Computer Graphics (CS 543)
Lecture 11 (Part 1): Image Manipulation

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

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

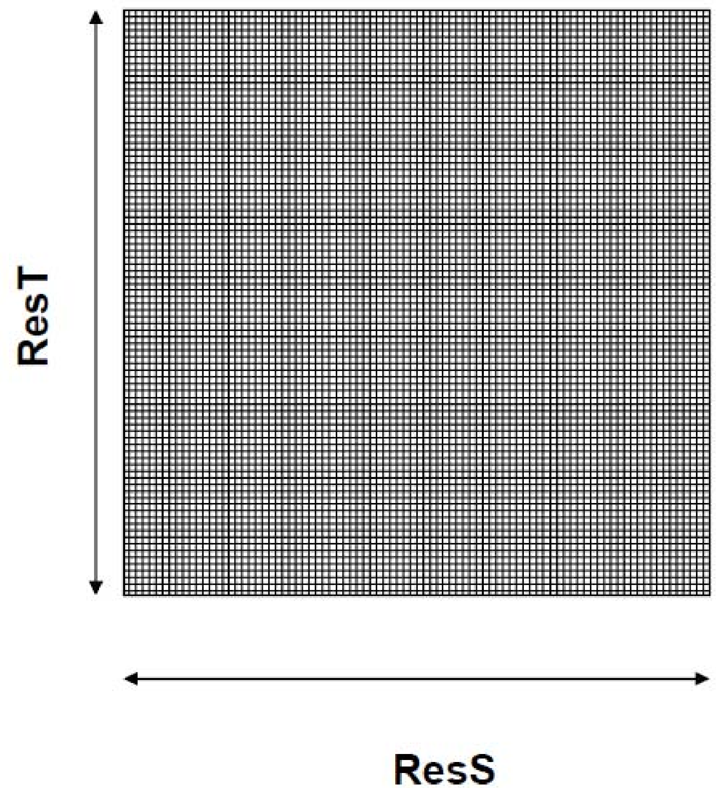
- Image processing the output of graphics rendering is called **post-processing**
- To post-process using GPU, rendered output usually written to offscreen buffer (e.g. color image, z-depth buffer, etc)
- Image in offscreen buffer treated as texture, mapped to screen-filling quadrilateral
- Fragment 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 );
```

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 it luminance by multiplying by a set of weights (0.2125, 0.7154, 0.0721). i.e.

Luminance = **R** * 0.2125 + **G** * 0.7154 + **B** * 0.0721

- Note that the weights sum to 1

Code (Fragment Shader) for Luminance

```
const vec3 W = vec3(0.2125, 0.7154, 0.0721);
vec3 irgb = texture( uImageUnit, vST).rgb;
float luminance = dot(irgb, W);

