GPU Efficient Texture Based Bezier Curve Evaluation

Alan Wolfe
Blizzard Entertainment

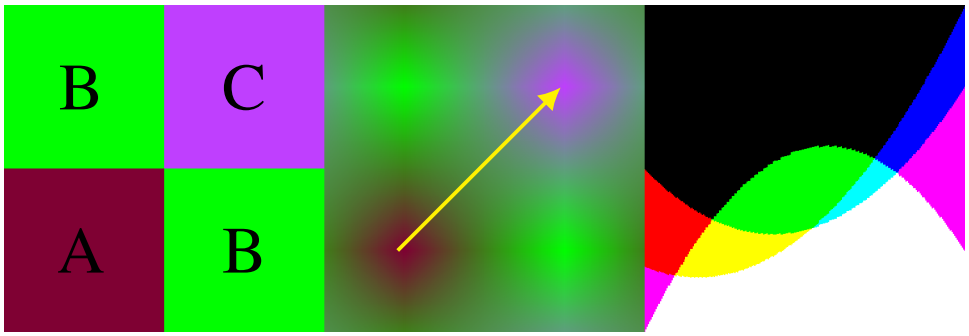

Figure 1. *Left:* 2x2 texture containing control points for a quadratic Bezier curve in each color channel. *Middle:* The texture as viewed with bilinear sampling. *Right:* The resulting curves when sampling along the yellow line (Alpha curve omitted).

Abstract

Modern graphics techniques expose internal parameters to allow re-use and to help seperate the technical implementation from artistic usage cases. A popular choice is to expose parameters as customizable curves and to quantize the curves into textures. This leads to either lower quality results, or more texture memory being used to accomodate 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 order, resulting in higher fidelity for the same costs. Piecewise curves, B-Splines and Nurbs are mentioned, and there are also limited applications towards vector graphics.

1. Introduction

There are two basic ways for a shader program to gain access to 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 data from the CPU to the shader program as shader constants.

 ISSN 2331-7418

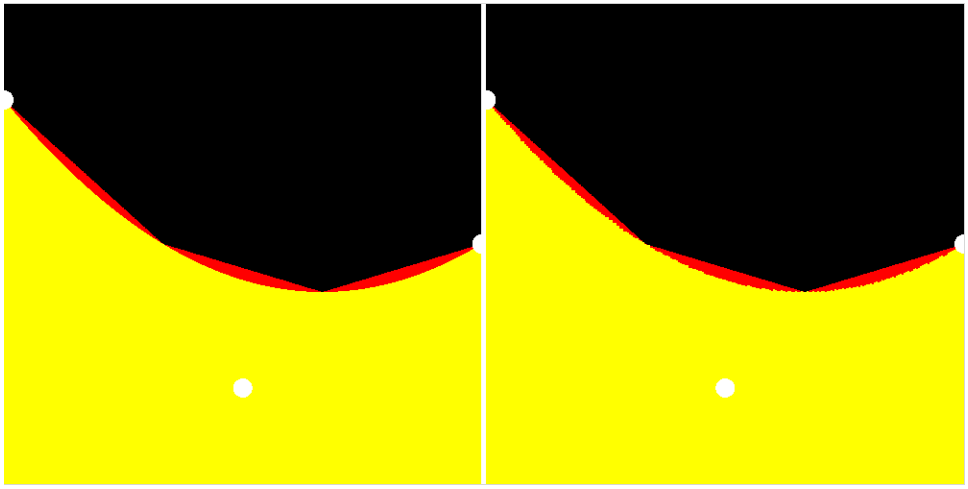

Figure 2. *Left:* A quadratic Bezier curve written to the green channel, and the baked out equivelant using 4 pixels written to the red channel. *Right:* The technique from this paper, also using 4 pixels, written to the green channel, and the baked out equivelant written to the red channel again for comparison.

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 has, 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 achieve this by increasing the resolution of the texture to be able 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 in a very specific way for the type of curve you want to use, as well as more shader program instructions to calculate specific curve points.

This paper shows a third method where:

- Curve data is encoded within a texture in such a way to allow the texture sampler to calculate 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 order 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.

