Overview

• Image Processing
  • GPU how?
• Blending
  • Porter-Duff and beyond
• Painting
Image Processing – Overview

• **Image Processing is the creation of a new image by processing the pixels of an existing image; each pixel in the output image is computed as a function of one or several pixels in the original image.**

• A generic way to do image processing on GPUs

• Color manipulation

• Convolution Filters
  • Separable Convolution Filters
  • Performance tricks
Getting Started

- 3D APIs (OpenGL, Direct3D) don’t allow reading or writing of pixel values directly
- GPUs can only draw triangles 😞
- Drawing is Processing Trick:
  - Modern GPUs can execute fragment program for every pixel drawn on a screen.
  - Use input-image as texture.
  - Draw screen aligned quad the size of the original image.
  - GPU executes fragment program for every pixel in the output image.
Drawing is Processing

Texture

Draw quad \((w,h)\)

GPU

// Edge detect

varying vec2 vUV;
uniform vec2 vDeltaU;
uniform vec2 vDeltaV;
uniform sampler2D oImage;
void main()
{
vec3 vEdgeU = texture2D(oImage, vUV+vDeltaU) - texture2D(oImage, vUV-vDeltaU);
vec3 vEdgeV = texture2D(oImage, vUV+vDeltaV) - texture2D(oImage, vUV-vDeltaV);
gl_FragColor = abs(vEdgeU)+abs(vEdgeV);
}
Fragment Programs

- Prior to programmable GPUs image processing possible but tedious
- Problems:
  - Only small number of algorithms possible
  - Each algorithm needs completely different render setup
- Solution:
  - Fragment programs
  - Increasingly flexible
  - Same approach for most processing tasks
Practical Issues I

- Traditionally texture-sizes powers of two
- Solution:
  - Use the next bigger power-of-two sized texture and store image in lower left area.
  - Use one of the non-power-of-two extensions:
    - ARB_texture_non_power_of_two
    - EXT_texture_rectangle
Non-Power-Of-Two

- **ARB_texture_non_power_of_two**
  - parametric addressing $[0,1] \times [0,1]$
  - all wrap modes (clamp, u/v-wrap, etc.)
  - mipmaps
  - borders

- **EXT_texture_rectangle**
  - addressing $[0,w] \times [0,h]$
  - no mipmapping, border
  - only clamp (no wrapping)
  - Only GL_NEAREST for 16 bit floating point formats
Simple Image Transforms

- Transformations like scale, rotate, skew, etc. are trivial.
- Drawing is processing – GPU is good for drawing.

For best image quality use anisotropic filtering and mipmaps.
Manipulating Color

- Result is function of the input pixel’s color
- Great example for “Drawing is Processing”
- Very well suited for GPU. Huge performance lead over CPU.
- Three examples:
  - De-Gamma
  - Gamma
  - Color response of film stock
Example 1: De-Gamma

- Stored images often gamma corrected (ready for display)
  - e.g. sRGB format (gamma = 2.2)
- Image processing algorithms usually assume linear color space

```glsl
varying vec2 vPixel;
uniform sampler2D oImage;

void main()
{
    vec3 vColor = texture2D(oImage, vPixel);
    gl_FragColor = pow(vColor, vec3(2.2, 2.2, 2.2));
}
```
Example 2: Monitor Gamma Correction

- Monitors expect Gamma Corrected Input
  - In full-screen mode DACs can gamma correct
- Allow for separate gamma per color channel

```glsl
varying vec2 vPixel;
uniform sampler2D oImage;
uniform vec3 vGamma;

void main()
{
    vec3 vColor = texture2D(oImage, vPixel);
    gl_FragColor = pow(vColor, vGamma);
}
```
Example 3: Film Stock Color Response

- Physical film’s color response for one color channel depends on value of other color channels:

\[ f : \mathbb{R}^3 \rightarrow \mathbb{R}^3 : \begin{pmatrix} r \\ g \\ b \end{pmatrix} \mapsto \begin{pmatrix} f_r(r, g, b) \\ f_g(r, g, b) \\ f_b(r, g, b) \end{pmatrix} \]

```cpp
varying vec2 vPixel;
uniform sampler2D oImage;
uniform sampler3D oLookUpTable;

void main()
{
    vec3 vColor = texture2D(oImage, vPixel);
    gl_FragColor = texture3D(oLookUpTable, vColor);
}
```
Convolution Filter