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

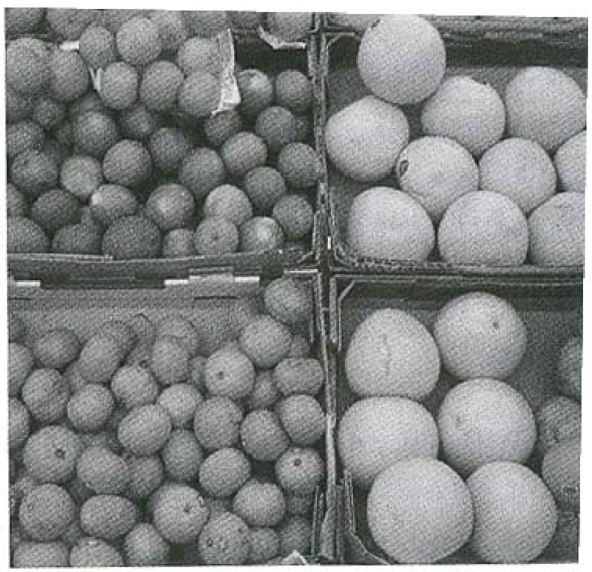


Image Negative

- 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 and values

Original Image

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

Sobel Filter applied

Sobel Filter

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

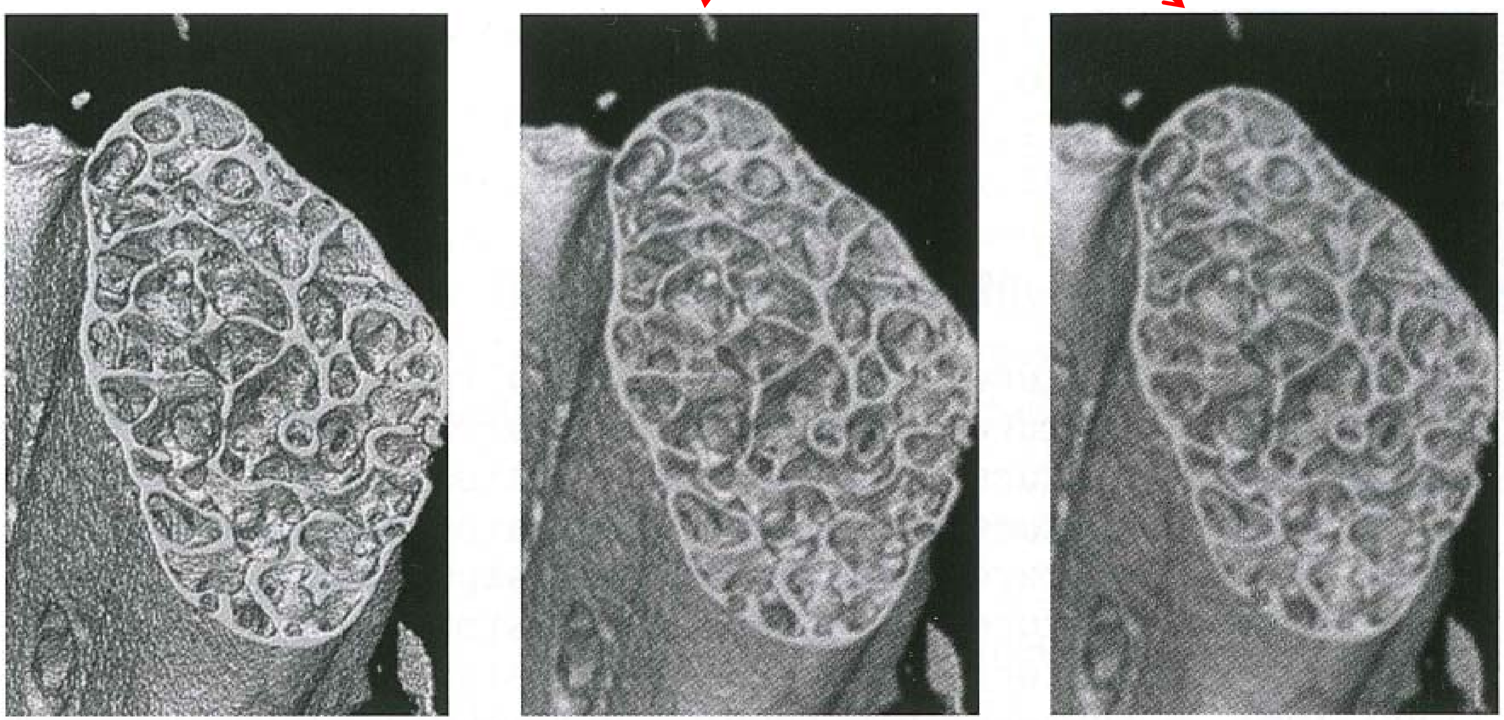


Image Blurring Fragment Shader

- 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 );
```

Image Blurring Fragment Shader (contd)

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

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);
```

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

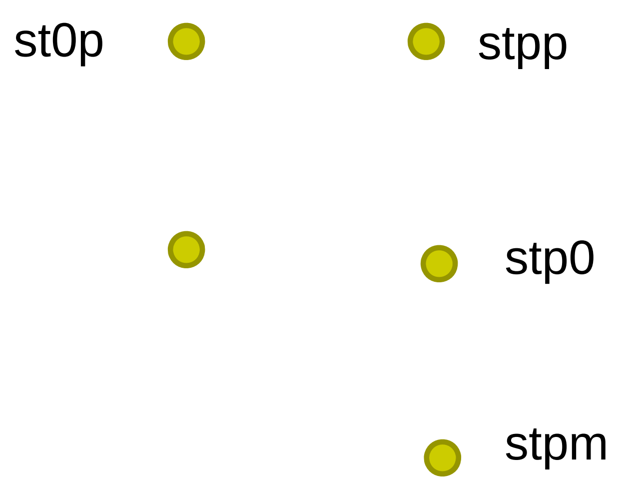


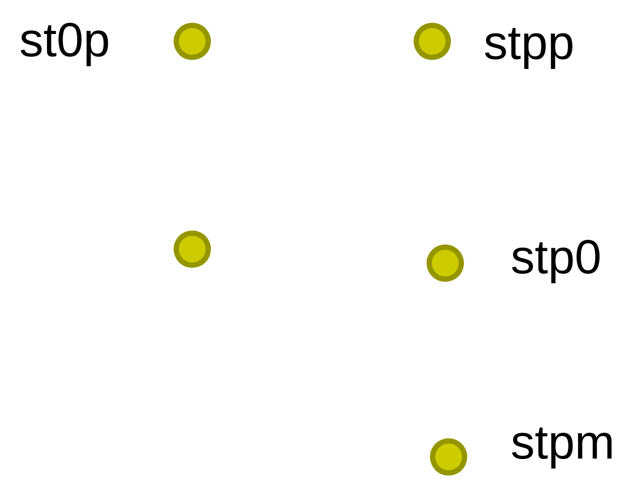
Image Blurring Fragment Shader (contd)

```
// 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;
```



