Real-Time 3D Scene Post-processing

Jason L. Mitchell
ATI Research
JasonM@ati.com
Overview

• Photorealistic Rendering
  – High Dynamic Range Rendering
  – Depth Of Field

• Non-Photorealistic Rendering
  – Halftoning
  – Posterization
  – Edge outlining
We seek to mimic the physical properties of real world illumination and the imaging devices used to photograph it

- **High Dynamic Range**
  - The range and precision of illumination in the real world far exceeds the values traditionally stored in frame buffers

- **Bloom**
  - The film/sensors in cameras can cause oversaturated values to bleed into neighboring cells or regions of the film

- **Depth of Field**
  - The optics of cameras don’t capture perfectly crisp images from the real-world. This is both an artifact and a creative tool.
High Dynamic Range Rendering
HDR Rendering Process

Scene Geometry lit with HDR Light Probes

HDR Scene → Bloom Filter → + → Tone Map → Displayable Image

Image Space Operations
Frame Postprocessing

HDR Scene

- ¼ Size Frame
- Vertical Gaussian Filter
- Horizontal Gaussian Filter

One Final Pass

Tone Map

One Pass Each

Real-Time 3D Scene Post-processing – Game Developer’s Conference 2003
Separable Gaussian Filter

- Some filters, such as a 2D Gaussian, are separable and can be implemented as successive passes of 1D filter
- We will do this by rendering into temporary buffer, sampling a line or column of texels on each of two passes
- One center tap, six inner taps and six outer taps
- Sample 25 texels in a row or column using a layout as shown below:
float4 hls1_gaussian_x (float2 tapZero : TEXCOORD0, float2 tap12 : TEXCOORD1,
    float3 tapMinus12 : TEXCOORD2, float2 tap34 : TEXCOORD3,
    float2 tapMinus34 : TEXCOORD4, float3 tap56 : TEXCOORD5,
    float3 tapMinus56 : TEXCOORD6 ) : COLOR
{
    float4 accum, Color[NUM_INNER_TAPS];
    Color[0] = tex2D (nearestImageSampler, tapZero); // sample 0
    Color[1] = tex2D (linearImageSampler, tap12); // samples 1, 2
    Color[2] = tex2D (linearImageSampler, tapMinus12); // samples -1, -2
    Color[3] = tex2D (linearImageSampler, tap34); // samples 3, 4
    Color[4] = tex2D (linearImageSampler, tapMinus34); // samples -3, -4
    Color[5] = tex2D (linearImageSampler, tap56); // samples 5, 6
    Color[6] = tex2D (linearImageSampler, tapMinus56); // samples -5, -6

    accum = Color[0] * gTexelWeight[0]; // Weighted sum of samples
    accum += Color[1] * gTexelWeight[1];
    accum += Color[2] * gTexelWeight[1];
    accum += Color[3] * gTexelWeight[2];
    accum += Color[4] * gTexelWeight[2];
    accum += Color[5] * gTexelWeight[3];
    accum += Color[6] * gTexelWeight[3];

    ...

Separable Gaussian Blur Part 2

```c
// Sample the outer taps
for (int i=0; i<NUM_OUTER_TAPS; i++)
{
    Color[i] = tex2D (linearImageSampler, outerTaps[i]);
}

accum += Color[0] * gTexelWeight[4]; // Accumulate outer taps
accum += Color[1] * gTexelWeight[4];
accum += Color[2] * gTexelWeight[5];
accum += Color[3] * gTexelWeight[5];
accum += Color[4] * gTexelWeight[6];
accum += Color[5] * gTexelWeight[6];
return accum;
```
Tone Mapping

Very Underexposed

Underexposed

Good exposure

Overexposed
Tone Mapping Shader

```c
float4 hls1_tone_map (float2 tc : TEXCOORD0) : COLOR
{
    float fExposureLevel = 32.0f;

    float4 original = tex2D (originalImageSampler, tc);
    float4 blur     = tex2D (blurImageSampler, tc);

    float4 color = lerp (original, blur, 0.4f);

    tc -= 0.5f; // Put coords in -1/2 to 1/2 range

    // Square of distance from origin (center of screen)
    float vignette = 1 - dot(tc, tc);

    // Multiply by vignette to the fourth
    color = color * vignette*vignette*vignette*vignette;

    color *= fExposureLevel; // Apply simple exposure level
    return pow (color, 0.55f); // Apply gamma and return
}
```
Depth Of Field

