GPU Efficient Texture Based Bezier Curve Evaluation

Alan Wolfe

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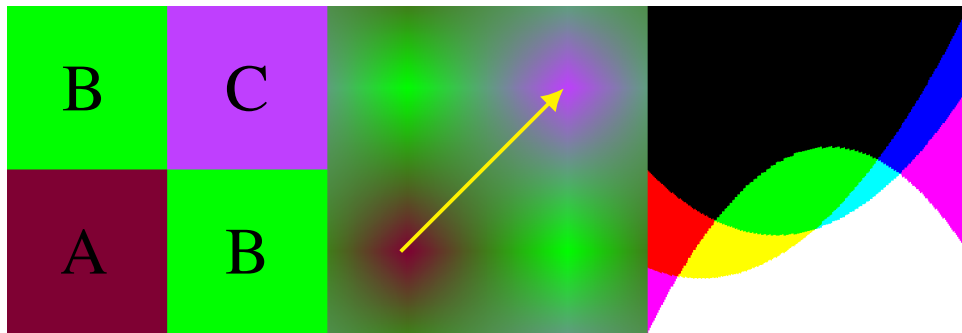
Figure 1. *Left:* 2x2 texture containing control points for a quadratic Bezier curve in each color channel. *Middle:* The texture as viewed with bilinear sampling. *Right:* The resulting curves when sampling along the yellow line (Alpha curve omitted).

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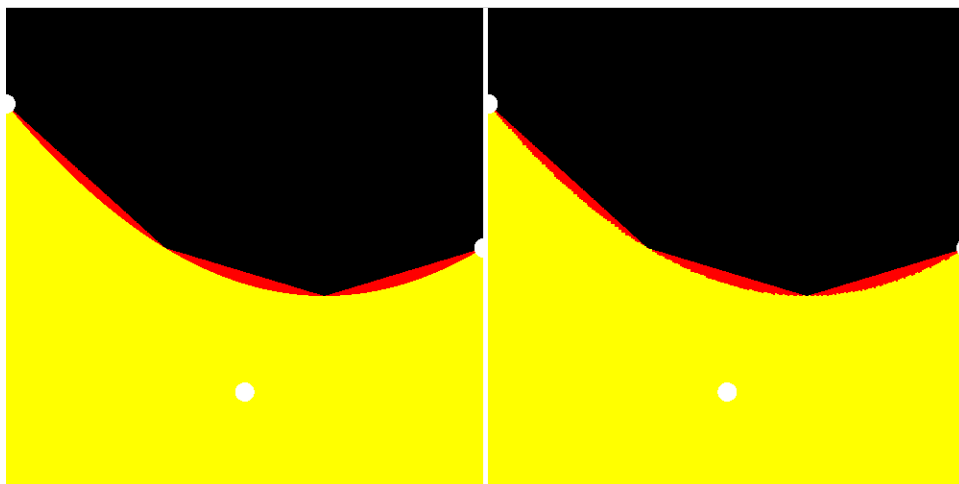
Figure 2. *Left:* A quadratic Bezier curve written to the green channel, and the baked out equivelant using 4 pixels written to the red channel. *Right:* The technique from this paper, also using 4 pixels, written to the green channel, and the baked out equivelant written to the red channel again for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

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.

```glsl
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);
}
```

Listing 1. GLSL implementation of bilinear interpolation and the De Casteljeau algorithm for a quadratic Bezier curve.
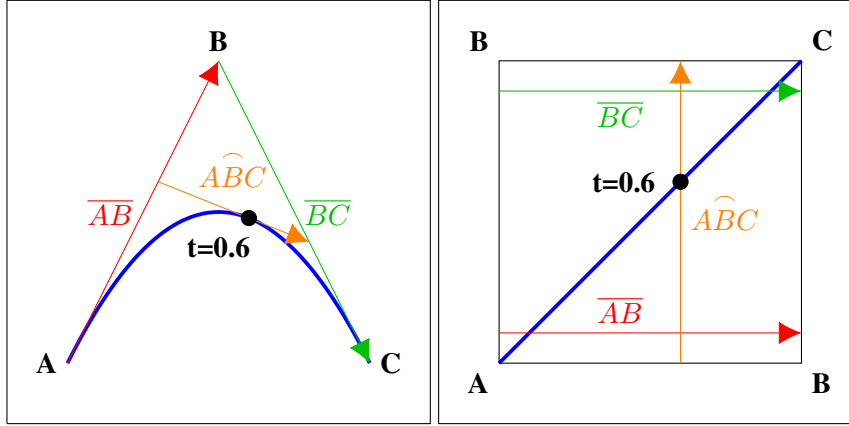
Figure 3. *Left:* A quadratic Bezier curve evaluated at **t**=0.6 for control points A,B,C using the De Casteljeau algorithm. *Right:* The De Casteljeau algorithm using bilinear interpolation. Note that in this case, the X axis is evaluated before the Y, but the Y axis could be evaluated before the X for the same results.

