MONOSCOPIC RENDERING OF A HOLOGRAM EFFECT AT INTERACTIVE FRAMERATES.

USING DX9, VERTEX SHADERS 2.0 AND PIXEL SHADERS 2.0

BY DAN EVANS, 23RD MARCH 2003

HTTP://WWW.BTINTERNET.COM/~DANBGS

INTRODUCTION

Before we dive into the theory, here are some samples from the output of the app and some screen shots of the media used for the hologram image.

CONCEPT: "HOLOGRAM SPACE"

This is essentially a mapping between each incremental pixel that is rasterised in screen space and a blend of one or more texels from a set of N*N frames depending on the angle between the viewer and the surface normal. The following figure highlights the seams that appear if only one pixel is used (i.e. no blend).

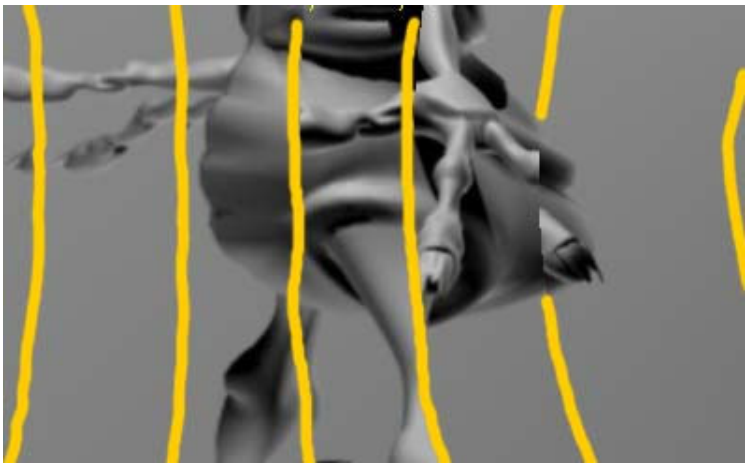


Fig 3.
Seams between frame boundaries in Hologram Space

The key point to Hologram Space is that the animation frames selected from the texture must be sampled at a position corresponding to the point in texture space. The following diagram explains further what is meant by this.

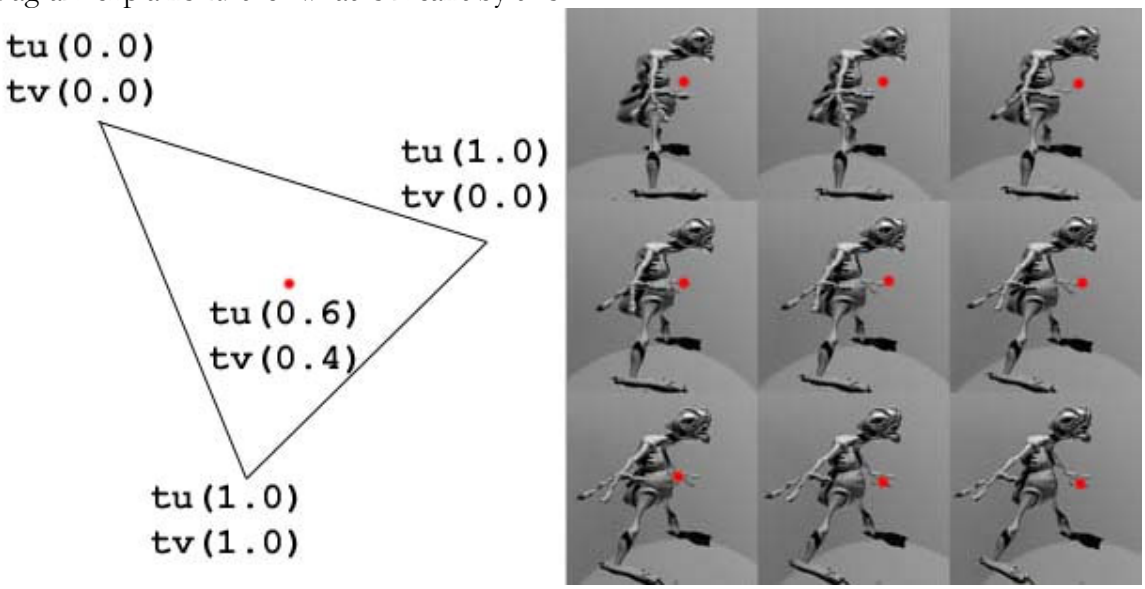


Fig 4.
Illustration of Hologram Space mapping

METHOD

There are a few different ways of implementing this effect. Essentially we want, at each pixel rendered, to choose a texel from the same part of "hologram space" but to select which frame of the pre-rendered animation to choose it from