ATI Bacteria Screensaver
Depth Of Field

- Important part of photo-realistic rendering
- Computer graphics uses a pinhole camera model
- Real cameras use lenses with finite dimensions
- See Potmesil and Chakravarty 1981 for a good discussion
Camera Models

- Pinhole lens lets only a single ray through
- In thin lens model if image plane isn’t in focal plane, multiple rays contribute to the image
- Intersection of rays with image plane approximated by circle
Real-time Depth Of Field Implementation On Radeon 9700

- Use MRT to output multiple data – color, depth and “blurriness” for DOF post-processing

- Use pixel shaders for post-processing
  - Use post-processing to blur the image
  - Use variable size filter kernel to approximate circle of confusion
  - Take measures to prevent sharp foreground objects from “leaking” onto background
Depth Of Field Using MRT

Pixel Pipeline Output

- Depth and “blurriness” in 16-bit FP format
- Blurriness computed as function of distance from focal plane

Real-Time 3D Scene Post-processing – Game Developer's Conference 2003
Circle Of Confusion Filter Kernel

- Vary kernel size based on the “blurriness” factor
Elimination Of “Leaking”

- Conventional post-processing blur techniques cause “leaking” of sharp foreground objects onto blurry backgrounds
- Depth compare the samples and discard ones that can contribute to background “leaking”
Semantic Depth Of Field

• Semantic depth of field – sharpness of objects controlled by “relevance”, not just depth
• Easy to accommodate with our technique
  – “Blurriness” is separate from depth
• Can be used in game menus or creatively in real-time cinematics to focus on relevant scene elements
Depth Of Field

Focal plane distance: 5.0
Max circle of confusion: 5.0
Focus range: 3.0
Depth Of Field

Focal plane distance: 8.4
Max circle of confusion: 5.0
Focus range: 3.0
Semantic Depth Of Field
float4 hlsl_depth_of_field_loop (float2 centerTap : TEXCOORD0) : COLOR
{
    float2 tap[NUM_DOF_TAPS];
    float4 Color[NUM_DOF_TAPS];
    float2 Depth[NUM_DOF_TAPS];

    // Fetch center samples from depth and focus maps
    float4 CenterColor = tex2D (ColorSampler, centerTap);
    float2 CenterFocus = tex2D (DoFSampler, centerTap);
    float fTotalContribution = 1.0f;
    float fContribution;
    float fCoCSize = CenterFocus.y * gMaxCoC; // Scale the Circle of Confusion

    for (int i=0; i<NUM_DOF_TAPS; i++) // Run through all of the taps
    {
        // Compute tap locations relative to center tap
        tap[i] = fCoCSize * gTapOffset[i] + centerTap;
        Color[i] = tex2D (ColorSampler, tap[i]);
        Depth[i] = tex2D (DoFSampler, tap[i]);

        // Compute tap's contribution to final color
        fContribution = (Depth[i].x > CenterFocus.x) ? CenterFocus.y : Depth[i].y;
        CenterColor += fContribution * Color[i];
        fTotalContribution += fContribution;
    }

    float4 FinalColor = CenterColor / fTotalContribution; // Normalize
    return FinalColor;
}
Non-Photorealistic Rendering

- Key Elements of NPR
  - Comprehensible
  - Visually interesting
    - Often inspired by artifacts of other media

- Today we’ll discuss three topics
  - Halftoning
  - Posterization
  - Edge outlining
Halftoning

- Artifact of the printing process
- Dots of a common color but of varying sizes are used to give a sense of shading
- Interesting stylistic element used often by pop artists such as Roy Lichtenstein
- Often combined with solid black outlining
Halftoning Process

- Render scene at lower resolution than screen
- Each $n \times n$ region of the screen will represent one pixel from the low resolution image
- Use a screen to define the shape of the dots
  - Can get very creative here…artistic screening
Halftoning Process
Halftoning Shader

```
sampler LowResolutionImageSampler;
sampler HalftoneScreenSampler;

float4 main (float2 LowResImageCoords : TEXCOORD0,
            float2 ScreenCoords : TEXCOORD1) : COLOR
{
    float4 vImage = tex2D(LowResolutionImageSampler, LowResImageCoords);
    float4 Screen = tex2D(HalftoneScreenSampler, ScreenCoords);

    // If color is dimmer than screen, output color, else output white
    if(dot(vImage, float4(0.3f, 0.59f, 0.11f, 0.0f)) < Screen.r)
    {
        return vImage;
    }
    else
    {
        return float4 (1.0f, 1.0f, 1.0f, 1.0f);
    }
}
```
Natural Extensions

• Artistic Screening
  – Just play games with the tiling screen
  – The *Matrix: Reloaded* teaser does something like this with their glowing green flipped katakana characters

