

Computer Graphics (CS 543)

Lecture 11b: Image Manipulation

Prof Emmanuel Agu

*Computer Science Dept.
Worcester Polytechnic Institute (WPI)*





Reference Book for Image Manipulation

- Graphics Shaders, Cunningham and Bailey, 2nd edition

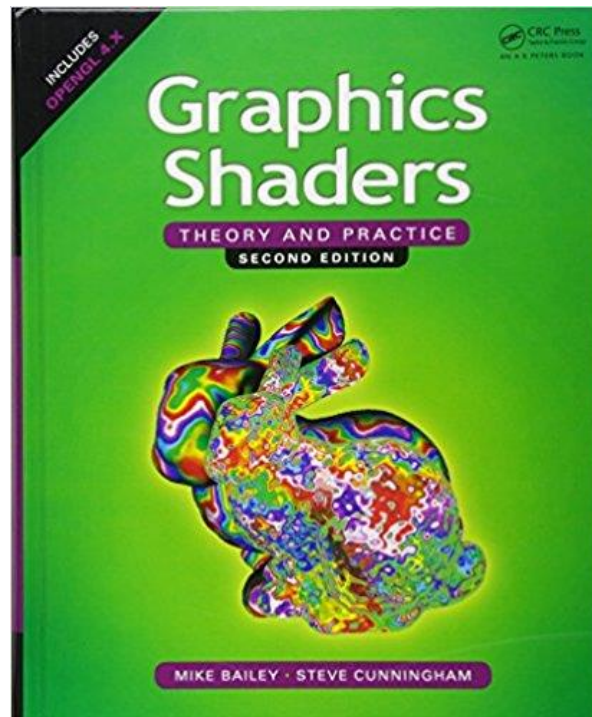




Image Processing

- Graphics concerned with creating artificial scenes from geometry and shading descriptions
- Image processing
 - Input is an image
 - Output is a modified version of input image
- Image processing operations include altering images, remove noise, super-impose images

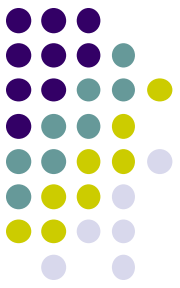


Image Processing

- Example: Sobel Filter



Original Image



Sobel Filter

Image Processing



- Image processing the output of graphics rendering is called **post-processing**
- To post-process using GPU, rendered output usually written to offscreen buffer (e.g. color image, z-depth buffer, etc)
- Image in offscreen buffer treated as texture, mapped to screen-filling quadrilateral
- Fragment shader invoked on each element of texture
 - Performs calculation, outputs color to pixel in color buffer
- Output image may be
 - Displayed, saved as a texture, output to a file



Image Manipulation Basics

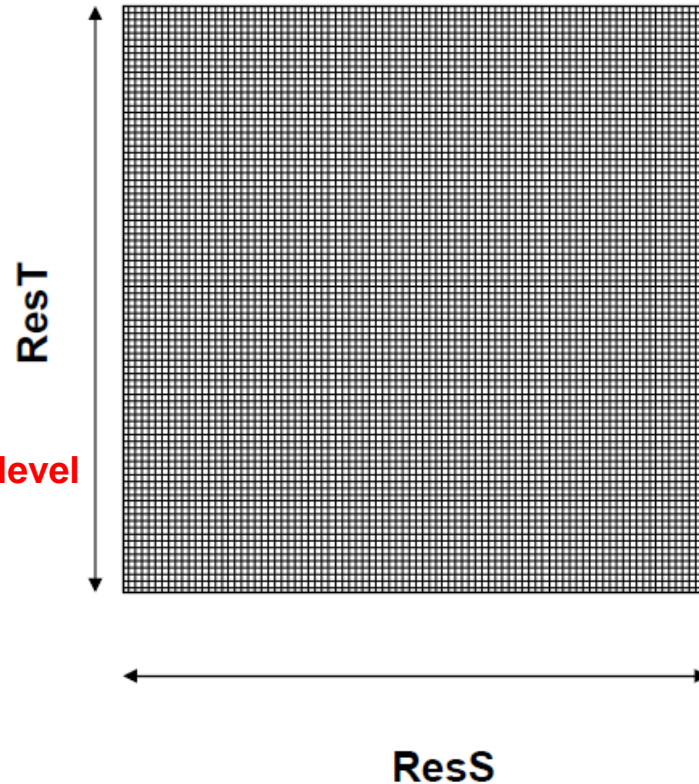
Treat the image as a texture.
The resolution of this texture
can be found by saying:

```
ivec2 ires = textureSize( ImageUnit, 0 );  
float ResS = float( ires.s );  
float ResT = float( ires.t );
```

Texture

Mipmap level

To get from the current texel to a
neighboring texel, add
 $\pm (1./ResS, 1./ResT)$
to the current (S,T)



Note: Since S and T range from 0 to 1
- Image center is at `vec2(0.5, 0.5)`



Vertex Shader

- Most image processing in fragment shader
- Vertex shader just sets texture coordinates

```
out vec2 vST;
```

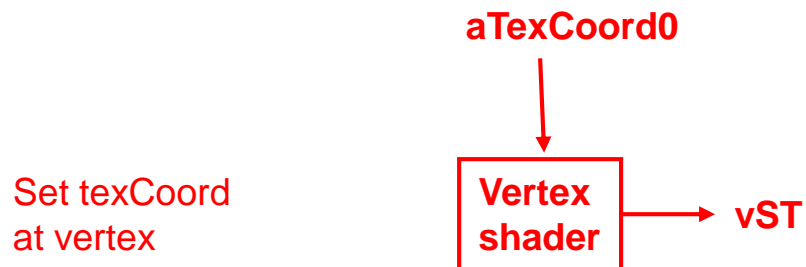
```
Void main( )
```

```
{
```

```
    vST = aTexCoord0.st;
```

```
    gl_Position = uModelViewProjectionMatrix * aVertex;
```

```
}
```



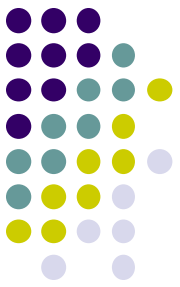


Luminance

- Luminance of a color is its **overall brightness**
- Given a color in R G B,
- Compute luminance by multiplying by a set of weights (0.2125, 0.7154, 0.0721). i.e.