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

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. );
```

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

Edge Detection

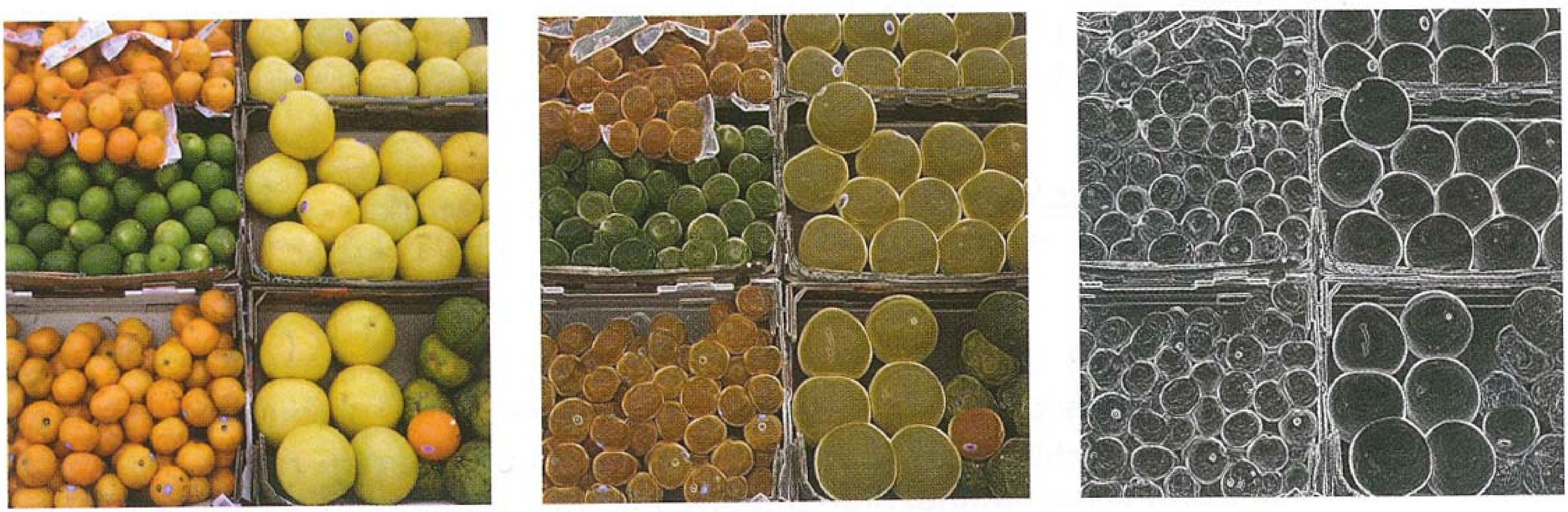
- Uses 2 filters: 1 vertical and 1 horizontal
- Vertical is actually horizontal rotated 90 degrees

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

$$S = \sqrt{H^2 + V^2} \qquad \Theta = \mathrm{atan2}(\mathrm{V}, \mathrm{H})$$

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. );
```