• Image mosaics
  – Use 3D texture to store series of images
  – Compute luminance and use as texture coordinate along axis which selects proper sub-image (probably want to use nearest filtering on that axis)
Hand-Drawn Posterized Style

• Inspired by *Waking Life*
• Apply edge-preserving Kuwahara filter
• Optionally composite edges on top

Frame from *Waking Life*
Posterization

• Kuwahara Filter
  – Non-linear edge-preserving smoothing operation
  – 4 overlapping 3×3 regions
  – Compute variance and mean brightness for each region
  – Output is the mean value of the region with smallest variance
Posterization

Original Image

Posterization & Edges

Real-Time 3D Scene Post-processing – Game Developer’s Conference 2003
One 3x3 Region of Kuwahara Filter

```cpp
sampler texture0: register (s0);
float2 sampleOffsets[8]: register (c10);

struct PS_INPUT
{ float2 texCoord:TEXCOORD0; };

float4 main( PS_INPUT In ) : COLOR
{ int i =0;
  float4 c = .5;
  float2 texCoords[9];
  float4 texSamples[9], mean, total = 0;
  float variance;

  // By adding In.texCoord to sampleOffsets[5] (c15) we get the total offset
  // for the 3x3 neighborhood that we can add to all samples.
  texCoords[0] = In.texCoord + sampleOffsets[5];

  for(i =1; i < 9; i++)
  { texCoords[i] = texCoords[0] + sampleOffsets[i-1]; }

  for(i=0; i <9; i++) // tap everyone at once
  { texSamples[i] = tex2D( texture0, texCoords[i]); }

  for(i=0; i <9; i++) // compute the mean
  { total += texSamples[i]; }

  mean = total / 9.0;
  total = 0;

  // now compute the squared variance
  for(i=0; i < 9 ; i++)
  { total += (mean-texSamples[i])*(mean-texSamples[i]); }

  variance = dot(total,1.0/9.0); // we dont need the root, but we need to add the r,g,and b 's together.
  c.xyz = mean;
  c.a = variance;
  return c;
}
```
Variance Selection

```c
sampler ulTexture : register (s11);
sampler urTexture : register (s12);
sampler llTexture : register (s13);
sampler lrTexture : register (s14);

struct PS_INPUT
{ float2 texCoord:TEXCOORD0; }

float4 main( PS_INPUT In )
{
    int i =0;
    float4 sample[4], s0, s1, s2, s3, lowestVariance, l2;
    float s0a, s1a, s2a, s3a, la, l2a;

    s0 = tex2D( ulTexture, In.texCoord);    s1 = tex2D( urTexture, In.texCoord);
    s2 = tex2D( llTexture, In.texCoord);    s3 = tex2D( lrTexture, In.texCoord);
    s0a = s0.a;  s1a = s1.a;  s2a = s2.a;  s3a = s3.a;

    if( s0a < s1a )
    {    lowestVariance = s0;
        la = s0a;
    }
    else
    {    lowestVariance = s1;
        la = s1a;
    }

    if( s2a < s3a )
    {    l2 = s2;
        l2a = s2a;
    }
    else
    {    l2 = s3;
        l2a = s3a;
    }

    if( l2a < la )
    {    lowestVariance = l2;
    }

    return lowestVariance;
}
```
Image Space Outlining for NPR

- Render alternate representation of scene into texture map
  - With the RADEON 9700, we’re able to render into up to four targets simultaneously, effectively implementing Saito and Takahashi’s G-buffer

- Run filter over image to detect edges
  - Implemented using pixel shading hardware
Normal and Depth

World Space Normal

Eye Space Depth

Real-Time 3D Scene Post-processing – Game Developer's Conference 2003
Outlines

Normal Edges

Depth Edges
Normal and Depth Negated in Shadow

World Space Normal Negated in Shadow

Eye Space Depth Negated in Shadow
Normal and Depth Outlines

Edges from Normals

Edges from Depth

Real-Time 3D Scene Post-processing – Game Developer’s Conference 2003
Object and Shadow Outlines

Outlines from selectively negated normals and depths

Real-Time 3D Scene Post-processing – Game Developer’s Conference 2003
Texture Region IDs
Edges at Texture Region Boundaries
Edge Filter