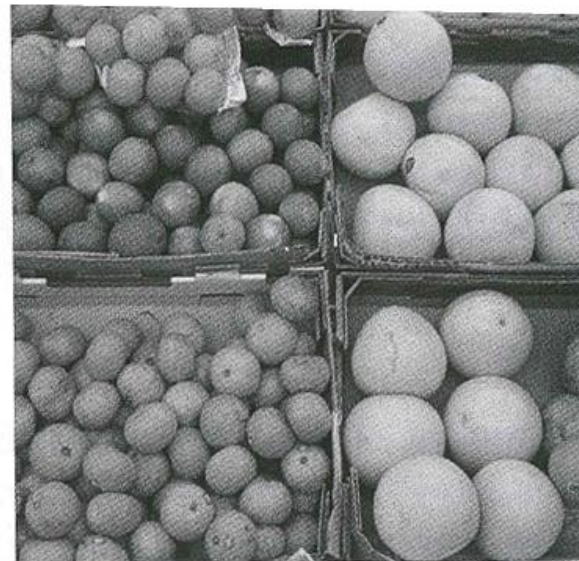
$$\text{Luminance} = \mathbf{R} * 0.2125 + \mathbf{G} * 0.7154 + \mathbf{B} * 0.0721$$

- Note that sum of weights **0.2125 + 0.7154 + 0.0721 = 1**



Code (Fragment Shader) for Luminance

```
const vec3 W = vec3(0.2125, 0.7154, 0.0721);  
vec3 irgb = texture( uImageUnit, vST).rgb; ← // look up RGB of texel at vST  
float luminance = dot(irgb, W);  
  
fFragColor = vec4( luminance, luminance, luminance, 1.);
```



Color with
R = G = B is
Shade of
gray

Image Negative



- Another example



(R, G, B)



(1.-R, 1.-G, 1.-B)



Image Filtering

- A filter convolves (weighted addition?) a pixel with its neighbors
- Different algorithms have different filter sizes (how many neighbors) and weight values



Original Image

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$



*Sobel Filter
applied*



Sobel Filter

What is a Filter?



- **Filters:** combine a pixel's value with its neighbors
- **E.g:** Compute average intensity of block of pixels (Blurring)

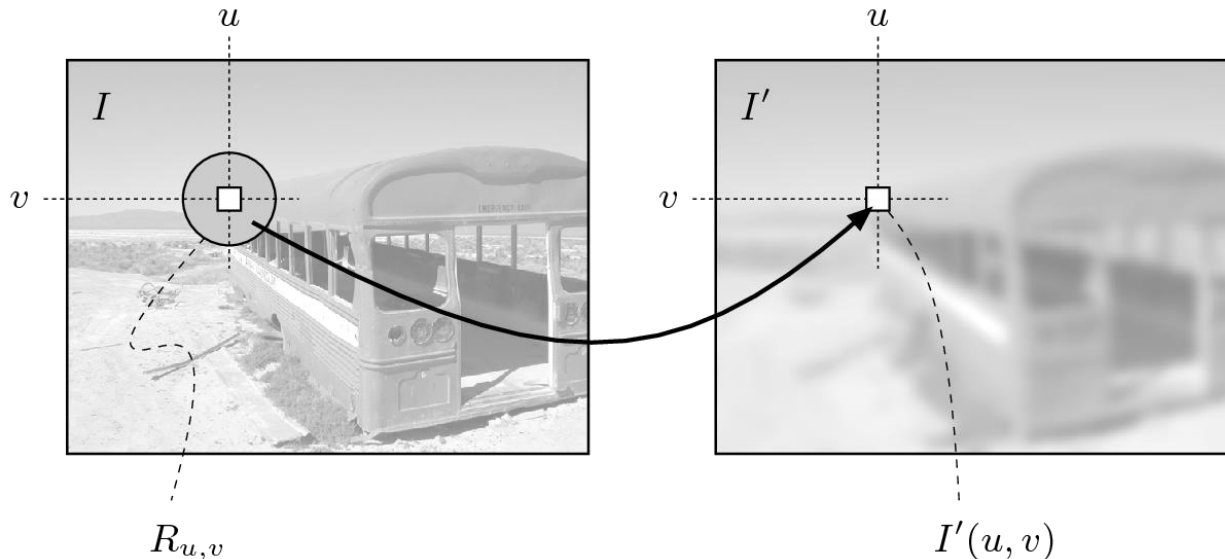


- Combining multiple pixels necessary for certain operations:
 - Blurring, Smoothing
 - Sharpening



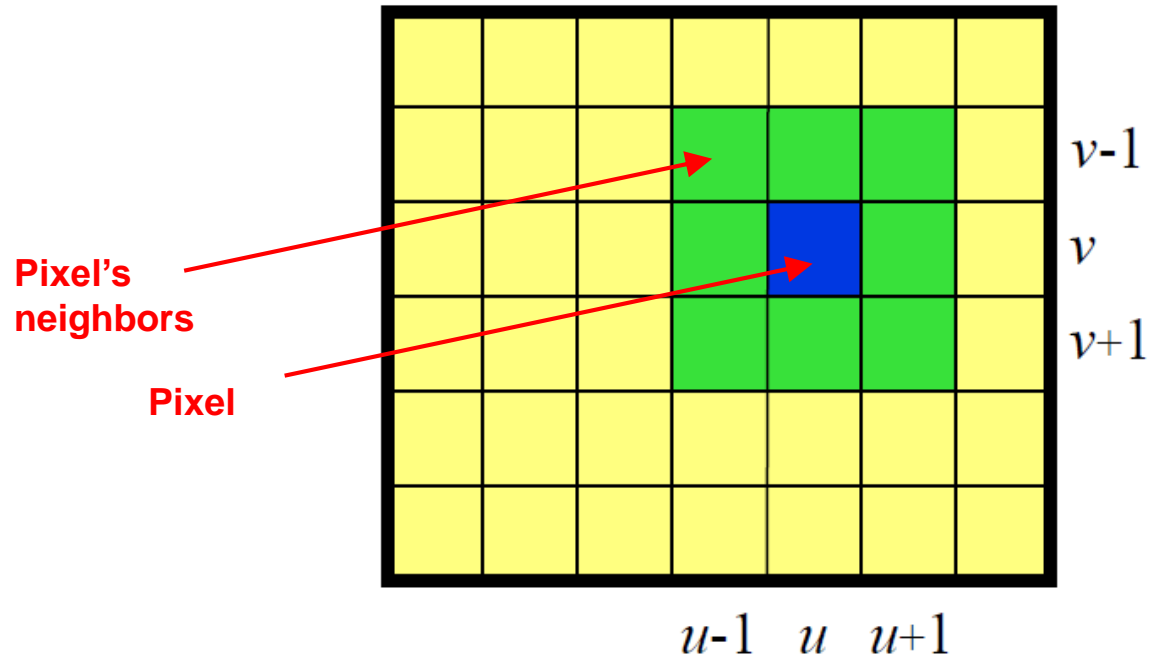
Definition: Spatial Filter

- An image operation that combines each pixel's intensity $I(u, v)$ with that of neighboring pixels
- **E.g:** average/weighted average of group of pixels





