two pixels should be P_1 and P_2. When we do a bilinear interpolation such that we plug t into both u and v, the end result will be that the interpolations between $\overline{P_0P_1}$ and $\overline{P_1P_2}$ will be calculated from the u axis interpolations, and then those two will be linearly interpolated across the v axis to give us the final result.

Knowing that we can linearly interpolate between two quadratic curves to create a cubic curve, and that we can combine three quadratic curves to create a quartic curve, the pattern seems to be that we need a texture size of $(2, 2 * (N - 1))$ to encode a curve of degree N into a two dimensional texture.

When actually creating that texture, we would once again see redundancies however. For a cubic curve with control points P_0, P_1, P_2, P_3 we would see that the top row of pixels was P_0, P_1, then the next row was P_1, P_2. That would be our first quadratic curve. The third row would start our second quadratic curve which would be P_1, P_2 and then the fourth and final row would be P_2, P_3. In this case, P_1, P_2 is redundant. If we increased the degree of the curve, the pattern that we would see is that every curve after the first curve would have a redundant row.

Removing that redundant row, it means that we start with a 2x2 texture for a quadratic curve, and then just add a row of pixels for each degree above quadratic. That makes the size of our texture required be only $(2, N)$ where N is greater than 1.

If M piecewise curves are desired, each of degree N, we can encode them across the X axis to require a texture size of $(2 * M, N)$ (Figure 8).

GLSL source code is also provided below to help see how that would work for sampling a cubic curve from a two dimensional texture, with two bilinear texture samples combined to create the final cubic curve.

```
uniform sampler2D uSampler;
const vec2 c_textureSize = vec2(2.0,4.0); // rounded up from 2x3

vec4 CubicCurveFromTexture2D_DeCasteljeau(in float t) {
    vec4 P0P1P2 = texture(uSampler,(vec2(0.5, 0.5)+t)/c_textureSize);
    vec4 P1P2P3 = texture(uSampler,(vec2(0.5, 1.5)+t)/c_textureSize);
    return mix(P0P1P2, P1P2P3, t);
}

vec4 CubicCurveFromTexture2D_Bernstein(in float t) {
    vec4 P0P1P2 = texture(uSampler,(vec2(0.5, 0.5)+t)/c_textureSize);
    vec4 P1P2P3 = texture(uSampler,(vec2(0.5, 1.5)+t)/c_textureSize);
    float s = (1 - t);
    // Linear Bezier Curve = A*s + B*t
    return P0P1P2*s + P1P2P3*t;
}
```

Listing 3. GLSL for evaluating a cubic curve encoded in a $(2, 4)$ pixel 2d texture. Bilinear texture sampling used to evaluate the first two levels of the De Casteljeau algorithm, then the process is continued both with the De Casteljeau algorithm, as well as the Bernstein form of a linear Bezier curve (lerp).

3.3. Three Dimensional Textures

Using three dimensional textures (often referred to as volumetric textures) allows the texture interpolator to evaluate trilinear interpolation, which allows calculations of 3rd degree (cubic) Bezier curves (Figure 6).

This allows us to either calculate a cubic Bezier curve with a single texture read, or allows us to get the cubic level values needed to make a higher degree curve.

Knowing that a cubic curve is just the linear interpolation of t between the points on two quadratic curves at t, we need to find a way to set up a 3d texture to make this happen with trilinear texture sampling. With a 2x2x2 texture, the front 2x2 pixels would encode a quadratic curve for P_0, P_1, P_2 and the back 2x2 pixels would encode a quadratic curve for P_1, P_2, P_3. Then, when we sample the texture, plugging t into u, v and w, the result will be that it will calculate the points on the quadratic curves P_0, P_1, P_2 at t and also P_1, P_2, P_3 at t, and then will interpolate across the w axis to give us the final result.

You can combine two cubic curves to make a quartic curve, or three cubic curves to make a quintic curve. If you were to put 2x2x2 cubic curves back to back, you would notice a redundancy like we saw in lower dimensionality textures. Instead of going that route, we can build on what we did for two dimensional textures.