THE ESSENCE OF THE APPROACH

The basic underlying principle of this implementation is a function that takes a 1D scalar value from a dot product (but needn't be only that) and uses it as a lookup into a 2D texture. That 2D texture is eventually mapped back onto the polygon in such a way as to appear to be different aspects of a single image. The technique has many other potential uses other than for the hologram effect outlined in this paper.

With respect to using the result of (E•N) as the input to this mapping, I'm sure it would be a useful way to represent unfading structural colour (as opposed to pigmented colour, which fades) as found in butterflies wings and various insects found in nature.

There may even be uses for this technique that use the results of cos and sin as inputs. It's an area for a possible follow up paper.

APPENDIX A

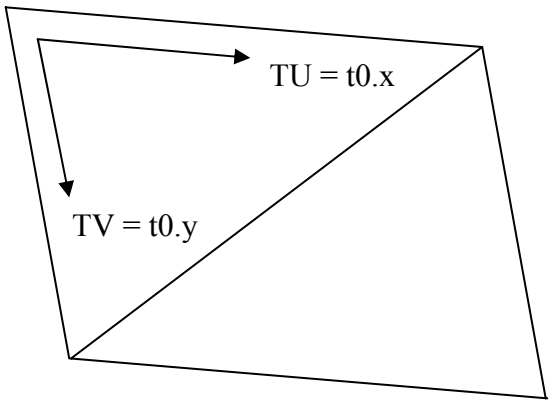
THE PIXEL SHADER STEP BY STEP

Step 1 – Interpolated Texture Coordinates

We're storing the surface tangent (a vector in the XZ plane with a +ve X component) in the vertex normals. The dot product referred to throughout this document is (Eye•T) and is sometimes called "DP"

We want this mapping for the TU direction;

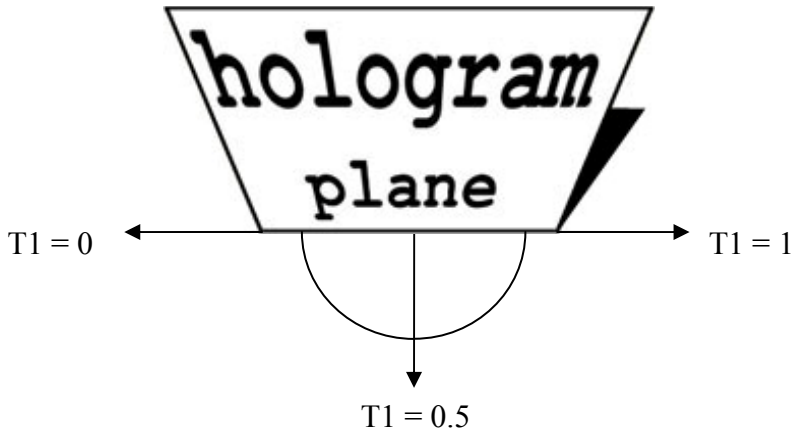
$0 \le (Eye \bullet T) \le 1/3$ $1 \le (Eye \bullet T) \le 1\,1/3$ $2 \le (Eye \bullet T) \le 2\,1/3$	$1/3 \le (Eye \bullet T) \le 2/3$ $1\,1/3 \le (Eye \bullet T) \le 1\,2/3$ $2\,1/3 \le (Eye \bullet T) \le 2\,2/3$	$2/3 \le (Eye \bullet T) \le 1$ $1\,2/3 \le (Eye \bullet T) \le 2$ $2\,2/3 \le (Eye \bullet T) \le 3$
Maps to 0	Maps to $1/3$	Maps to $2/3$




```
mov r1, t0 ; put the ordinary tu, tv's in r1 (value should be in range 0..1)
```