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 3).

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.

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

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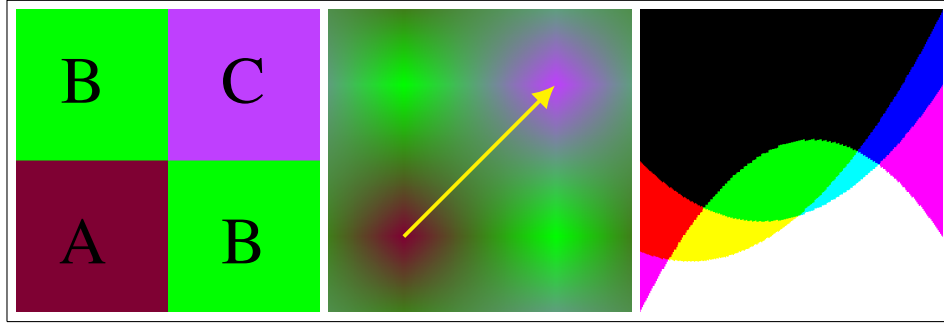
Figure 4. *Left:* a 2x2 texture storing control points for a quadratic Bezier curve in each color channel. *Middle:* The same texture as viewed when using bilinear texture sampling. The yellow line indicates where texture samples are taken from to evaluate the quadratic Bezier curve. *Right:* The curves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

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.

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

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.

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

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

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

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

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.

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

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.

TODO: give the details of the architecture imprecision sources and cite the two papers cited from the other place!
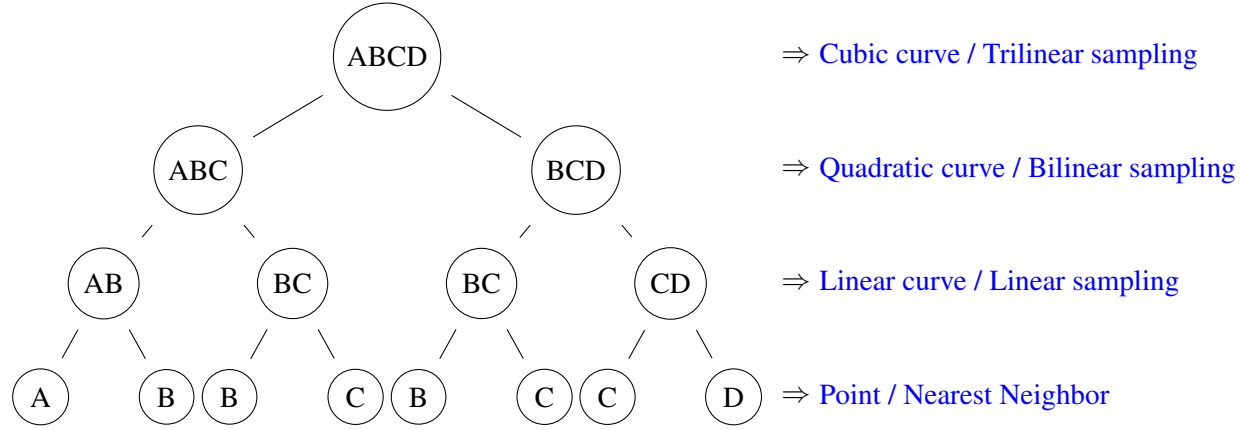
Figure 6. A tree showing the De Casteljeau algorithm for a cubic curve. The labels on the right show what type of curves are evaluated at that level, as well as the N dimensional sampling that is required to evaluate nodes at that level with a single texture read.

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

One dimensional textures are textures which have W pixels in a single row or column, such as $(8, 1)$, $(16, 1)$ or more generally $(W, 1)$.

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 basis functions (Figure 6).

If you want a Bezier curve higher than order 1, you can take more texture samples and combine the results in the shader program. While this is more expensive computationally than moving to a texture of higher dimensionality, it does have the benefit of doing more of the work in the shader program, which has higher precision through use of full float math, so the end result will be a higher quality curve.