The size of a three dimensional texture needed to encode a curve of degree N

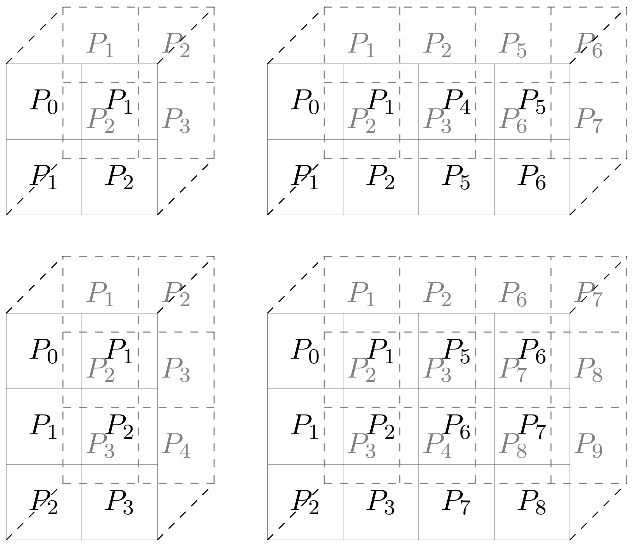

Figure 9. *Top Left:* A cubic curve encoded in a $(2, 2, 2)$ sized texture, decoded with a single trilinear texture sample. *Top Right:* Two piecewise cubic curves encoded in a $(4, 2, 2)$ sized texture, decoded with a single trilinear texture sample. *Bottom Left:* A Quartic curve encoded in a $(2, 3, 2)$ sized texture, decoded with two trilinear texture samples. *Bottom Right:* Two piecewise quartic curves encoded in a $(4, 3, 2)$ sized texture, decoded with two trilinear texture samples.

is $(2, N - 1, 2)$. If you wanted to store M piecewise curves, each of degree N, the formula becomes $(2 * M, N - 1, 2)$ (Figure 9).

Below is some GLSL source code showing how you would sample a cubic curve stored in a 3d 2x2x2 texture, using trilinear texture sampling.

```
// 2x2x2 3d texture.
uniform sampler2D uSampler;
const vec3 c_textureSize = vec3(2.0, 2.0, 2.0);

vec4 CubicCurveFromTexture3D(in float t) {
    return texture(uSampler, vec3(0.5 + t) / c_textureSize);
}
```

Listing 4. GLSL for evaluating a cubic curve encoded in a $(2, 2, 2)$ pixel 3d texture. Trilinear texture sampling used to evaluate all three levels of the De Casteljeau algorithm.

3.4. Summary of Texture Dimensionality

Increasing texture dimensionality allows you to evaluate the same degree curve with fewer texture reads but comes at the cost of using more texture memory. We evaluated texture dimensions one, two and three, but the pattern continues for higher dimensions as well.

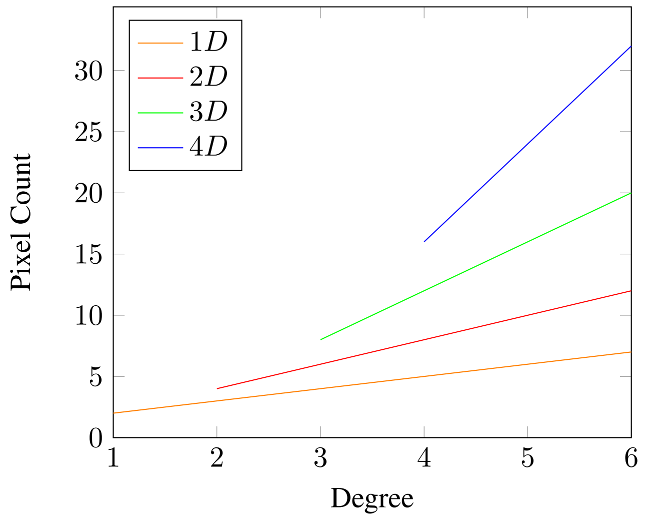

Figure 10. Pixel count for curve degree, with different texture dimensionalities.