5-tap Filter
Edge Filter Code

```plaintext
ps.2.0
def c0, 0.0f, 0.80f, 0, 0
def c3, 0, .5, 1, 2
def c8, 0.0f, 0.0f, -0.01f, 0.0f
def c9, 0.0f, -0.25f, 0.25f, 1.0f
def c12, 0.0f, 0.01f, 0.0f, 0.0f
dcl_2d s0
dcl_2d s1
dcl t0
dcl t1
dcl t2
dcl t3
dcl t4

// Sample the map five times

texld r0, t0, s0 // Center Tap

texld r1, t1, s0 // Down/Right

texld r2, t2, s0 // Down/Left

texld r3, t3, s0 // Up/Left

texld r4, t4, s0 // Up/Right

// NORMALS

mad r0.xyz, r0, c3.w, -c3.z
mad r1.xyz, r1, c3.w, -c3.z
mad r2.xyz, r2, c3.w, -c3.z
mad r3.xyz, r3, c3.w, -c3.z
mad r4.xyz, r4, c3.w, -c3.z

// Take dot products with center

dp3 r5.r, r0, r1

dp3 r5.g, r0, r2

dp3 r5.b, r0, r3

dp3 r5.a, r0, r4

// Subtract threshold

sub r5, r5, c0.g

// Make 0/1 based on threshold

cmp r5, r5, cl.g, cl.r

// detect any 1's

dp4_sat r11, r5, c3.z

// Take four deltas

add r10.r, r0.a, -r1.a
add r10.g, r0.a, -r2.a
add r10.b, r0.a, -r3.a
add r10.a, r0.a, -r4.a

// Take absolute value

cmp r10, r10, r10, -r10

// Subtract threshold

add r10, r10, c8.b

// Make black/white

cmp r10, r10, cl.r, c1.g
dp4_sat r10, r10, c3.z

// Sum up detected pixels

mad_sat r10, r10, cl.b, c1.w
mul r11, r11, r10

// Scale and bias result

// Combine with previous

// TexIDs

// Sample the map five times

texld r0, t0, s1 // Center Tap
texld r1, t1, s1 // Down/Right
texld r2, t2, s1 // Down/Left
texld r3, t3, s1 // Up/Left
texld r4, t4, s1 // Up/Right

// Get differences in color

sub r1.rgb, r0, r1
sub r2.rgb, r0, r2
sub r3.rgb, r0, r3
sub r4.rgb, r0, r4

// Calculate magnitude of color differences

dp3 r1.r, r1, c3.z
dp3 r1.g, r2, c3.z
dp3 r1.b, r3, c3.z
dp3 r1.a, r4, c3.z

// Take absolute values

cmp r1, r1, r1, -r1

// Subtract threshold

sub r1, r1, cl2.g

// Make black/white

cmp r1, r1, cl.r, c1.g
dp4_sat r10, r1, c3.z

// Total up edges

mad_sat r10, r10, cl.b, c1.w
mul r11, r11, r10

// Scale and bias result

// Combine with previous

// Output

mov oC0, r11
```

Real-Time 3D Scene Post-processing – Game Developer's Conference 2003
Morphology

Real-Time 3D Scene Post-processing – Game Developer’s Conference 2003
Dilation Shader

ps.2.0

def c0, 0, .5, 1, 2
def c1, 0.4f, -1, 5.0f, 0
dcl_2d s0
dcl t0
dcl t1
dcl t2
dcl t3
dcl t4

// Sample the map five times
texld r0, t0, s0 // Center Tap
texld r1, t1, s0 // Up
texld r2, t2, s0 // Left
texld r3, t3, s0 // Down
texld r4, t4, s0 // Right

// Sum the samples
add r0, r0, r1
add r1, r2, r3
add r0, r0, r1
add r0, r0, r4
mad_sat r0, r0.r, c1.r, c1.g // Threshold
mov oC0, r0
Real World Example

*MotoGP* from Climax Brighton

Images Courtesy Shawn Hargreaves @ Climax Brighton

Image space outlining over toon shading
Summary

• Photorealistic Rendering
  – High Dynamic Range Rendering
  – Depth Of Field

• Non-Photorealistic Rendering
  – Halftoning
  – Posterization
  – Edge outlining
Acknowledgments

• Thanks to…
  – John Isidoro for the HDR shaders
  – Guennadi Riguer for the Depth of Field app
  – Marwan Ansari for the Kuwahara filter implementation
For More information

- ATI Developer Relations
  - www.ati.com/developer
- High Dynamic Range Rendering
  - www.debevec.org
- Great NPR Books
  - Strothotte & Schlechtweg
    - isgwww.cs.uni-magdeburg.de/pub/books/npr/
  - Gooch & Gooch
    - www.cs.utah.edu/npr/
- Great NPR Papers
  - Saito & Takahashi
  - Ostrovoukhov