- Belong into class of linear filters
- Interesting because amenable to Fourier analysis
- Described by filter kernel $K_{i,j}$ (discrete case)
- Example kernel size $(2r+1) \times (2r+1)$ indices $i,j$ in $\{-r, \ldots, r\}$.

$$I'_{i,j} = \sum_{u=-r}^{r} \sum_{v=-r}^{r} K_{u,v} I_{i+u,j+v}$$

- Example: $r = 2 \rightarrow 5 \times 5$ filter kernel
Convolution Filter Implementation

- Naïve implementation

```glsl
varying vec2 vPixel;
uniform sampler2D oImage;
uniform sampler2D oKernel;
uniform vec2 vImageScale;
uniform vec2 vWeightScale;

void main() {
    vec4 vSum = vec4(0.0, 0.0, 0.0, 0.0);
    vec2 vOffset;
    for(int i = -N_RADIUS; i < N_RADIUS; i++)
        for (int j = -N_RADIUS; j < N_RADIUS; j++) {
            vOffset = vec2(i, j);
            vSum += texture2D(oImage, vPixel + vImageScale*vOffset)
                * texture2D(oKernel, vec2(N_RADIUS + 1, N_RADIUS + 1)
                               + vWeightScale*vOffset);
        }
    gl_FragColor = vSum;
}
```
Problems with Naïve Implementation

- Other than GeForce 6 Series GPUs unroll the nested loops
  - Max instruction count limits filter size
  - GeForce FX GPUs have 1000 instructions
  - ... other DX9 cards have 96 instructions
- Relatively slow
Minor Improvements

- Symmetrical filter
  - \( K_{i,j} = K_{i,-j} = K_{i,j} = K_{-i,j} \)
    - Single lookup into \( K \) per four pixel lookups.
  - \( K_{i,j} = K_{i,j} \) or \( K_{i,j} = K_{i,-j} \)
    - Single lookup into \( K \) per two pixel lookups.

- Using NV_rectangle with non-parametric texture addressing simplifies texture coordinate calculation.

- Doesn’t really change \( O(n^2) \) complexity.
Separable Convolution Filter

- 2D filter is separable if outer product of two 1D filters:
  - i.e.
  $$S_{i,j} = A_i \cdot B_j$$
- Gaussian filter is separable:
  $$g_{2D}(x, y) = \frac{1}{2\pi\sigma} e^{-\frac{x^2+y^2}{2\sigma^2}} = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{x^2}{2\sigma^2}} \cdot \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{y^2}{2\sigma^2}}$$
  $$= g_{1D}(x) \cdot g_{1D}(y)$$
Separable Filters Implementation

- Complexity reduced to $O(2n)$
- Shorter programs allow for larger filter kernels
- But implementation requires two passes:
  - OpenGL various options:
    - `glCopyTexImage2D()` copy frame-buffer data to texture.
    - Render-to-texture P-Buffers
Performance Tricks

- Texture filtering math is free
- Can do bi-linear interpolation of 4 texel values per texture access.
- One-dimensional example:
  - Naïve implementation:
    \[
    F_i = \sum_{k=-r}^{r} w_r I_{i+r}
    \]
  - Using \texttt{GL\_LINEAR} filtering hardware calculates
    \[
    I(x) = (1 - \alpha)I_i + \alpha I_{i+1} \quad \text{with} \quad i := \lfloor x \rfloor \quad \text{and} \quad \alpha := x - i
    \]
Texture Filtering

- Idea: Position sample location according to weights $w_n$ and $w_{n+1}$
- Half the number of texture lookups!

$$w'_k = w_{2k-r} + w_{2k-r+1} \text{ and } \alpha_k = \frac{w_{2k-r+1}}{w_{2k-r} + w_{2k-r+1}}$$

\[ \begin{array}{c c c}
\alpha_k & (1-\alpha_k) \\
\hline
w'_k & w'_k+1
\end{array} \]
Image Processing Summary

- Powerful fragment programs allow
  - implementation of wide variety of image processing task
  - unified approach to GPU image processing.
Demo