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

Embossing

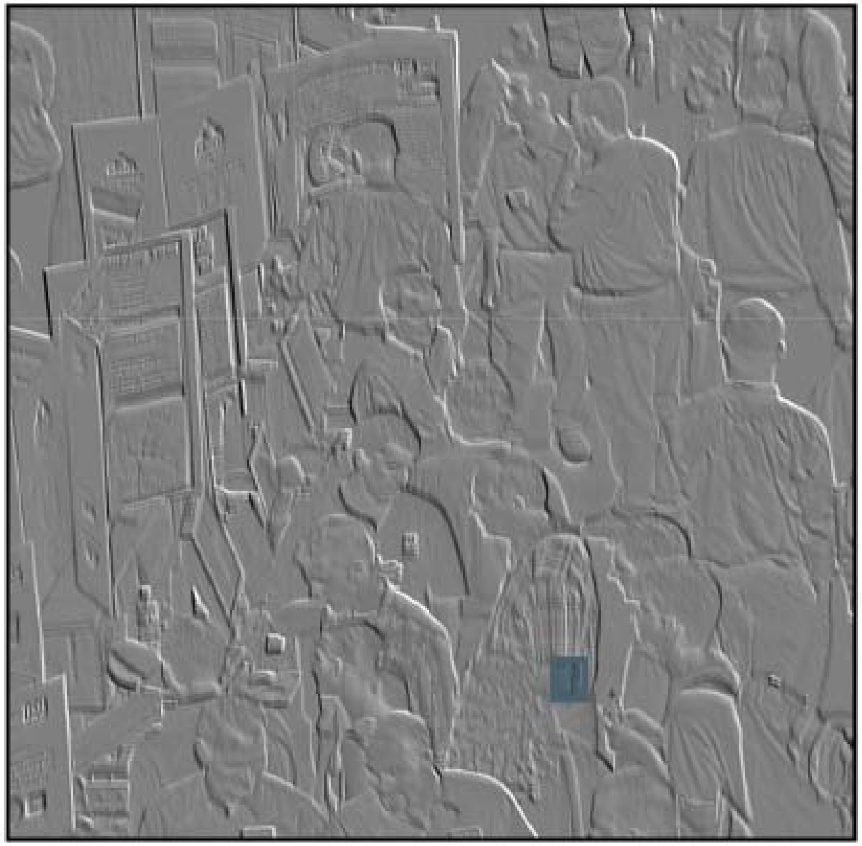
- Embossing is similar to edge detection
- Depending on edge angle (how sharp)
  - Replace color by luminance
  - Highlight images differently depending on edge angles