Example: Mean of 3x3 Neighborhood



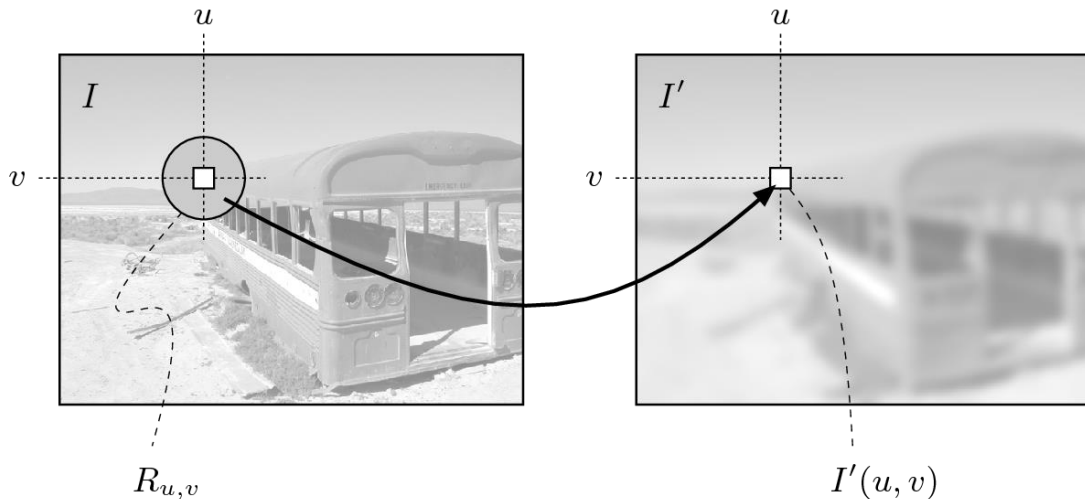
$$I'(u, v) = \frac{1}{9} \sum_{i=-1}^1 \sum_{j=-1}^1 I(u + i, v + j)$$



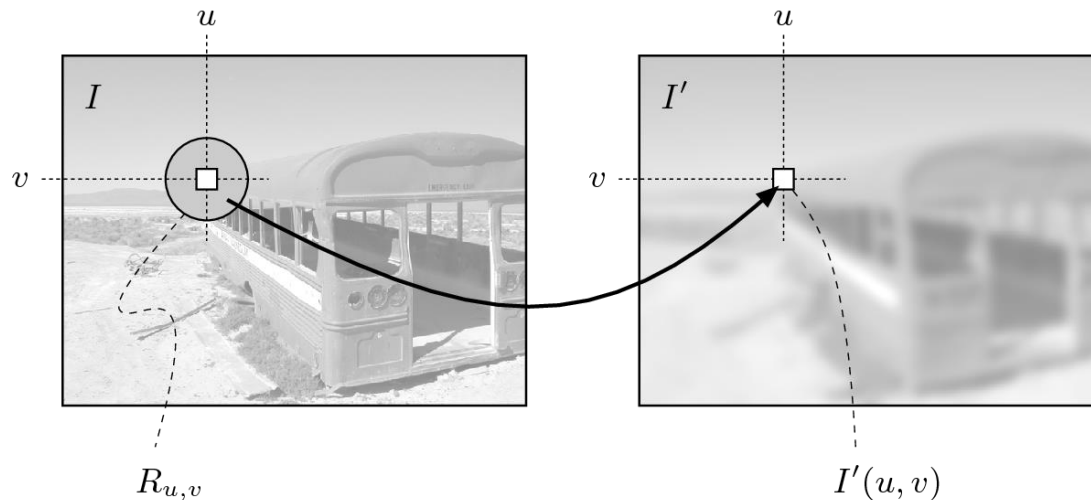
Smoothing an Image by Averaging

- Replace each pixel by average of neighboring pixels
- For 3x3 neighborhood:

$$I'(u, v) \leftarrow \frac{p_0 + p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8}{9}$$



Smoothing an Image by Averaging



- Filter applies a function over small pixel neighborhood
- **Filter size (size of neighborhood):** 3x3, 5x5, 7x7, ..., 21x21, ...
- **Filter shape:** not necessarily square, can be rectangle, circle...
- **Filters function:** can be linear or nonlinear



Mean Filters: Effect of Filter Size



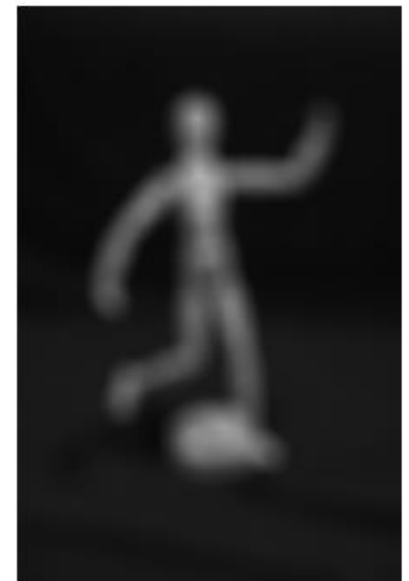
Original



7×7



15×15



41×41

Larger filter = more blurring



Usually make the Weights Integers

$$H(i, j) = \begin{bmatrix} 0.075 & 0.125 & 0.075 \\ 0.125 & \underline{0.200} & 0.125 \\ 0.075 & 0.125 & 0.075 \end{bmatrix} = \frac{1}{40} \begin{bmatrix} 3 & 5 & 3 \\ 5 & \underline{8} & 5 \\ 3 & 5 & 3 \end{bmatrix}$$