The table below is a comparison of texture dimensionality options. M is the number of piecewise curves being stored and N is the degree of the curve. To get an idea of how dimensionality and degree affects the size of the required textures, take a look at Figure 10.

	Min. Degree	**# Samples**	**Dimensions**	**# Pixels**
1D	1 (Linear)	N	$(M*(N+1))$	$M*(N+1)*1$
2D	2 (Quadratic)	$N-1$	$(2*M,N)$	$M*(N+0)*2$
3D	3 (Cubic)	$N-2$	$(2*M,N-1,2)$	$M*(N-1)*4$
4D	4 (Quartic)	$N-3$	$(2*M,N-2,2,2)$	$M*(N-2)*8$

4. Extensions

There are a few ways to extend this technique for more options.

4.1. Combining Color Channels

If you encode Bezier curves into a texture which has multiple color channels - such as the standard four R,G,B,A - you can set up the textures such that once you read in the R,G,B,A, you can combine the channel values together such that you get a fewer number of curves per texture read, but that they are higher degree.

For instance, if you had a 2x2 texture, you could encode the control points P_0, P_1, P_2 into the red channel, and the control points P_1, P_2, P_3 into the green channel. Then, after you did a texture read, you could do a lerp between the red and the green channel values by t. Since the red and green channel each contain a quadratic curve, the result would be a cubic curve, defined by the control points P_0, P_1, P_2, P_3 However, you've used up two of the channels for a single curve, so are getting fewer curves per texture read, but they are higher degree. The blue and the alpha channel could either be quadratic curves, or could also be encoded such that they could be combined together to provide another cubic curve.

You could also set it up such that R,G,B,A were meant to be combined together to make a quintic (degree 5) curve. You'd only get a single curve value per texture read, but it would be 3 degrees higher. You could combine the color channels either with De Casteljeau's algorithm, or with the cubic Bernstein Bezier form.

If you you are combining M color channels, each of which are values from curves of degree N, the resulting curve value will be degree $N + M - 1$. The intuitive explanation to this, is that each channel has the ability to add one more control point to the curve, regardless of what texture dimensionality you are working with, and each control point added raises the curve one degree.

An interesting side effect to combining color channels is that the accuracy of the resulting curve improves. This is because you are doing some of the math in full precision floating point, instead of the limited precision of the fixed point math in the texture sampler.

4.2. Piecewise Curves

As mentioned earlier, you can encode multiple Bezier curves end to end within textures of any dimensionality.

When you are in need of a curve with lots of control points, piecewise curves have several benefits over increasing the degree of the curve.

The main benefit of piecewise curves are that they allow us to use lower degree curves, which allows us to use lower dimensional textures, and have less computation in the shader programs.

Another benefit is that there can be discontinuities between curve segments, which is sometimes desired by artists and graphic designers.

B-splines can also be converted to piecewise Bezier curves using Boehm's algorithm, so even though this technique only explicitly deals with Bezier curves, B-splines could be converted to piecewise Bezier curves and could be used in that form.

Evaluating a piecewise curve with this technique is very simple. As an example, let's say that we have a texture that is 8x2, and encodes 4 quadratic Bezier curves end to end. The left most 2x2 pixels encode the quadratic curve used at $t\epsilon[0, 0.25)$. The next block of 2x2 pixels encode the quadratic curve used at $t\epsilon[0.25, 0.5)$. The third block encodes $t\epsilon[0.5, 0.75)$, and the last block encodes $t\epsilon[0.75, 1.0]$.