- SDK Example available at developer.nvidia.com
- FX Composer image processing examples
Blending on the GPU

- Blending in OpenGL
  - Simple Porter-Duff blending
  - Blend modes in OpenGL 1.5
- “Manual Blending” in Fragment Program
Porter-Duff Blending Algebra

- Color C=(r,g,b) plus coverage \( \alpha \) represented as pre-multiplied color \( c=(r\alpha, g\alpha, b\alpha, \alpha) \)
- Various advantages:
  - Ready for display
    - anti-aliased image on black background.
  - Simple blending equation:
    - \( c_O = F_Ac_A + F_Bc_B \)
    - Porter-Duff operators set \( F_A \) and \( F_B \) to 0, 1, \( \alpha_{A/B} \), and 1- \( \alpha_{A/B} \)
Blending in OpenGL

- Equation:
  \[ c_o = \begin{pmatrix}
  r_S F_r^S + r_D F_r^D \\
  g_S F_g^S + g_D F_g^D \\
  b_S F_b^S + b_D F_b^D \\
  \alpha_S F_\alpha^S + \alpha_D F_\alpha^D
  \end{pmatrix} \]

- Specify blend factors via:
  - `glBlendFunc(source, destination);`
    - `GL_ZERO, GL_ONE`
    - `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`
    - `GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA`
    - `GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR`
    - `GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR`
More Flexibility–OpenGL 1.5

• OpenGL 1.5 integrated several extensions into the standard:
  • EXT_blend_color
  • EXT_blend_equation_separate
  • EXT_blend_minmax
  • EXT_blend_subtract
**EXT_blend_color**

- Specify a fixed blend factor \((F_r, F_g, F_b, F_a)\)
  - `glBlendColor(r, g, b, a)`
- Blend factors tokens:
  - `GL_CONSTANT_COLOR`
  - `GL_ONE_MINUS_CONSTANT_COLOR`
  - `GL_CONSTANT_ALPHA`
  - `GL_ONE_MINUS_CONSTANT_ALPHA`
- Example:
  - Simulate photographic color filters
  - Blend between complete images without touching the image data’s alpha.
**EXT_blend_equation_separate**

- Specify different blend-factors for color and alpha
  - `glBlendFuncSeparate(srcRGB, dstRGB, srcAlpha, dstAlpha)`
EXT_blend_minmax & EXT_blend_subtract

- Change the underlying blend equation:
  - `GL_FUNC_ADD` sets default: $C = C_sS + C_dD$
  - `GL_FUNC_SUBTRACT` sets: $C = C_sS - C_dD$
  - `GL_FUNC_REVERSE_SUBTRACT` sets: $C = C_dD - C_sS$
  - `GL_MIN` sets: $C = \min(C_s, C_d)$
  - `GL_MAX` sets: $C = \max(C_s, C_d)$
What More Could You Want?

- 16 bit floating-point blending only available on GeForce 6 Series
- 32 bit floating-point blending not available yet
- Example: Adobe’s “basic” compositing formula
  \[ C_r = (1 - \frac{\alpha_s}{\alpha_r}) \times C_b + \frac{\alpha_s}{\alpha_r} \times [(1 - \alpha_b) \times C_s + \alpha_b \times B(C_b, C_s)] \]
- \( B(C_b, C_s) \) determines blending modes like ColorDodge, etc.
  - see PDF Reference Manual
Blending using Fragment Shader

- Complex math no problem but...
- Shader can’t access frame-buffer
- Workaround:
  - Copy FB to texture.
  - Use texture to get FB data.
  - Catch: slower than native blending
Normal Blending

- Normal draw loop (GPU takes care of blending):

  ```
  clearBuffer(backgroundColor);
  for all Object in Scene:
      setBlendFunc(Object.blendFunc);
      render(Object);
      displayBuffer();
  ```
clearBuffer(background);
copyBuffer(bufferSize, texture);

for all Object in Scene:
    setShader(Object.blendShader);
    bind(texture);
    render(Object);
    unbind(texture);
    copyBuffer(Object.boundingBox, oTexture);

display(oTexture);

etc.