Step 2 – Interpolated the re-normalized dot product

The Vertex shader does the dot product and re-normalizes it. *(N.B. note to self : This might be an issue? Perhaps it needs to be renormalized in the pixel shader)*




```
mov r2, t1 ; put the scaled & biased dot product in r2
```

Step 3 – Scale up dot product by N^2

```
rcp r3.x, c0.x    ; this is actually used in later steps
mul r4.x, r2.x, c0.x
mul r4.x, r4.x, c0.x
```

We now have a value that maps 0..1 to 0..N^2. In our case a 3x3 grid means 0..9

Step 4 – Get Fractional Part of N^2 scaled DP

```
frc r5.x, r4.x
```

Step 5 – Use step 4 result to get whole part of N^2 scaled up dot product

```
sub r5.x, r4.x, r5.x
```

Step 6 – Scale this whole part down by column width (1/n)

```
mul r5.x, r5.x, r3.x
```

We now have a value that's between 0 and n (e.g. 0..3)
Note, this is the step at which we need to add 1/n and store it in a separate register and also do the same with subtracting. We then complete the following steps on all three registers to give our 3 look-ups to blend between.

Step 7 – Get the fractional part from the result of step 6

```
frc r5.x, r5.x
```

This is our absolute "tu left" in the texture. We will interpolate in the range
From `tu_left` to `tu_left + column width`
Where `column width = 1/n`

- For example, say the interpolated dot product from step 2 at this pixel was 0.9 (don't forget these calculations are per pixel).
- Then if our texture had 3x3 pictures in it (so n=3) step 3 would give us a value of 0. 9 * 3 * 3 = 8.1
- The fractional part of this from step 4 is 0.1
- Step 5 gives us 8.1 – 0.1 = 8
- Step 6 gives us 8 / 3 = 2.666666 (note this is correctly inside the range 0..3)
- Step 7 yields 0.666666

So 0.66666 is our `tu_left` and 0.66666 + 1/3 is our `tu_right` (note tu_right is merely a concept which represents the upper tu bound and is never actually used as a variable in the pixel shader).
From this you can clearly see that it's a straight forward matter to take the 0..1 interpolated texture coordinate (see step 1), scale it down by the column width and add it to `tu_left` to get our complete function. Now from all angles, every interpolated pixel across the triangle has a valid look up for the tu direction.

That just leaves the tv direction.

Step 8 – Scale down the original texture coordinates

Scale original TU down by 1/n (i.e. shrink by 1/nth)

```
mul r1.x, r1.x, r3.x
```

Scale original TV down by 1/n (i.e. shrink by 1/nth)

```
mul r1.y, r1.y, r3.x
```

The reason for doing this is because we want our interpolation from one vertex to the other over a triangle to be mapped to a smaller portion of the whole texture so that just the frame we want is interpolated for the current pixel. In other words for one particular view direction (or if the viewer were at infinity) we would get this mapping