4.3. Rational Bezier Curves

Rational Bezier curves are useful because they can be used to make shapes which cannot be formed from integral Bezier curves, including perfectly representing conic

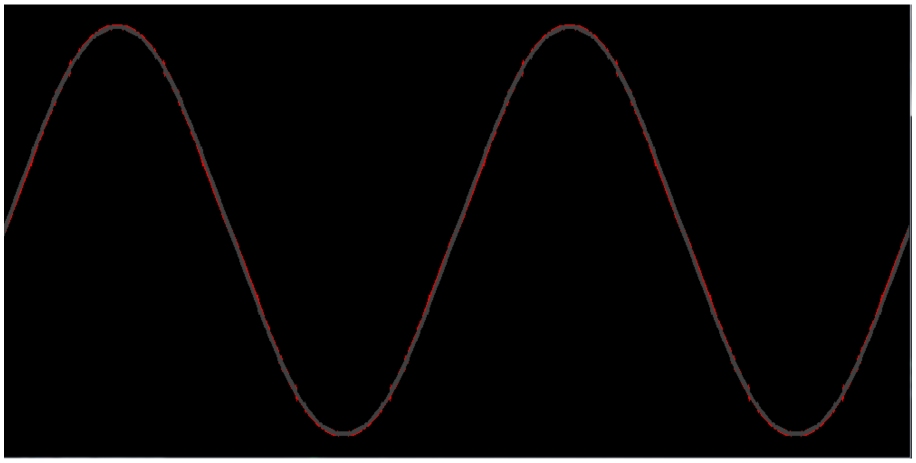

Figure 11. Rational Bezier spline vs glsl sin() function. Sin() values in grey on top of the Bezier spline in red. Red pixels show where the values don't match.

sections. This also means that they can accurately represent sine and cosine as seen in Figure 11. Rational Bezier curves are defined by the equation:

$$\mathbf{B}(t) = \frac{\sum_{i=0}^{n} \binom{n}{i}(1-t)^{n-i}t^i w_i \mathbf{P}_i}{\sum_{i=0}^{n} \binom{n}{i}(1-t)^{n-i}t^i w_i} \tag{7}$$

If we were to multiply the weight into the control point for the numerator, We can see that we essentially have one Bezier curve divided by another. Using the techniques in this paper, we could easily encode the numerator into one color channel, and the denominator into another.

If we wanted to get two rational Bezier curves per texture read, we could encode the two numerators into R and G, and the denominators into B and A.

Or, if we had three curves that all used the same weights, that would mean that the denominator was the same for all three curves. In this case, we could encode the three numerators into R,G,B and encode the denominator in A, which would give us three rational curves with a single texture read.

To compute the value of rational Bezier curves, you first take your texture samples, combining them as per normal if there were multiple taken (if needed), and then after that, you divide the numerators by the denominators to get the final values.

There is a limitation to encoding rational Bezier curves in this form due to the fact that texture values range from 0 to 1, but when using rational Bezier curves, weights greater than 1 are sometimes desired. The good news is that you can normalize the weights by dividing all weight values by the largest weight value, such that the largest weight value becomes 1. The result will be the same as if you had values greater than 1 in the weights.

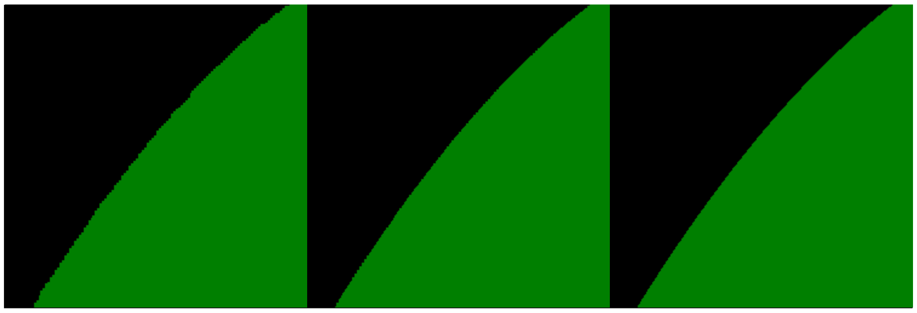

Figure 12. Quality versus texture read count. *Left:* One Texture Read (one bilinear sample). *Middle:* Two Texture Reads (two linear samples). *Right:* Three Texture Reads (three nearest neighbor samples).