1.1. Related Work

TODO: this

2. The Technique

The core of this technique is that the linear texture interpolation capabilities on the GPU can be mathematically equivelant to De Casteljeau's algorithm for evaluating Bezier curves.

A one dimensional texture with linear sampling results in the evaluation of a linear Bezier curve (order 1), a two dimensional texture with linear sampling can result in a quadratic bezier curve (order 2), and a three dimensional texture (volumetric texture) with linear sampling can result in a cubic Bezier curve (order 3).

The pattern continues for higher dimensionality textures, and curves of lower degrees can be combined in the shader program to create curves of higher degrees by either continuing De Casteljeau's algorithm or by using the Bernstein form of Bezier curves.

2.1. Intuition

The De Casteljeau algorithm shows us that we can find points on a Bezier curve by evaluating a hierarchy of linear interpolations. For example, a quadratic curve is the linear interpolation between the linear interpolation of $\overline{AB}$ and $\overline{BC}$, using the same time t for each interpolation.

Bilinear interpolation is used when linear sampling is enabled and a texture sample is taking from a 2d texture. Bilinear interpolation is achieved by linearly interpolating across one axis (say, the X axis) and then interpolating the two results by the other axis (the Y axis). In bilinear interpolation, you have a t and a u interpolation value, and you have four values to interpolate between A,B,C,D.

```
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);
}

float QuadraticBezier (
  in float t,
  in float A, in float B, in float C
) {
    return mix(mix(A,B,t), mix(B,C,t), t);
```

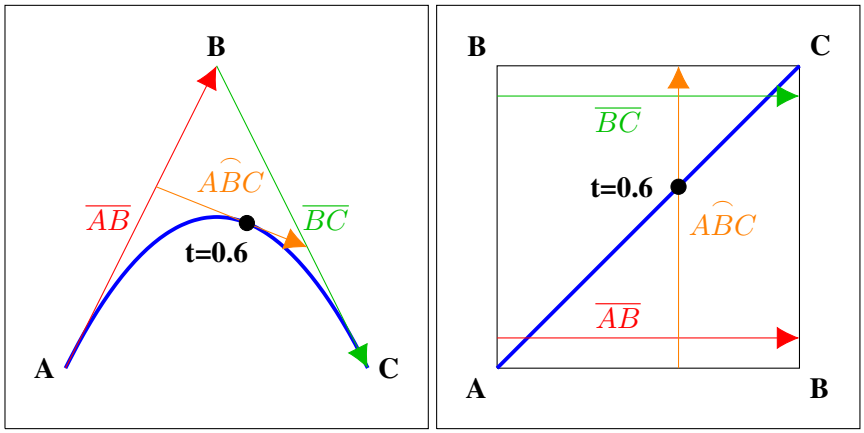

Figure 3. *Left:* A quadratic Bezier curve evaluated at **t**=0.6 for control points A,B,C using the De Casteljeau algorithm. *Right:* The De Casteljeau algorithm using bilinear interpolation. Note that in this case, the X axis is evaluated before the Y, but the Y axis could be evaluated before the X for the same results.

```
}
```

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

Comparing the source code for these two operations (Listing 1), we begin to get an idea of how to set up the points in our texture and the u and v coordinates to use such that when performing bilinear sampling, that we will get, as output, points on a quadratic bezier curve (Figure 10).

Since we need to linearly interpolate between $\overline{AB}$ and $\overline{BC}$, and between those results, all by the same time t, we need to ensure that our u coordinate equals our v coordinate.

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 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).

2.2. Mathematical Basis

The De Casteljeau algorithm is equivelant to the Bernstein form of Bezier curves.