“Manual” Blending
Conclusion

- GPUs natively support Porter-Duff blending
- Additional flexibility:
  - Source/destination color, constant color and alpha as blend factors
  - Subtraction, min/max blend functions
- Fragment shader blending
  - Total flexibility
  - More programming overhead
  - Performance penalty
Painting

- Basic idea
- Simple Soft-Brush
- Clone Brush
- Liquefy Brush
Basic Idea

- Use over-operator to compose brush with background.
- Brush could be:
  - Geometric primitive
  - Texture
- Fractional alpha-values: Anti-Aliasing
Soft-Edged Circular Brush

- Brush has “hardness” control (h)
Soft-Edged Circular Brush

- $x < h$ brush completely opaque
- $x > 1$ fully transparent
- smooth fall-off

$$\alpha(x) = 1 - \text{smoothstep}(h, 1, x)$$

$$\text{smoothstep}(a, b, x) = \begin{cases} 
0 & \text{if } x < a \\
3(x-a)^2 - 2(x-a)^3 & \text{if } a \leq x < b \\
1 & \text{if } x \geq b
\end{cases}$$
Drawing the Brush

• If blend mode supported by HW simply draw quad

```cpp
varying vec2 vUV;
uniform sampler2D oBrush;
uniform vec4 vColor;

void main()
{
    gl_FragColor = vColor * texture2D(oBrush, vUV).a;
}
```
“Manually Blending” the Brush

```glsl
varying vec2 vBrushUV;
varying vec2 vBackgroundUV;

uniform sampler2D oBrush;
uniform sampler2D oBackground;

uniform vec4 vColor;

void main()
{
  float nAlpha = texture2D(oBrush,vBrushUV).a;
  gl_FragColor = nAlpha * vColor
               + (1-nAlpha) * texture2D(oBackground,
                                       vBackgroundUV);
}
```

©2004 NVIDIA Corporation. All rights reserved.
Clone Brush

- Manual Blending gives access to “background”
- Use background twice, as source AND destination

```glsl
varying vec2 vBrushUV;
varying vec2 vDstUV;
varying vec2 vSrcUV;

uniform sampler2D oBrush;
uniform sampler2D oImage;
uniform vec4 vColor;

void main()
{
    float nAlpha = texture2D(oBrush,vBrushUV).a;
    gl_FragColor = nAlpha * texture2D(oImage, vSrcUV) + (1-nAlpha) * texture2D(oImage, vDstUV);
}
```
Liquefy Brush

- Liquefy Brush drags colors with it
- new pixel-values found in opposite direction as brush stroke.
Liquefy Brush (cont’d)

- Idea: Don’t manipulate image but paint (store) brush stokes in separate offset texture.
  - offset texture in floating point format
  - paint x-motion in red channel, y-motion in green channel.
- Use original image and offset texture to render final image.
Liquefy Shader

```
varying vec2 vUV;

uniform sampler2D oImage;
uniform sampler2D oOffset;

uniform float nScale;

void main()
{
    vec2 vOffset = nScale * texture2D(oOffset, vUV);
    gl_FragColor = texture2D(oImage, vUV - vOffset);
}
```
Conclusion

- Painting is Compositing
- Example brushes
  - simple fragment programs
  - used “manual blending” for clone brush
Paint Demo

- Courtesy of Simon Green

**HDR Paint**

This example demonstrates the use of floating point textures and render-to-texture to implement interactive high dynamic range painting. It uses fragment programs to implement several different display and brush modes. The application is resolution-independent - all rendering is performed to an offscreen floating point pbuffer, which can then be displayed at any size or position. Each brush stroke is rendered as a single textured quad. Floating point blending is implemented in the shader using two pbuffers which are alternated between each brush stroke. One is used as the source buffer and the other is the destination. The modified area is copied back from the destination to the source for the next frame.
Apple Motion
Questions
clearBuffer1(backgroundColor);
clearBuffer2(backgroundColor);

pActive = Buffer1;
pTexture = Buffer2;

for all Object in Scene:
    setShader(Object.blendShader);
    bind(pTexture);
    render(Object);
    swap(pActive, pTexture);
    setShader(copyShader);
    bind(pTexture);
    render(Object.boundingBox);
    display(pActive)