Just as b-splines can be converted into piecewise Bezier curves, NURBS can also be converted into piecewise rational Bezier curves, again using Boehm's algorithm, which means these techniques can also apply to NURBS, if they are first preprocessed into piecewise rational Bezier curves.

4.4. Multidimensional Bezier Curves

So far we have only talked about one dimensional — or explicit — Bezier curves which take the form of $y = f(x)$. Multidimensional Bezier curves are evaluated separately per axis though, which means that you can encode information per axis in different color channels.

To name a few possibilities with a standard RGBA 2x2 texture, you could encode a fourth dimensional Bezier curve, or you could encode 2 two dimensional Bezier curves, or you could even encode a three dimensional rational Bezier curve using R,G,B for the numerator of each axis, and A as the denominator for all three curves.

5. Addressing Accuracy Issues

Even though there are accuracy limitations in modern GPU texture samplers, accuracy can be increased by doing more of the calculations in the shader program, which operate at full floating point precision.

A naive way to do this would be to use nearest neighbor texture sampling instead of linear, and do the linear interpolations yourself in the shader program. This would mean twice as many texture reads with a 1D texture, four times as many texture reads with a 2D texture, and eight times as many texture reads with a 3D texture. You would also be doing a lot more blending in the shader program, so your shader programs would become heavier weight. This solution is essentially doing software linear texture sampling.

A better approach that gives the same results would be to only do one nearest

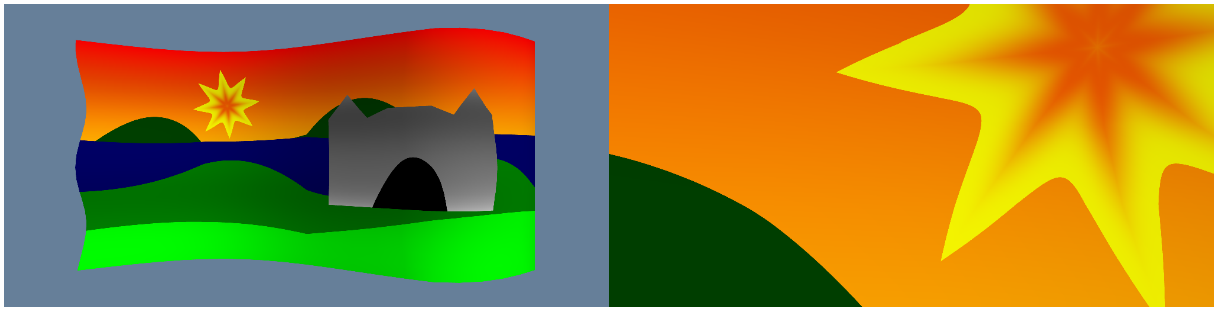

Figure 13. *Left:*An animated flag textured with vector graphics using the techniques in this paper, using an 8x4 RGBA source texture to generate curves and color gradients. *Right:* Zooming in on part of the image to show the quality of the vector graphics.

neighbor texture sample per control point in the texture, and then in the shader program, either use De Casteljeau's algorithm to combine them, or use the Bernstein form equation of Bezier curves. At this point, you are best off just using a 1D texture to store your control points in, since you don't need any of the built in interpolation.

These two techniques do all of the curve math in the shader program at full floating point precision and so are the highest quality, but are also the most computationally expensive.

Another way to address accuracy would be to do some of the work in the texture sampler, and some of the work in the shader program. This allows you to have a sliding scale for accuracy vs cost.

For instance, if you had a 2x2 image which encoded a quadratic Bezier curve, you might do two texture reads, letting the texture sampler do the x axis interpolation, and then you could linearly interpolate those results in the shader program in full precision floating point math for the y axis. This would give you a result somewhere between full hardware and full software interpolation. You can compare the results in Figure 12.