Why? Integer math more efficient

Filters



- Filters are usually square matrix and odd. E.g. 3x3 or 5x5
- Example of a 5x5 image blur filter

$$\frac{1}{273} * \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

- Example of 3x3 image blur filter

$$\frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

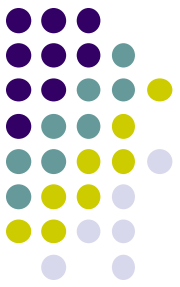


Image Blurring

- Sample images from 3x3 and 5x5 blur filters

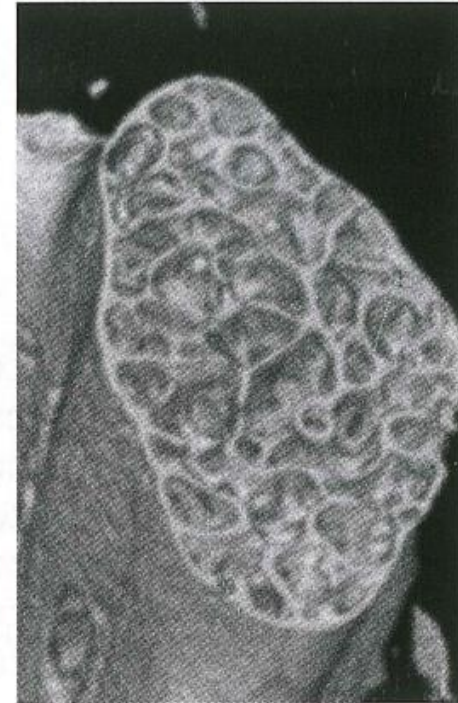
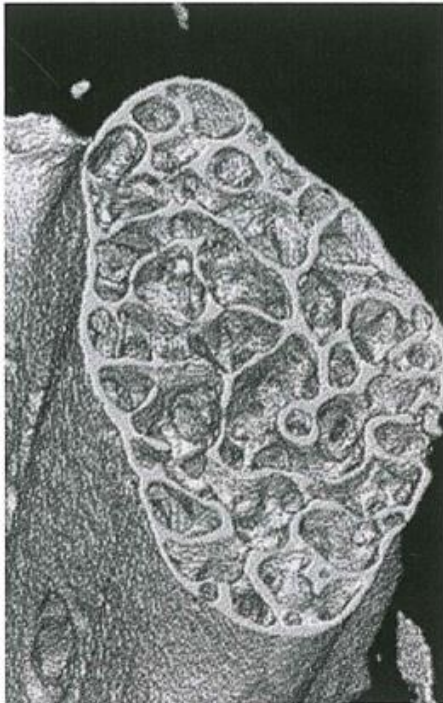




Image Blurring Fragment Shader

- Applying filter

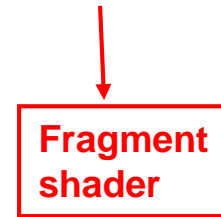
$$\frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

```
Uniform sampler2D uImageUnit;  
in vec2 VST;  
out vec4 fFragColor;
```

```
void main( )  
{
```

```
    ivec2 ires = textureSize( uImageUnit, 0 );  
    float ResS = float( ires.s );
```

VST



fFragColor

Texture

Mipmap level



Image Blurring Fragment Shader (contd)

```
float ResT = float( ires.t );
vec3  irgb  = texture(uImageUnit, VST ).rgb;

          s      t
vec2  stp0 = vec2(1.ResS, 0. ); //texel offsets
vec2  st0p = vec2(0.   , 1./ResT);
vec2  stpp = vec2(1./ResS, 1./ResT);
vec2  stpm = vec2(1./ResS, -1./ResT);
```

$$\frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

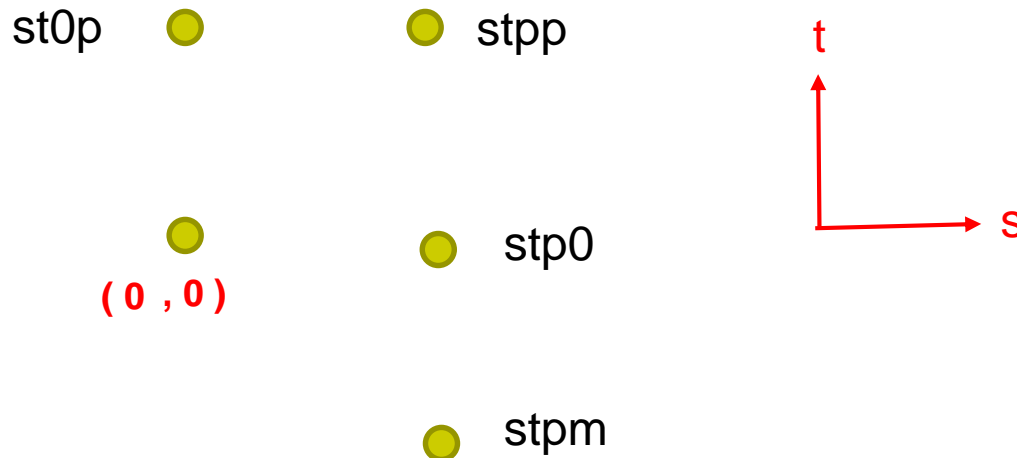




Image Blurring Fragment Shader (contd)

```
// 3x3 pixel colors next
vec3 i00 = texture( uImageUnit, vST ).rgb;
vec3 im1m1 = texture( uImageUnit, vST-stpp ).rgb;
vec3 ip1p1 = texture( uImageUnit, vST+stpp ).rgb;

vec3 im1p1 = texture( uImageUnit, vST-stpm ).rgb;
vec3 ip1m1 = texture( uImageUnit, vST+stpm ).rgb;

vec3 im10 = texture( uImageUnit, vST-stp0 ).rgb;
vec3 ip10 = texture( uImageUnit, vST+stp0 ).rgb;

vec3 i0m1 = texture( uImageUnit, vST-st0p ).rgb;
vec3 i0p1 = texture( uImageUnit, vST+st0p ).rgb;
```

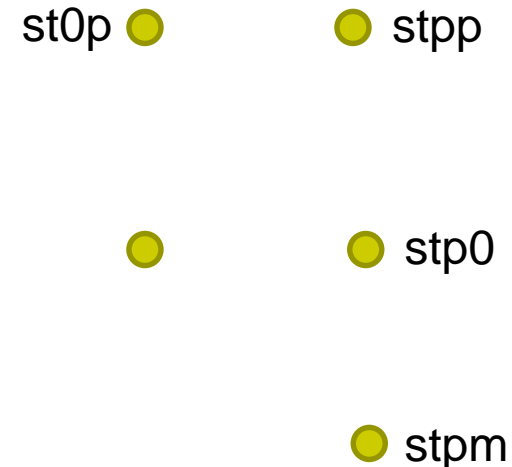
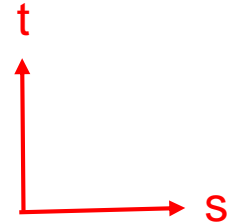