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

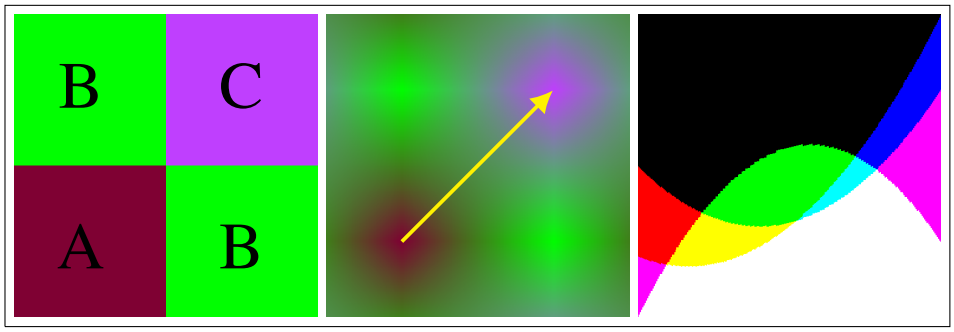

Figure 4. *Left:* a 2x2 texture storing control points for a quadratic Bezier curve in each color channel. *Middle:* The same texture as viewed when using bilinear texture sampling. The yellow line indicates where texture samples are taken from to evaluate the quadratic Bezier curve. *Right:* The curves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

That they are equivelant 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.

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

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.

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

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, t and u are the same in

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

our usage case so we'll just replace u with t. We will also replace $(1-t)$ with s for readability.

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

If we replace A,B,C with P_0, P_1, P_2, and s with $(1-t)$ we then get the familiar equation back out.

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

This pattern continues for trilinear interpolation and beyond

TODO: How to prove for N dimensions? This section needs help TODO: could somehow show how N-linear sampling is same as De Casteljeaus for all N's? tree diagram or something else.

2.3. Accuracy

While the Bernstein form of Bezier curves is equivelant to evaluating a Bezier curve with the De Casteljeau algorithm, in practice there are differences that come up between the two. It is well known that the De Casteljeau algorithm is slower but more numerically robust than evaluating the Bernstein form.

The technique presented in this paper also has numerical differences in practice. While it's been shown that the multidimensional linear interpolation capabilities can be mathematically equivelant to the Bernstein form as well as the De Casteljeau algorithm, there are mathematical precision issues due to current GPU architectures (Figure 5).

As shown in Figure 4, the quality can be significantly higher than a baked out curve for the same number of pixels however. There is more information on addressing accuracy in section 5.

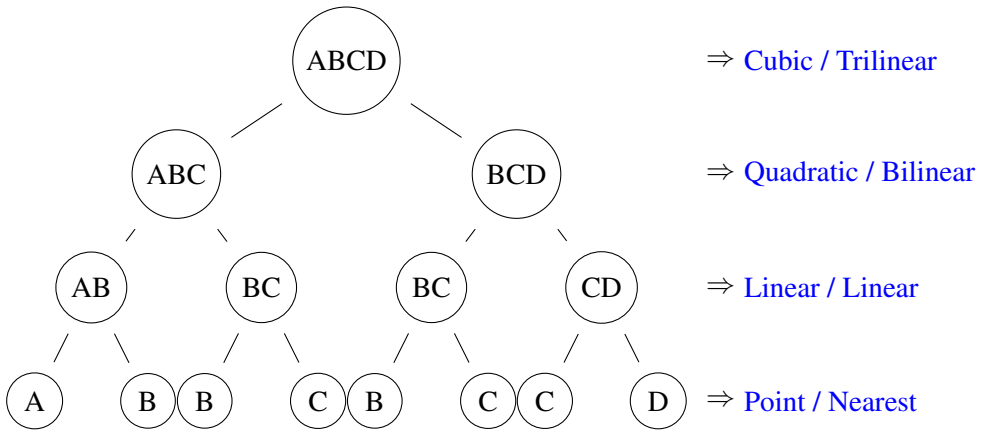

Figure 6. A tree showing the De Casteljeau algorithm for a cubic curve. The labels on the right show what type of curves are evaluated at that level, as well as the N dimensional sampling that is required to evaluate nodes at that level with a single texture read.

TODO: give the details of the architecture imprecision sources and cite the two papers cited from the other place!