Also, if you want a higher degree curve than your texture can provide with a single read, forcing you to combine multiple curve points, combining those curve points in the shader program has the nice side benefit of increasing accuracy as well.

6. Limited Applications for Vector Graphics

This technique does have applications to vector graphics, but they are pretty limited and there are better solutions if using Bezier curves as vector art is the goal [Loop and Blinn 2005; Nehab and Hoppe 2008], or if the goal is to use textures for vector art data storage [Green 2007]. The limitation is due to the fact that this technique is good at calculating the point on a curve at t, but when using curves in vector graphics situations, you usually do not know the t that you want to know the curve point for, and it is usually difficult to find it.

However, there are certain special situations where you can easily get the t value of the curve that you care about. Two of the most common situations for these are when rendering 1D explicit Bezier curves, and when rendering 1D explicit Bezier curves in polar form.

Explicit Bezier curves are curves which have scalar control points and are in the form of $y = f(x)$. For these curves, when you plug in a value for x you get a y value out. They are unable to make curves that have multiple y values for the same x, and the curves can't cross over themselves to make closed shapes. This does make them not as useful to vector graphics as 2D Bezier curves which are of the form $(x, y) = f(t)$, but they still do have some use cases. Rendering explicit Bezier curves using this technique is very simple. If rendering a quad, you just choose either the x or y axis, normalize it from 0 to 1 and use that to calculate the other axis value by doing a texture sample. At this point, you can either choose a half space to color in, or you can calculate the gradient of the function (by using the built in dFdx style functions to get partial derivatives), and use that as an estimated distance to the curve.

This idea can easily be extended to 1D Bezier curves rendered in polar form. If rendering a quad, you can use arc tangent to find the angle of the current pixel from the center, normalize that angle to 0 to 1, and then use that value to do a texture look up to calculate the value of the curve at that point. Once again you can then either choose a half space to color in, or you can use the gradient information to get an estimated distance to the curve, to draw a curved line.

Another problem with using this technique for vector graphics, is that since the texture sampler is doing at least some of the calculations for us, we don't have access to the values of the control points, which can make some operations harder. Luckily, one property of Bezier curves is that affine transformations done to control points is equivalent to affine transformations done to points on the curve. This means that we can do affine transformations on the curve points provided to us by the texture sampler, and it will be as if we had performed them on the control points themselves.

Lastly, in regards to vector graphics, this technique can be used to allow people to define custom color gradients. The benefit is that custom, non linear color gradients can be defined, and can be zoomed into without decaying to linear interpolation.

A demonstration of these usage cases can be seen in Figure 13.

7. Performance Characteristics

The table below was created by running a 1000x1000 pixel shader program where each pixel evaluated a curve using either linear texture sampling (HW or hardware evaluation), or did multiple texture reads to get the control points and then calculated the curve values inside of the shader program (SW or software shader program evaluation). The numbers shown are the time spent rendering each frame, in microseconds and the texture format is R8G8B8A8. Actual performance characteristics in real world scenarios will be based on other factors, such as whether the texture is in the texture cache, or whether you are texture fetch bound or compute bound in the shader program.

	NV GF GTX 980m		NV GF GTX 770	
	HW	**SW**	**HW**	**SW**
1D Texture / Linear Curve	131.8 μs	132.7 μs	103.0 μs	104.1 μs
2D Texture / Quadratic Curve	131.1 μs	135.1 μs	104.1 μs	118.2 μs
3D Texture / Cubic Curve	135.4 μs	146.2 μs	104.1 μs	130.1 μs

The performance characteristics don't change very much based on texture dimensionality. This shows that you can up the degree of the curve by upping the dimension of the texture you store the curve in, without incurring any significant performance costs. While software implementations are a bit slower, they are not significantly slower. Again, the usage case for deciding whether to offload curve work to the sampler (and if so, how much of it), or to do it in the shader program depends on your specific circumstances, and the resource usage of your application.