Embossing

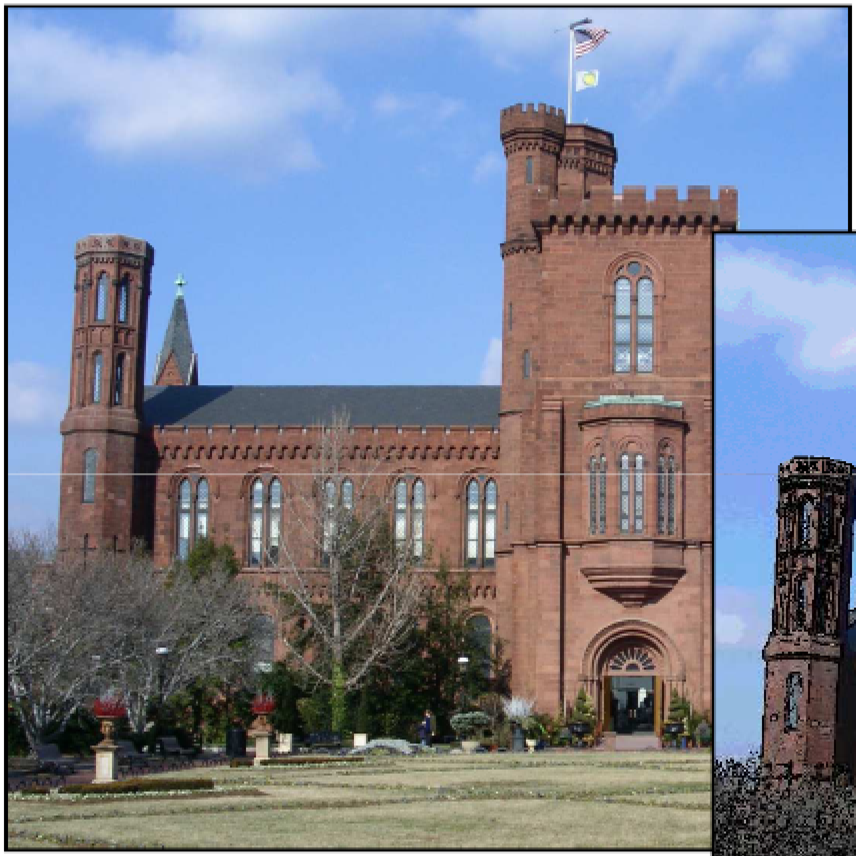
```
vec2 stp0 = vec2( 1./ResS,  0. );
vec2 stpp = vec2( 1./ResS,  1./ResT);
vec3 c00    = texture2D(    uImageUnit, vST ).rgb;
vec3 cp1p1  = texture2D( uImageUnit, vST + stpp ).rgb;

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;

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
  - 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

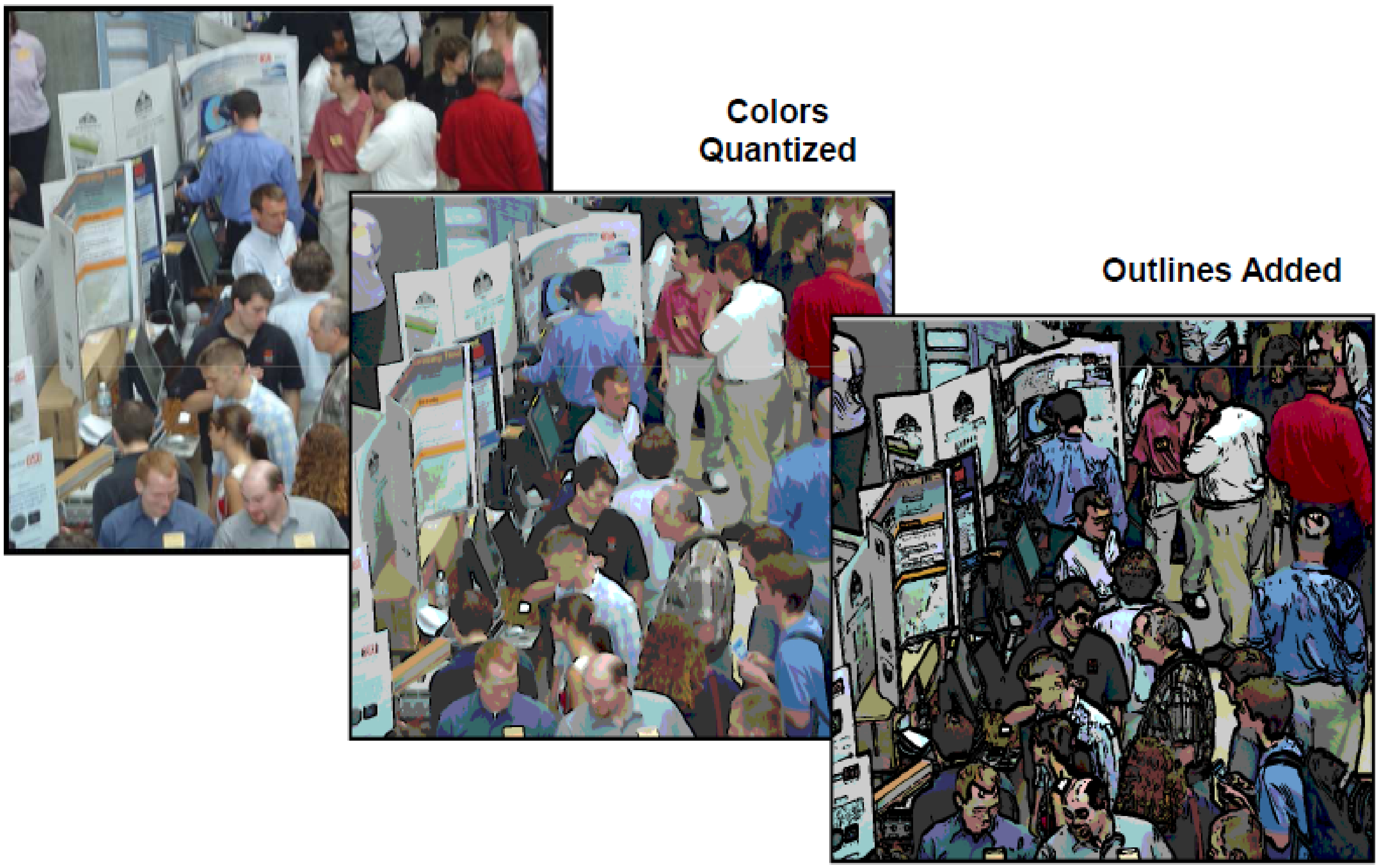


Image Flipping, Rotation and Warping

- 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 *s t* !!

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