The size of the one dimensional texture needed for M piecewise curves, each of degree N is $(N + 1) * M$. For instance, a cubic (degree 3) curve requires 4 pixels $((3 + 1) * 1)$. If you want to encode two piecewise cubic curves end to end, it requires 8 pixels $((3 + 1) * 2)$.

If $C0$ continuity is desired for piecewise curves, a redundant control point can be removed from the beginning of each curve after the first curve, which makes the size of texture required become $N * M + 1$, requiring only 7 pixels for two cubic curves, instead of 8, or 10 pixels for three cubic curves, instead of 12 (Figure 7).

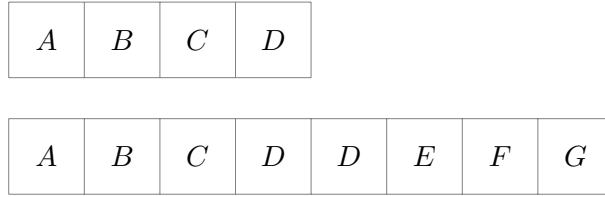In the case of a cubic curve with control points A, B, C, D, the one dimensional

A	B	C	D

A	B	C	D	D	E	F	G

Figure 7. *Top:* A cubic curve with control points A, B, C, D encoded in a 1d texture with a size of 4 pixels. *Bottom:* 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. If C0 continuitiy is always desired between the curves, a redundant D could be removed to store these curves in 7 pixels instead of 8.

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 (order 2) to elevate those three points from order 1 to order 3, to get the final point $\widehat{ABCD}$.

```glsl
// 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 used to evaluate the first 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.

A	B
B	C
C	D

A	B	D	E
B	C	E	F
C	D	F	G

Figure 8. *Left:* 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.

3.2. Two Dimensional Textures

Two dimensional textures are textures which are (W, H) pixels in size, such as $(8, 8)$ or $(16, 16)$ or $(32, 4)$.

Using two dimensional textures allows the texture interpolator to perform bilinear interpolation, which allows calculations of 2nd order (quadratic) Bezier basis functions (Figure 6).

If you want a Bezier curve higher than order 2, once again you can take more texture samples and combine the results in the shader program. While more computationally expensive that moving to a higher dimensionality texture, it again results in higher precision results due to more of the computation happening in full floating point precision.

The size of the two dimensional texture needed for M piecewise curves, each of degree N is $(2 * M, N)$. For instance, a cubic (degree 3) curve will be $(2, 3)$ pixels. If you want to encode two piecewise cubic curves end to end, it requires $(4, 3)$ pixels (Figure 8).

```glsl
// 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

Three dimensional textures (often refered to as volumetric textures) are textures which are MxNxO pixels in size such as 8x8x2 or 16x8x4.

Using three dimensional textures allows the texture interpolator to evaluate trilinear interpolation, which allows calculations of 3rd order (cubic) Bezier basis functions.

Again, you can take more texture samples and combine the results in the shader program to get higher than order 3 curves if desired.

TODO: flesh out details, show diagram of texture layout. TODO: GLSL for cubic and quartic, but images for cubic, quartic with comparison vs ground truth. TODO: explain how linear and quadratic could still be achieved even if it isn't that useful (ye? Hw/sw) TODO: beyond 3d textures? Could do 4d with mipmapped volumetric textures!

3.4. Summary of Implications of Texture Dimensionality

TODO: dimensionality / curve order comparisons. A table? Maybe side by side images? TODO: texture size changes, number of textures used, number of texture samples taken TODO: include quadrilinear and above? TODO: we might want to do this in the performance results section instead and provide perf metrics as well.

4. Extensions

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

4.1. Combining Color Channels

TODO: this. Higher order curves without more texture reads

4.2. Piecewise Curves

TODO: this. Pros and cons. B-splines can be converted to piecewise Bezier with Boem's algorithm (what's it called again?).

4.3. Rational Bezier Curves

TODO: this. TODO: Nurbs can be converted to rational bezier curves with Boem's algorithm as well

4.4. Multidimensional Bezier Curves

TODO: this. Each color channel is an axis.

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.

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   Raymond Trace
Colortech, Inc.   Graphica University
29 Red Blvd.   37 Rue De Lambert
New York, NY 10511   Paris, 75009 France
roy@colortech.com   rtrace@graphica.edu
                                 http://graphica.edu/˜rtrace