Future Work

In the past, work has been done to evaluate curves on the GPU using textures, but for the purposes of texture interpolation [Ruijters et al. 2008]. The progress made on those types of techniques could likely be used in a similar fashion to directly evaluate other curve types on the GPU for purposes other than texture sampling.

Extending this technique to the third dimension to make surfaces would be an interesting usage case, as well as exploring ways to make the vector art usage cases more attractive.

There may also be usefulness in curves in polar form to approximate lobes for lighting algorithms.

Lastly, throughout this paper, texture coordinates were set up such that the texture samples were taken only across the diagonal of N-dimensional hypercubes. It would be interesting to explore whether deviating from the diagonal created useful properties, more complex curves, or more complex curve types so that even more usefulness could be achieved with the same number of pixels used.

Acknowledgements

I would like to give my deepest thanks to Ian McCown, James Anhalt III and Dr. Josh Grass for reading this over and giving feedback before submission. You all had significant responsibilities at the time, ranging from having a young newborn to moving to another continent, and as people I have a lot of respect for, I really appreciate that all three of you made time to have a look and give me meaningful feedback. Thank you also to my wife Chanel for your support and patience during this process!

References

GREEN, C. 2007. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, ACM, New York, NY, USA, SIGGRAPH '07, 9–18. URL: `http://doi.acm.org/10.1145/1281500.1281665`, doi:10.1145/1281500.1281665. 17

LOOP, C., AND BLINN, J. 2005. Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph. 24*, 3 (July), 1000–1009. URL: `http://doi.acm.org/10.1145/1073204.1073303`, doi:10.1145/1073204.1073303. 17

NEHAB, D., AND HOPPE, H. 2008. Random-access rendering of general vector graphics. *ACM Trans. Graph. 27*, 5 (Dec.), 135:1–135:10. URL: `http://doi.acm.org/10.1145/1409060.1409088`, doi:10.1145/1409060.1409088. 17

NVIDIA. 2015. F.2 linear filtering. In *CUDA C Programming Guide*, NVIDIA. [Online; accessed 15-July-2015]. URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#linear-filtering`. 6

RUIJTERS, D., TER HAAR ROMENY, B. M., AND SUETENS, P. 2008. Efficient gpu-based texture interpolation using uniform b-splines. *Journal of Graphics, GPU, and Game Tools 13*, 4, 61–69. arXiv:http://dx.doi.org/10.1080/2151237X.2008.10129269, doi:10.1080/2151237X.2008.10129269. 19

Index of Supplemental Materials

TODO: put information about where the zip can be downloaded.

The zip file contains a folder named WebGL which contains the WebGL demos, and a folder named GLFW which contains the GLFW (windows) C++ OpenGL demos.

WebGL Folder

- *index.html* — An index of the demos.
- *demo.html* — A sandbox for exploring the various options of the technique.
- *flag.html* — A vector art demo showing a waving flag and allowing you to modify some of the options used when rendering.
- *rational.html* — Rational curve demo, showing sine and cosine compared to their actual values as computed by a shader program.

GLFW Folder

- *Curves* — The folder that contains the source code and the project and solution file for the demos.
- *glew-1.12.0* — The glew library used for the demos.
- *glfw-3.1.1* — The glfw library used for the demos.

Author Contact Information

Alan Wolfe
Blizzard Entertainment
16215 Alton Parkway
Irvine, CA 92618
awolfe@blizzard.com http://blog.demofox.org

Alan Wolfe, GPU Efficient Texture Based Bezier Curve Evaluation, *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, no. 1, 1–1, 2014
http://jcgt.org/published/0003/01/01/

Received:	2014-02-07		
Recommended:	2014-02-07	Corresponding Editor:	Editor Name
Published:	2014-02-07	Editor-in-Chief:	Morgan McGuire