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

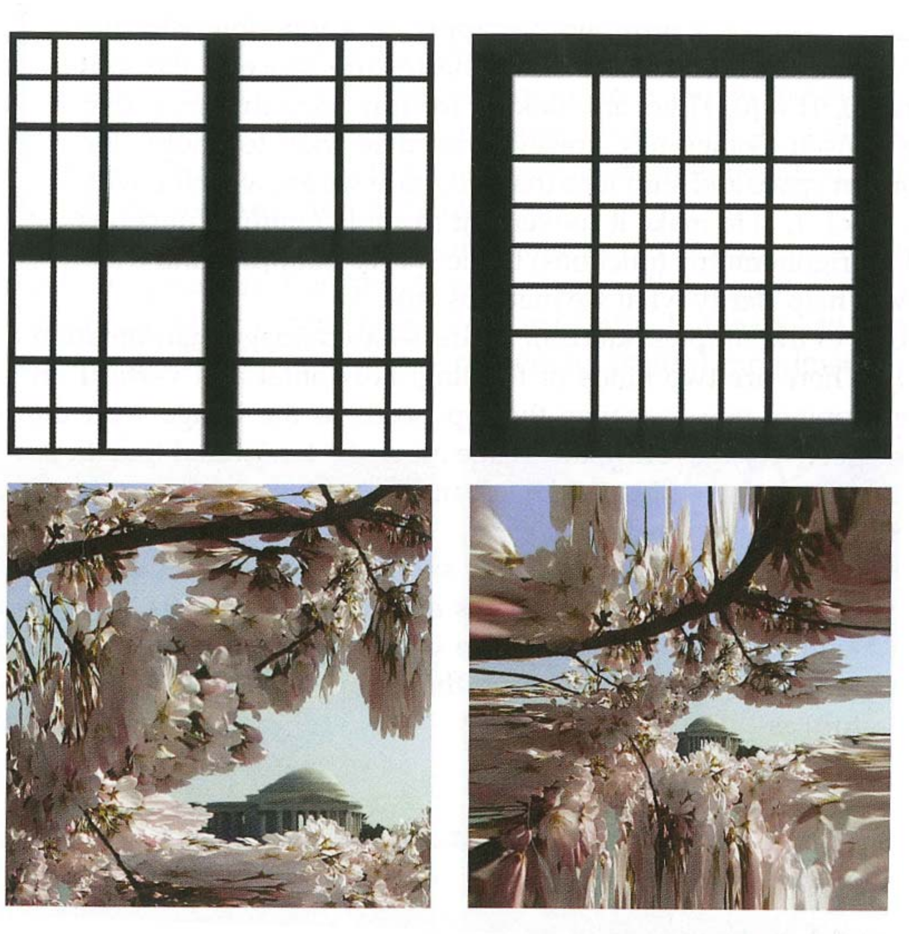


Image Flipping, Rotation and Warping

- Fragment shader code to implement

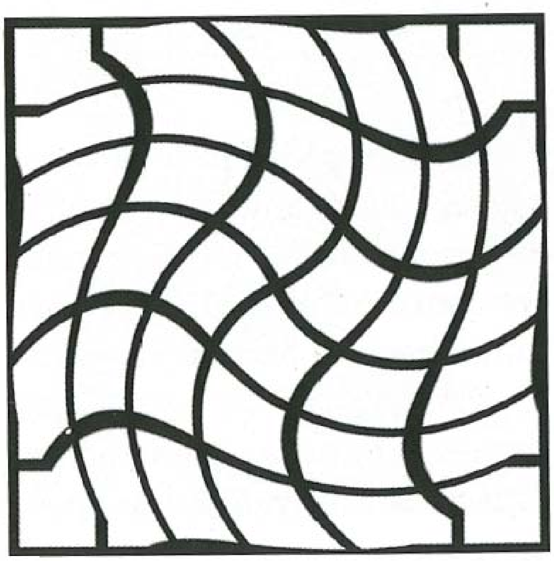
$$x = x + t * \sin(\pi * x)$$

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

Image Warping: Twirl Transformation

- Twirl transformation rotates image by angle α around anchor point (x_c, y_c)
- Rotation angle:
  - Varies from α at the center,
  - Increases linearly away from the center as it approaches limiting radius r_{max}



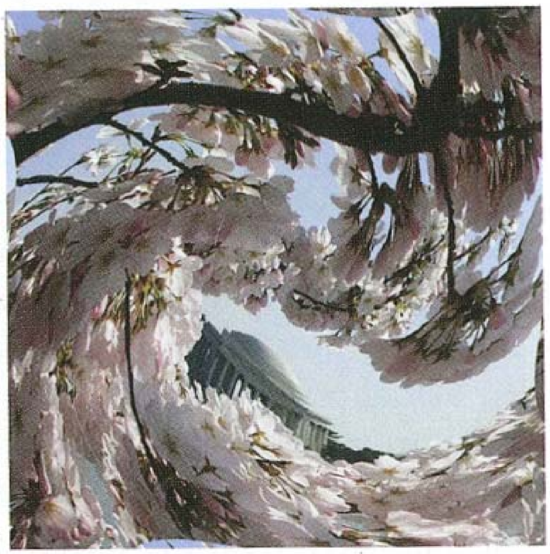
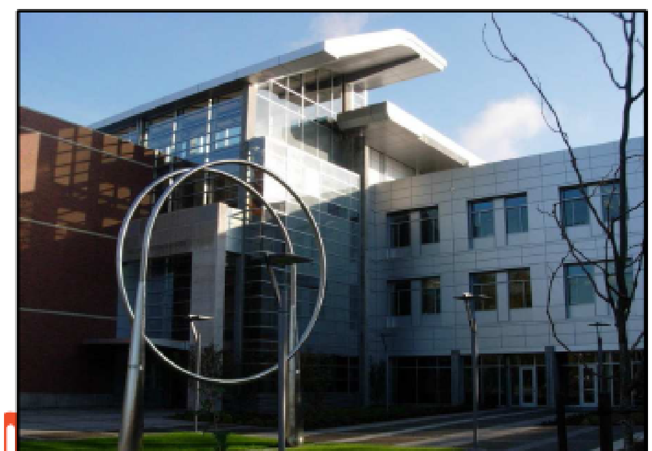


Image Warping