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 order (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 order curves. For an order 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 order 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 order 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

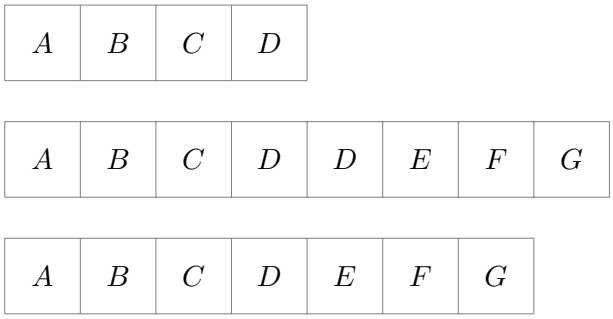

Figure 7. *Top:* A cubic curve with control points A, B, C, D encoded in a 1d texture with a size of 4 pixels. *Middle:* Two piecewise curves encoded in a 1d texture. The control points for the first curve are A, B, C, D and the control points for the second curve are D, E, F, G. *Bottom:* With $C0$ continuitiy, a redundant D can be removed to store these curves in 7 pixels instead of 8.

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 order 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 order 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 control points A, B, C, D, the one dimensional texture will be 4 pixels in size, and three texture reads will need to be done to get the interpolations for $\overline{AB}, \overline{BC}, \overline{CD}$. Those will then need to be combined using either

the De Casteljeau algorithm, or by using the Bernstein form of a quadratic Bezier curve ($P = A * s^2 + B * st + C * t^2$) to elevate those three points from order 1 to order 3, to get the final point $A\overset{\frown}{BC}D$. Sample code of this example is provided in Listing 2.

```
// 4 pixel 1d texture with control points encoded:  A,B,C,D
uniform sampler1D uSampler;
const float c_textureSize = 4.0;

vec4 CubicCurveFromTexture1D_DeCasteljeau(in float t) {
    vec4 AB = texture(uSampler, (t + 0.5) / c_textureSize);
    vec4 BC = texture(uSampler, (t + 1.5) / c_textureSize;
    vec4 CD = texture(uSampler, (t + 2.5) / c_textureSize);
    vec4 ABC = mix(AB, BC, t);
    vec4 BCD = mix(BC, CD, t);
    return mix(ABC, BCD, t);
}

vec4 CubicCurveFromTexture1D_Bernstein(in float t) {
    vec4 AB = texture(uSampler, (t + 0.5) / c_textureSize);
    vec4 BC = texture(uSampler, (t + 1.5) / c_textureSize);
    vec4 CD = 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 AB*s2 + BC*2.0*s*t + CD*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 order (quadratic) Bezier curves (Figure 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 order curve.

Knowing that a quadratic curve is just the linear interpolation of time t between the points on two linear curves at time t, we know that our two dimensional texture needs to set up to be able to make this happen. 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

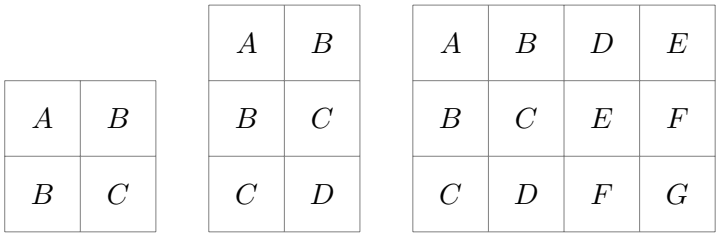

Figure 8. *Left:* A quadratic curve with control points A, B, C encoded in a 2d texture that is $(2, 2)$ in size. *Middle:* A cubic curve with control points A, B, C, D encoded in a 2d texture that is $(2, 3)$ in size. *Right:* Two piecewise curves encoded in a 2d texture that is $(4, 3)$ in size. The control points for the first curve are A, B, C, D and the control points for the second curve are D, E, F, G.

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 $\overline{P_0P_1}$ and $\overline{P_1P_2}$ will be calculated from the u axis interpolation, 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 order 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 order of the curve, the pattern that we would see is that every curve after the first curve would have a redundant row.

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

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).