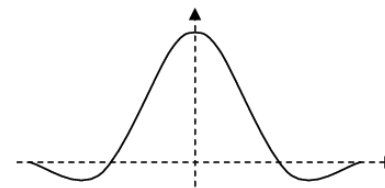
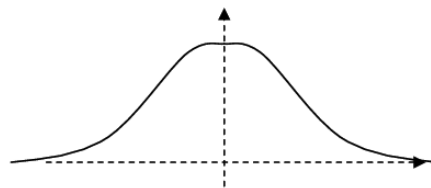
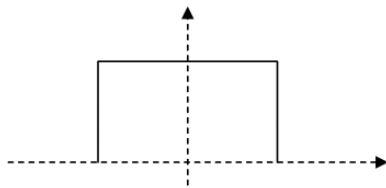
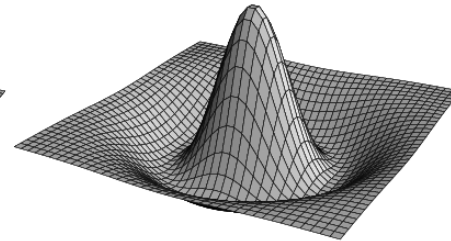
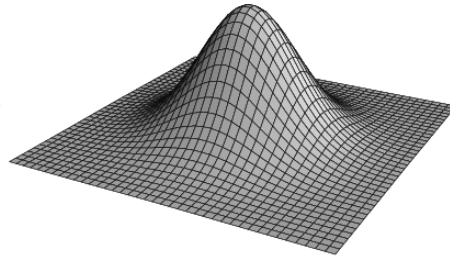
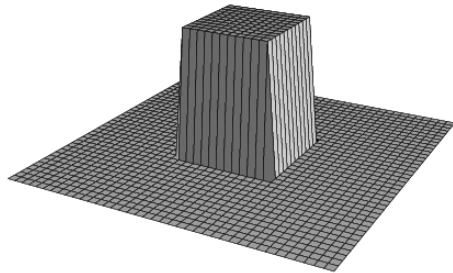
Image Blurring Fragment Shader (contd)

```
vec3 target = vec3(0., 0., 0.);  
target += 1.*(im1m1+ip1m1+ip1p1+im1p1); // apply blur  
target += 2.*(im10+ip10+i0m1+i0p1);  
target += 4.*(i00);  
  
target /= 16.;  
  
fFragColor = vec4( target, 1. );
```

$$\frac{1}{16} * \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Apply weights

Types of Linear Filters



0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

(a)

Box

0	1	2	1	0
1	3	5	3	1
2	5	9	5	2
1	3	5	3	1
0	1	2	1	0

(b)

Gaussian

0	0	-1	0	0
0	-1	-2	-1	0
-1	-2	16	-2	-1
0	-1	-2	-1	0
0	0	-1	0	0

(c)

Laplace



Edge Detection

- Uses 2 filters: 1 vertical and 1 horizontal
- Vertical is actually horizontal rotated 90 degrees

Horizontal
Filter

$$H = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

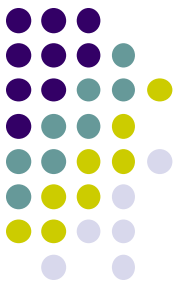
$$V = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Vertical
Filter

$$S = \sqrt{H^2 + V^2}$$

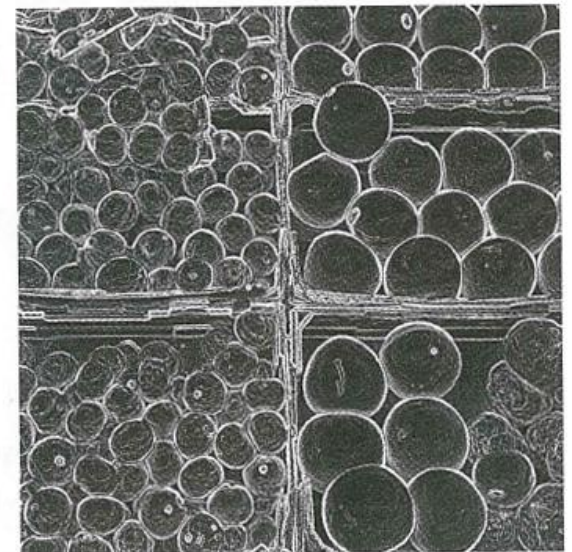
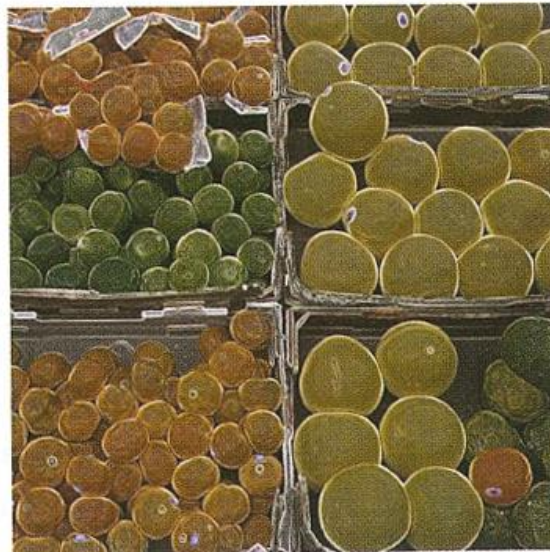
$$\Theta = \text{atan2}(V, H)$$

For an edge
S will be large



Edge Detection

- Algorithm:
 - Compare 2 columns (or rows)
 - If difference is “large”, this is an edge
 - If difference is “small”, not an edge
- Comparison can be done in color or luminance





Edge Detection Fragment Shader

```
const vec3 LUMCOEFFS = vec3( 0.2125,0.7154,0.0721 );
. . .
vec2 stp0 = vec2(1./ResS, 0. );
vec2 st0p = vec2(0.          , 1./ResT);
vec2 stpp = vec2(1./ResS, 1./ResT);
vec2 stpm = vec2(1./ResS, -1./ResT);
float i00 = dot( texture2D( uImageUnit, vST ).rgb          , LUMCOEFFS );
float im1m1 = dot( texture2D( uImageUnit, vST-stpp ).rgb, LUMCOEFFS );
float ip1p1 = dot( texture2D( uImageUnit, vST+stpp ).rgb, LUMCOEFFS );
float im1p1 = dot( texture2D( uImageUnit, vST-stpm ).rgb, LUMCOEFFS );
float ip1m1 = dot( texture2D( uImageUnit, vST+stpm ).rgb, LUMCOEFFS );
float im10 = dot( texture2D( uImageUnit, vST-stp0 ).rgb, LUMCOEFFS );
float ip10 = dot( texture2D( uImageUnit, vST+stp0 ).rgb, LUMCOEFFS );
float i0m1 = dot( texture2D( uImageUnit, vST-st0p ).rgb, LUMCOEFFS );
float i0p1 = dot( texture2D( uImageUnit, vST+st0p ).rgb, LUMCOEFFS );
float h = -1.*im1p1 - 2.*i0p1 - 1.*ip1p1 + 1.*im1m1 + 2.*i0m1 + 1.*ip1m1;
float v = -1.*im1m1 - 2.*im10 - 1.*im1p1 + 1.*ip1m1 + 2.*ip10 + 1.*ip1p1;

float mag = sqrt( h*h + v*v );
vec3 target = vec3( mag,mag,mag );
color = vec4( mix( irgb, target, T ), 1. );
```