```
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. );
}
```



Motion Blur

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

With motion blur

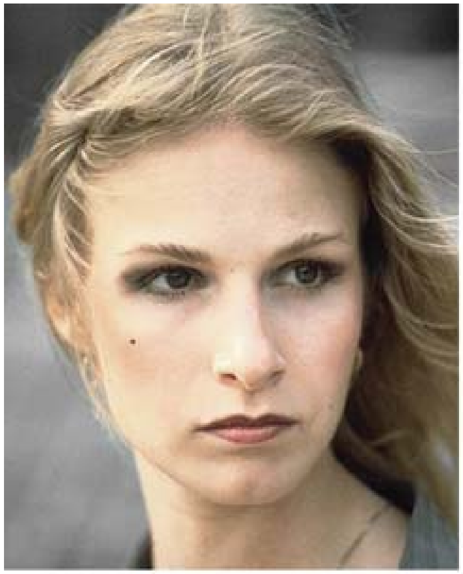
Without motion blur

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
  - 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

After Levels Adjustment

Original

After Levels Adjustment

Color Correction


Original Shot

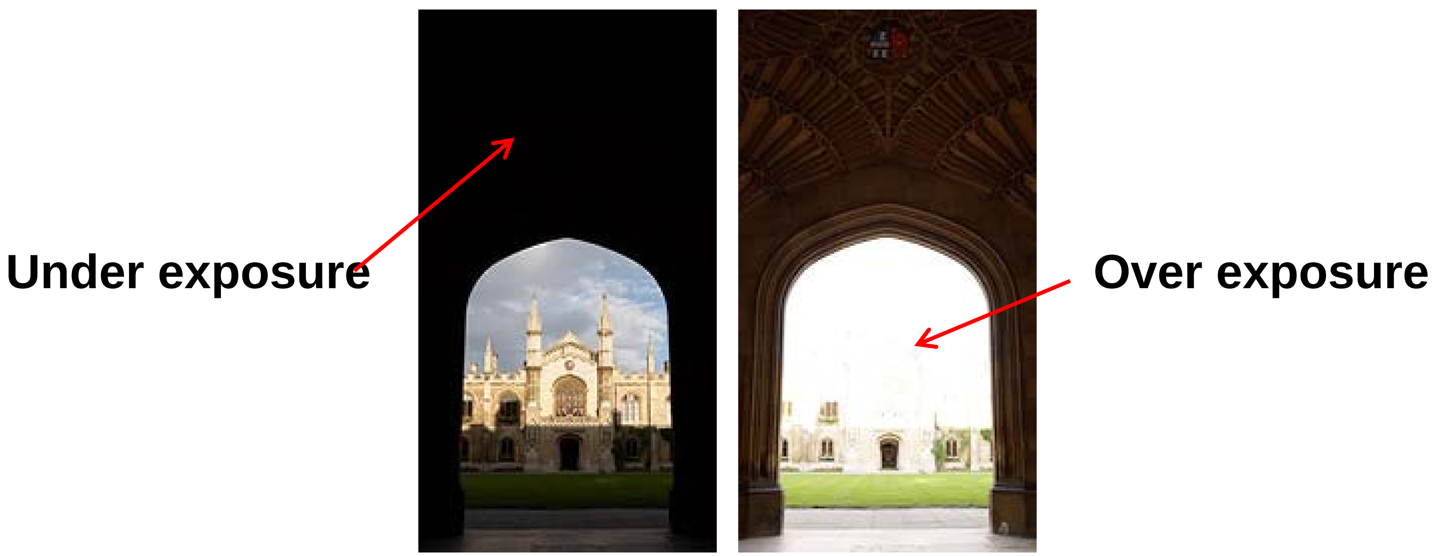

Day-for-Night Color Corrected shot

High Dynamic Range

- Sun’s brightness is about 60,000 lumens
- Dark areas of earth has brightness of 0 lumens
- Basically, world around us has range of 0 – 60,000 lumens **(High Dynamic Range)**
- However, monitor has ranges of colors between 0 – 255 **(Low Dynamic Range)**
- New file formats have been created for HDR images (wider ranges). (E.g. OpenEXR file format)

High Dynamic Range

- Some scenes contain **very bright** + **very dark** areas
- Using uniform scaling factor to map actual intensity to displayed pixel intensity means:
  - Either some areas are unexposed, or
  - Some areas of picture are overexposed



Tone Mapping

- Technique for scaling intensities in real world images (e.g HDR images) to fit in displayable range