```
// 2x4 2d texture, rounded to the nearest powers of 2 from 2x3.
uniform sampler2D uSampler;
const vec2 c_textureSize = vec2(2.0,4.0);

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

```
vec4 CubicCurveFromTexture2D_Bernstein(in float t) {
    vec4 ABC = texture(uSampler, (vec2(0.5, 0.5)+t) / c_textureSize);
    vec4 BCD = texture(uSampler, (vec2(0.5, 1.5)+t) / c_textureSize);
    float s = (1 - t);
    // Linear Bezier Curve = A*s + B*t
    return ABC*s + BCD*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 levesl 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 refered to as volumetric textures) allows the texture interpolator to evaluate trilinear interpolation, which allows calculations of 3rd order (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 order curve.

Knowing that a cubic curve is just the linear interpolation of time t between the points on two quadratic curves at time t, we know that our three dimensional texture needs to be set up to be able to make this happen. 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 time t and also P_1, P_2, P_3 at time 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 though, we can just build on the redundancy removement from two dimensional textures.

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

```
// 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);
```

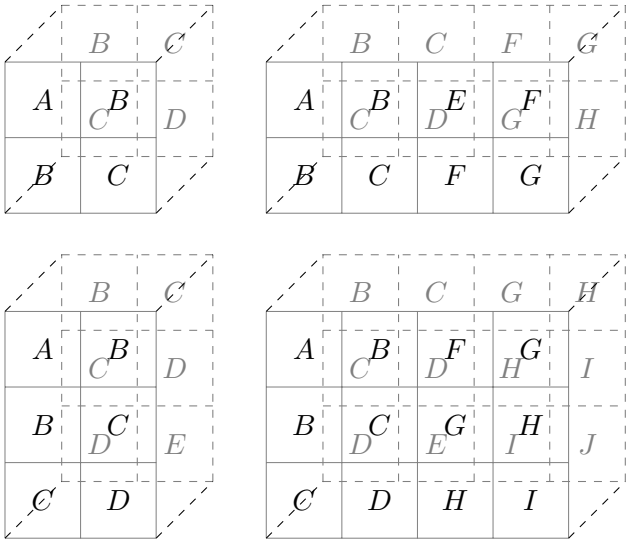

Figure 9. *Top Left:* A cubic curve encoded in a $(2, 2, 2)$ sized texture, decoded with a single trilinear texture sample. *Top Right:* Two piecewise cubic curves encoded in a $(4, 2, 2)$ sized texture, decoded with a single trilinear texture sample. *Bottom Left:* A Quartic curve encoded in a $(2,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