Multiply by luminance coeffs
to convert colors to gray

Compute horizontal
and vertical filters

(1 - T).irgb + T.target

For an edge, target will be large, color will be washed out (> 1 or white)

$$H = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad V = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$



Embossing

- Embossing is similar to edge detection
- Depending on edge angle (how sharp)
 - Replace color by luminance
 - Highlight images differently depending on edge angles (magnitude of difference)



Embossing



color cp1p1



color, c00



stp0

```
vec2 stp0 = vec2( 1./ResS, 0. );  
vec2 stpp = vec2( 1./ResS, 1./ResT);  
vec3 c00 = texture2D( ulmageUnit, vST ).rgb;  
vec3 cp1p1 = texture2D( ulmageUnit, vST + stpp ).rgb;
```

```
vec3 diffs = c00 - cp1p1;  
float max = diffs.r;  
if( abs(diffs.g) > abs(max) )  
    max = diffs.g;  
if( abs(diffs.b) > abs(max) )  
    max = diffs.b;
```

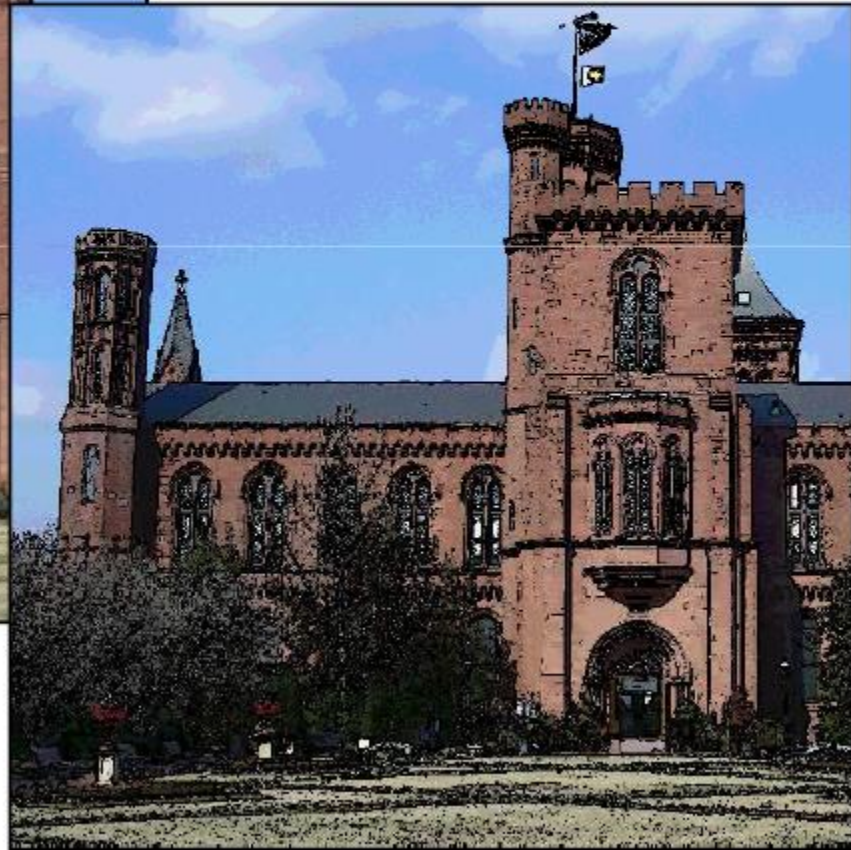
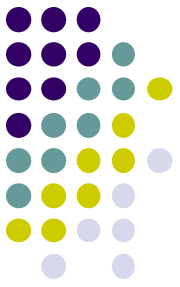
Find largest
difference, r, g or b

Convert largest
difference to gray

```
float gray = clamp( max + .5, 0., 1. );  
vec4 grayVersion = vec4( gray, gray, gray, 1. );  
vec4 colorVersion = vec4( gray*c00, 1. );  
fFragColor = mix( grayVersion, colorVersion, T );
```



Toon Rendering for Non-Photorealistic Effects





Toon Shader

- Implement Toon shader based using Sobel filter
- Algorithm
 - Calculate luminance of each pixel (brightness)
 - Apply Sobel edge detection filter and get a magnitude
 - If magnitude $>$ threshold, color pixel black
 - Else, quantize pixel's color
 - Output the colored pixel



Toon Fragment Shader (Some Code)



```
... insert code for Sobel Filter
```

```
// Calculate magnitude, then draw edges or quantize  
float mag = length( vec2(h, v) ); // how much change?
```

```
if( mag > uMagTo ) // if too much, use black
```

```
    fFragColor = vec4( 0., 0., 0., 1.);
```

```
else{ // else quantize the color
```

```
    rgb.rgb *= uQuantize; // multiply by number of quanta
```

```
    rgb.rgb += vec3( .5, .5, .5); // round
```

```
    ivec3 intrgb = ivec3( rgb.rgb ); // truncate
```

```
    rgb.rgb = vec3( intrgb ) / Quantize; // calc. quantized color
```

```
    fFragColor = vec4( rgb, 1.);
```

```
}
```

Toon Rendering



Original
Image



Colors
Quantized



Outlines Added





Image Flipping, Rotation and Warping

- We can transform image (flip, rotate, warp)
- Basic idea: Look up a **transformed pixel address** instead of the current one
- To flip an image upside down:
 - At pixel location st , look up the color at location $s(1-t)$
 - Fragment shader code:

```
vec2 st = vST;  
st.t = 1 - st.t;  
vec3 irgb = texture( uImageUnit, st ).rgb;  
fFragColor = vec4( irgb, 1);
```

Note: For horizontal flip, look up $(1-s)t$ instead of st !!