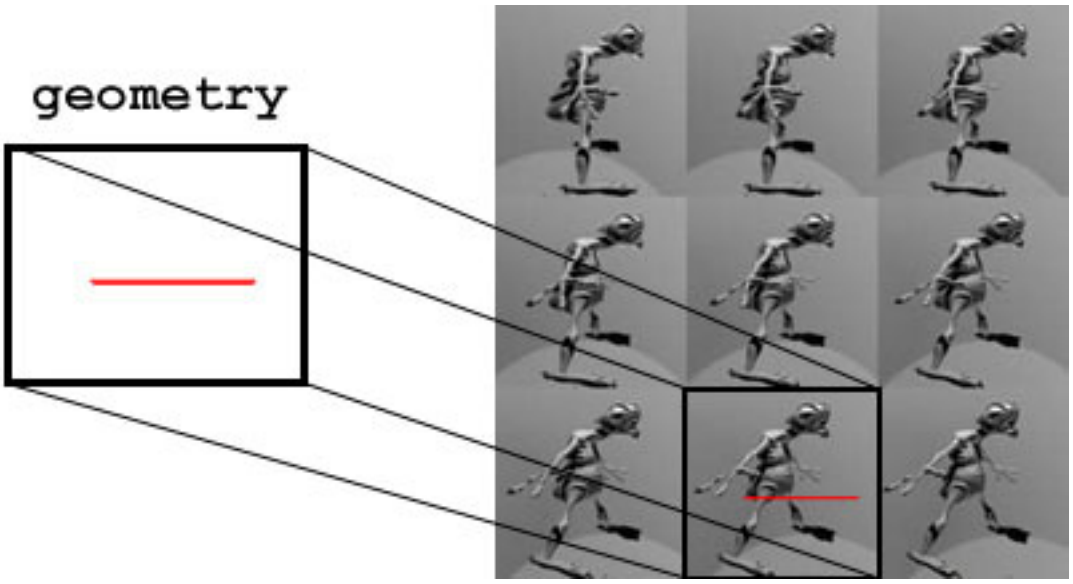


Fig. 5
Illustration of mapping

Step 9 – add step 8 result to our "tu left" offset

```
add r5.x, r5.x, r1.x
```

At this point, our algorithm is working for "left to right" interpolations in the TU axis. Now we have to find which TV offset into our texture page at which to begin our interpolations in the vertical TV direction.

Step 10 – store result of Scalar DP * N in a register

```
mul r4.w, c0.x, r2.x    ; DP * n
```

What we want this mapping for the TV direction;
0<=DP<=1 maps to 0
1<=DP<=2 maps to 0.333333
2<=DP<=3 maps to 0.666666

In our example, N=3 and 0<=DP<=1 so we're scaling up to 0<=DP<=3.

Appendix B

SCREEN SHOTS AND FURTHER ILLUSTRATIVE FIGURES

Fig 1.

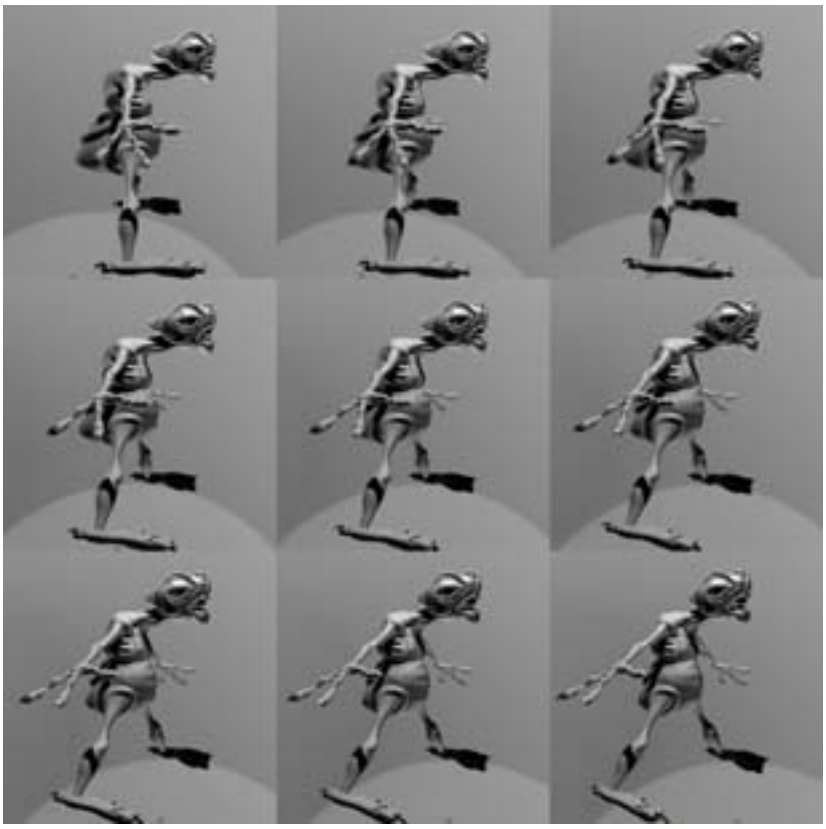


Fig 2.