- Try to capture feeling of real scene: **non-trivial**
- **Example:** If coming out of dark tunnel, lights should seem bright
- **General idea:** apply different scaling factors to diffferent parts of the image

Tone Mapping

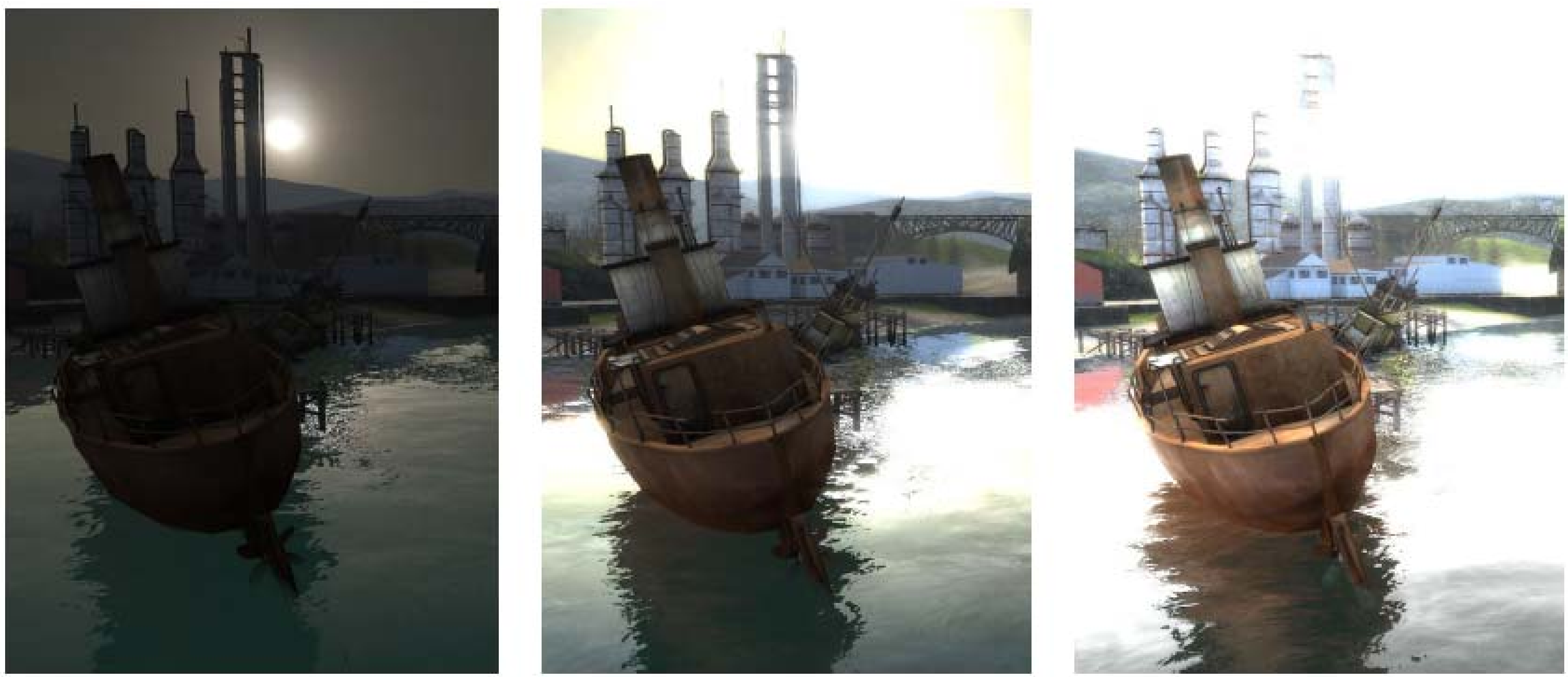


***Figure 10**. Scene from Lost Coast at Varying Exposure Levels*

Depth of Field

- We can simulate a real camera
- In photographs, a range of pixels in focus
- Pixels outside this range are out of focus
- This effect is known as **Depth of field**

Motion Blur

- Motion blur caused by exposing film to moving objects
- Motion blur: Blurring of samples taken over time (temporal)
- Makes fast moving scenes appear less jerky
- 30 fps + motion blur better than 60 fps + no motion blur

Motion Blur

- Basic idea is to average series of images over time
- Move object to set of positions occupied in a frame, blend resulting images together
- Can blur moving average of frames. E.g blur 8 images
- **Velocity buffer:** blur in screen space using velocity of objects

References

- Mike Bailey and Steve Cunningham, Graphics Shaders (second edition)
- 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