Image Flipping, Rotation and Warping

- Rotating an image 90 degrees counterclockwise:
 - Look up $(t, 1 - s)$ instead of $s t$
- **Image warping:** we can use a function to select which pixel somewhere else in the image to look up
- For example: apply function on both texel coordinates (s, t)

$$x = x + t * \sin(\pi * x)$$

Image Flipping, Rotation and Warping



$$x = x + t * \sin(\pi * x)$$

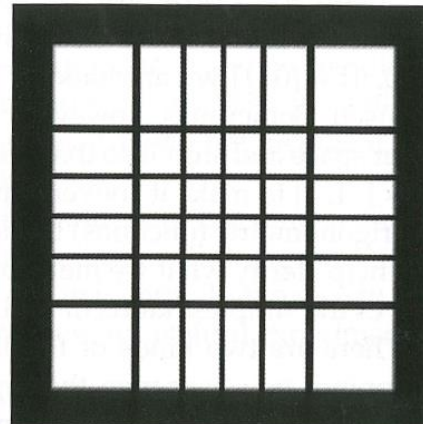
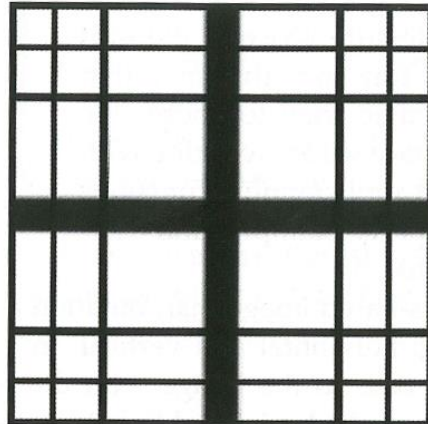




Image Flipping, Rotation and Warping

- Fragment shader code to implement

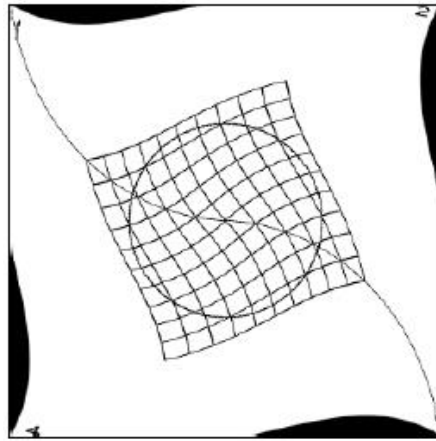
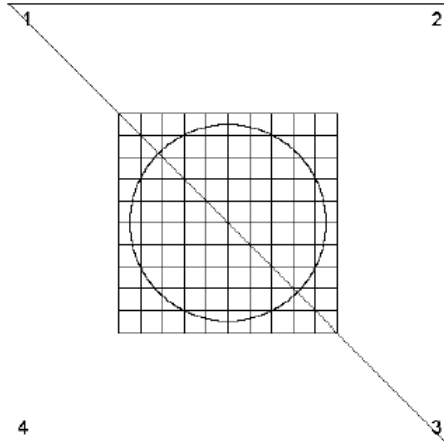
$$x = x + t * \sin(\pi * x)$$

```
const float PI = 3.14159265
uniform sampler2D uImageUnit;
uniform float uT;
in vec2 vST;    out vec4 fFragColor;

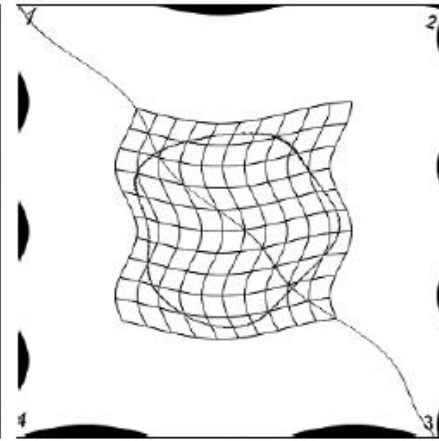
void main( ){
    vec2 st = vST;
    vec2 xy = st;
    xy = 2. * xy - 1;    // map to [-1,1] square
    xy += uT * sin(PI*xy);
    st = (xy + 1.)/2.;    // map back to [0,1] square
    vec3 = irgb = texture(uImageUnit, st ).rgb; // use transformed st

    fFragColor = vec4( irgb, 1.);    }
```

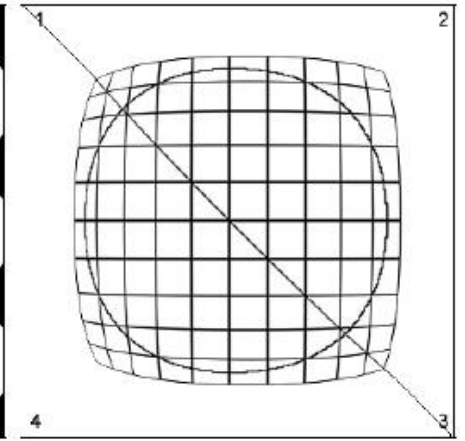
Non-Linear Image Warps



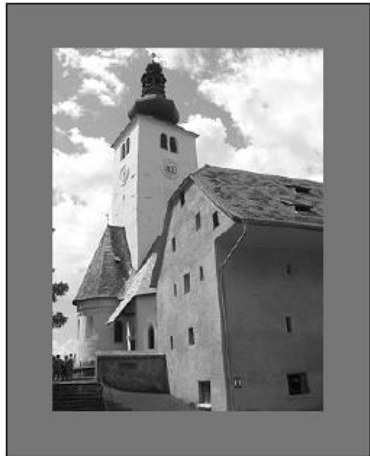
(a)



(b)



(c)



Original



(d)

Twirl



(e)

Ripple



(f)

Spherical

Twirl

- **Notation:** Instead using texture colors at (x',y') , use texture colors at twirled (x,y) location
- Twirl?
 - Rotate image by angle α at center or anchor point (x_c,y_c)
 - Increasingly rotate image as radial distance r from center increases (up to r_{max})
 - Image unchanged outside radial distance r_{max}

$$T_x^{-1} : x = \begin{cases} x_c + r \cdot \cos(\beta) & \text{for } r \leq r_{max} \\ x' & \text{for } r > r_{max}, \end{cases}$$

$$T_y^{-1} : y = \begin{cases} y_c + r \cdot \sin(\beta) & \text{for } r \leq r_{max} \\ y' & \text{for } r > r_{max}, \end{cases}$$

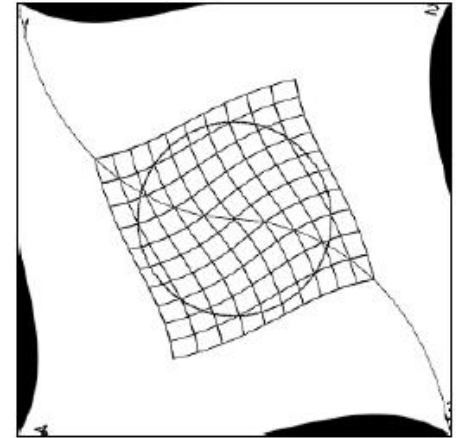
with

$$d_x = x' - x_c,$$

$$d_y = y' - y_c,$$

$$r = \sqrt{d_x^2 + d_y^2},$$

$$\beta = \text{Arctan}(d_y, d_x) + \alpha \cdot \left(\frac{r_{max} - r}{r_{max}} \right).$$