```
}
```

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 order 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.

The table below is a comparison of texture dimensionality options. M is the number of piecewise curves being stored and N is the order of the curve.

	Min. Order	**# 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)	$N - 3$	$(2 * M, N - 2, 2, 2)$	$M * (N - 2) * 8$

TODO: do we want this graph? it does show pixel usage per dimensionality

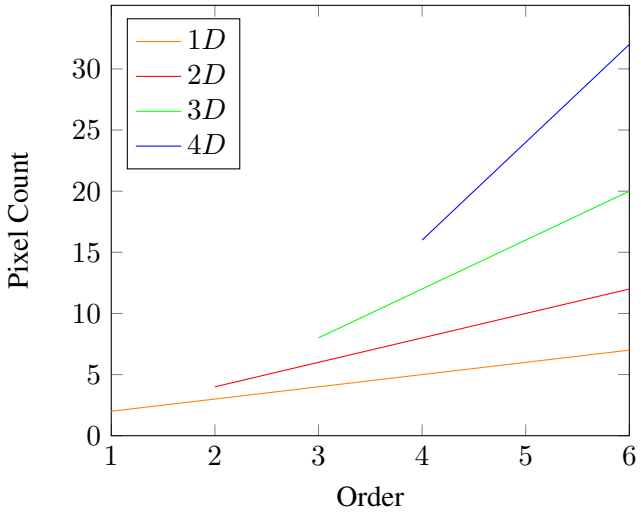

Figure 10. Pixel count for curve order, with different texture dimensionalities.

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 order.

For instance, if you had a 2x2 texture, you could encode the control points A,B,C into the red channel, and the control points B,C,D into the green channel. Then, after you did a texture read, you could do a lerp between the red and the green channel. Since the red and green channel each contain a quadratic curve, the result would be a cubic curve, defined by the control points A,B,C,D. 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 order. 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 Casteljeaus algorithm, or with the cubic Bernstein Bezier form.

If you you are combining M color channels, each of which are values from curves of order N, the resulting curve value will be order N+M-1. The inutitive 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.

An interesting side effect to combining color channels is that the accuracy of the resulting curve gets better. 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 order of the curve.

Besides the usual benefits, when encoding Bezier curves in textures, piecewise curves allow us to use lower order 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.

B-splines can also be converted to piecewise Bezier curves using Boehm's algorithm, so even though this paper talks only about Bezier curves, you could do pre-processing of B-splines to convert them to piecewise Bezier curves, and use the techniques in this paper.

For a deeper look at the benefits of piecewise curves, look at.... TODO: find a paper that talks about the pros and cons?

and for a deeper discussion about the benefits of b-spline curves, look at.... TODO: find a paper that talks about benefits of bsplines over bezier curves

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 for time $t\epsilon[0, 0.25)$. The next block of 2x2 pixels encode the quadratic curve used for time $t\epsilon[0.25, 0.5)$. The third block encodes time $t\epsilon[0.5, 0.75)$, and the last block encodes time $t\epsilon[0.75, 1.0]$.

4.3. Rational Bezier Curves

Rational Bezier curves are defined by the equation:

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

Looking closely, we can see that we essentially have one Bezier curve devided by another. The numerator has a weighting and a control point, while the denominator

has only a weighting, but if we were to multiply the weighting into the control point in the numerator, we would essentially have one Bezier curve divided by another.

Using this technique, we can 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.

To compute the value of rational Bezier curves, you first take your texture samples, combining them as per normal if there were multiple taken, 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.

Rational Bezier curves have many uses, as they have much more flexibility over the shapes they can take, including the ability to exactly represent all conic sections. You can find more information about that at... TODO: find paper about rational bezier curves?

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 rational Bezier curves.

4.4. Multidimensional Bezier Curves

So far we have only talked about one dimensional - or explicit - bezier curves. Multidimensional Bezier curves are evaluated seperately per axis though, which means that you can encode information per axis in different color channels.

With a standard R,G,B,A 2x2 texture, you could either encode four 1 dimensional Bezier curves, or you could encode two 2 dimensional Bezier curves (one in R,G and the other in B,A), or you could encode a three dimensional curve in R,G,B and have a one dimensional curve in A, or you could encode a four dimensional curve in R,G,B,A.

5. Addressing Accuracy Issues

TODO: this. HW vs SW vs HWSW hybrid vs read points yourself and do math w/o using interpolator. Everythign in between. TODO: images to show differences. TODO: doing more work in shader, because using 1d textures or soemthing. more texture reads but higher quality

6. Limited Applications for Vector Graphics

TODO: this TODO: mention that it isn't ideal since this technique is only good if you already know time "t" to evaluate, which is difficult to get otherwise. TODO: mention that you don't have access to the control points since the interpolator does that level of math for you, but that affine transformations on the resulting curve point is equivelant to doing then on the control points. TODO: mention 1d curve usage TODO: mention polar curve usage TODO: mention use for color gradients TODO: picture of flag TODO: put real usages here: particle properties, and whatever else.

7. Comparisons With Other Techniques

TODO: real apples to apples comparisons with texture baking (can be any order!) as well as shader constants.

8. Performance Characteristics

TODO: Show some real performance values. TODO: show difference between 3d texture doing cubic, vs 2d texture with 2 texture reads and shader program results. TODO: show results from multiple GPUs?

Future Work

TODO: this

Acknowledgements

TODO: this

References

TODO: this

Index of Supplemental Materials

TODO: this

Author Contact Information

Roy G. Biv
Colortech, Inc.
29 Red Blvd.
New York, NY 10511
roy@colortech.com

Raymond Trace
Graphica University
37 Rue De Lambert
Paris, 75009 France
rtrace@graphica.edu
http://graphica.edu/~rtrace

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