(a)



(d)

Twirl Fragment Shader Code



```
const float PI = 3.14159265
uniform sampler2D uImageUnit;
uniform float uD, uR;

in vec3 vST;
out vec4 fFracColor;

void main( ){
    ivec2 ires = textureSize( uImageUnit, 0);
    float Res = float( ires.s ); // assume it's a square texture image

    vec2 st = vST;
    float Radius = Res * uR;
    vec2 xy = Res * st; // pixel coordinates from texture coords

    vec2 dxy = xy - Res/2.; // twirl center is (Res/2, Res/2)
    float r = length( dxy );
    float beta = atan( dxy.y, dxy.x) + radians(uD) * (Radius - r)/Radius;
```

Twirl Fragment Shader Code (Contd)



```
vec2 xy1 = xy;
if(r <= Radius)
{
    xy1 = Res/2. + r * vec2( cos(beta), sin(beta) );
}
st = xy1/Res; // restore coordinates

vec3 irgb = texture( uImageUnit, st ).rgb;
fFragColor = vec4( irgb, 1. );
}
```

Ripple

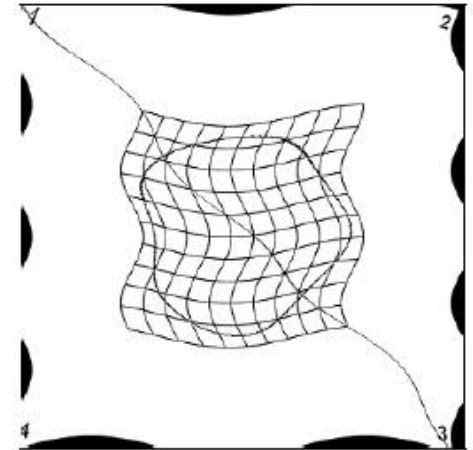
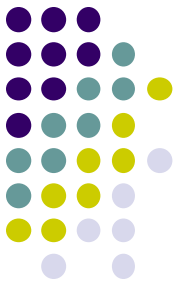
- Ripple causes wavelike displacement of image along both the x and y directions

$$T_x^{-1} : x = x' + a_x \cdot \sin\left(\frac{2\pi \cdot y'}{\tau_x}\right),$$

$$T_y^{-1} : y = y' + a_y \cdot \sin\left(\frac{2\pi \cdot x'}{\tau_y}\right).$$

- Sample values for parameters (in pixels) are

- $\tau_x = 120$
- $\tau_y = 250$
- $a_x = 10$
- $a_y = 15$



(b)



(e)

Spherical Transformation

- Imitates viewing image through a lens placed over image
- Lens parameters: center (x_c, y_c) , lens radius r_{max} and refraction index ρ
- Sample values $\rho = 1.8$ and $r_{max} = \text{half image width}$

$$T_x^{-1} : x = x' - \begin{cases} z \cdot \tan(\beta_x) & \text{for } r \leq r_{max} \\ 0 & \text{for } r > r_{max}, \end{cases}$$

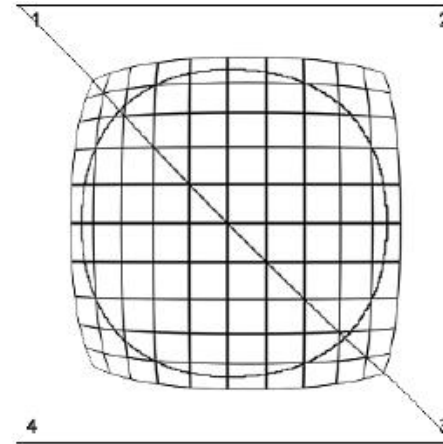
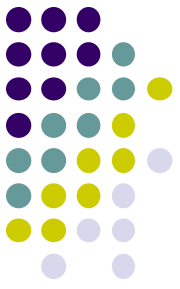
$$T_y^{-1} : y = y' - \begin{cases} z \cdot \tan(\beta_y) & \text{for } r \leq r_{max} \\ 0 & \text{for } r > r_{max}, \end{cases}$$

$$d_x = x' - x_c, \quad r = \sqrt{d_x^2 + d_y^2},$$

$$d_y = y' - y_c, \quad z = \sqrt{r_{max}^2 - r^2},$$

$$\beta_x = \left(1 - \frac{1}{\rho}\right) \cdot \sin^{-1}\left(\frac{d_x}{\sqrt{(d_x^2 + z^2)}}\right),$$

$$\beta_y = \left(1 - \frac{1}{\rho}\right) \cdot \sin^{-1}\left(\frac{d_y}{\sqrt{(d_y^2 + z^2)}}\right).$$



(c)



(f)



Image Warping

```
uniform float      uS0, uT0;
uniform float      uPower;
uniform sampler2D  uTexUnit;
in vec2            vST;
out vec4           fFragColor;

void
main( )
{
    vec2 delta = vST - vec2(uS0,uT0);
    st = vec2(uS0,uT0) + sign(delta) * pow( abs(delta), uPower );
    vec3 rgb = texture2D( uTexUnit, vST ).rgb;
    fFragColor = vec4( rgb, 1. );
}
```



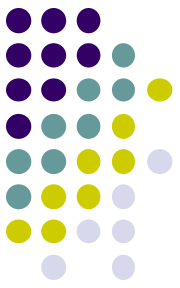


Image Morphing

- Mark similar points on the images (e.g. nose)
- Distort nose position + fade image 1 into image 2





Motion Blur

- Texture element may be combined with neighboring texture elements to create motion blur

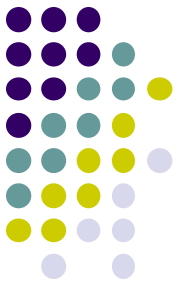


With motion blur



Without motion blur

Color Correction



Original



After Levels Adjustment



Original



After Levels Adjustment



Color Correction

- Color correction uses a function to convert colors in an image to some other color
- Why color correct?
 - Mimic appearance of a type of film
 - Portray a particular mood
 - Convert from one color space to another (e.g. RGB to CIE)
 - Example of conversion from RGB to CIE's XYZ color space

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Color Correction



Original Shot



Day-for-Night Color Corrected shot



References

- Mike Bailey and Steve Cunningham, Graphics Shaders (second edition)
- Wilhelm Burger and Mark Burge, Digital Image Processing: An Algorithmic Introduction using Java, Springer Verlag Publishers
- OpenGL 4.0 Shading Language Cookbook, David Wolff
- Real Time Rendering (3rd edition), Akenine-Moller, Haines and Hoffman
- Suman Nadella, CS 563 slides, Spring 2005