Real-Time 3D Scene Post-processing

Chris Oat
ATI Research
Overview

• Image Space Post Processing
  – High Dynamic Range Rendering
  – Blooms, Tone Mapping and Vignettes
  – Depth Of Field
  – Heat and Haze
Photorealistic Rendering

- We seek to mimic the physical properties of real world illumination and the imaging devices used to capture it
- High Dynamic Range Imaging
  - 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.
- Heat and Haze Effects
  - Changes in atmospheric density cause light to bend producing shimmering effects
High Dynamic Range Rendering
Frame Postprocessing

HDR Scene

One Pass Each

¼ Size Frame

Vertical Gaussian Filter

Horizontal Gaussian Filter

Tone Map

One Final Pass
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:

```
-6 -5 -4 -3 -2 -1  0  1  2  3  4  5  6
```

Gaussian Filter

- Center Tap (nearest filtering)
- Inner Tap
- Outer Tap, dependent read
- Texel
Separable Gaussian Blur Part 1

```c
float4 hlsl_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];
}
```

Real-Time 3D Scene Post-processing – Game Developer’s Conference Europe 2003
Separable Gaussian Blur Part 2

```c
float2 outerTaps[NUM_OUTER_TAPS];
outerTaps[0] = tapZero + gTexelOffset[0];    // coord for samples 7, 8
outerTaps[1] = tapZero - gTexelOffset[0];    // coord for samples -7, -8
outerTaps[2] = tapZero + gTexelOffset[1];    // coord for samples 9, 10
outerTaps[3] = tapZero - gTexelOffset[1];    // coord for samples -9, -10
outerTaps[4] = tapZero + gTexelOffset[2];    // coord for samples 11, 12
outerTaps[5] = tapZero - gTexelOffset[2];    // coord for samples -11, -12

// 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;
```

Real-Time 3D Scene Post-processing – Game Developer's Conference Europe 2003
Tone Mapping

Very Underexposed

Underexposed

Good exposure

Overexposed
Tone Mapping Shader

```hlsl
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
}
```
Kawase’s Bloom Filter

- Developed by Masaki Kawase of Bunkasha Games
- Used in DOUBLE-S.T.E.A.L. (aka Wreckless)
- From his GDC2003 Presentation: Frame Buffer Postprocessing Effects in DOUBLE-S.T.E.A.L (Wreckless)
Kawase’s Bloom Filter

- Start by downsampling ($\frac{1}{4} \times \frac{1}{4} = \frac{1}{16}$)
- Apply small blur filters repeatedly
- More iterations for more blurriness
- Each iteration “Ping-Pongs” between two renderable textures used for storing intermediate results

Pixel being Rendered
Texture sampling points
Iterating Kawase’s Bloom Filter

Each iteration takes the previous results as input and applies a new kernel to increase blurriness.
float3 SiWrecklessBloomFilterRGB(sampler tSource, float2 texCoord, float2 pixelSize, int iteration)
{
    float2 texCoordSample = 0;
    float2 halfPixelSize = pixelSize / 2.0f;
    float2 dUV = (pixelSize.xy * float(iteration)) + halfPixelSize.xy;
    float3 cOut;

    // Sample top left pixel
    texCoordSample.x = texCoord.x - dUV.x;
    texCoordSample.y = texCoord.y + dUV.y;
    cOut = tex2D(tSource, texCoordSample);

    // Sample top right pixel
    texCoordSample.x = texCoord.x + dUV.x;
    texCoordSample.y = texCoord.y + dUV.y;
    cOut += tex2D(tSource, texCoordSample);

    // Sample bottom right pixel
    texCoordSample.x = texCoord.x + dUV.x;
    texCoordSample.y = texCoord.y - dUV.y;
    cOut += tex2D(tSource, texCoordSample);

    // Sample bottom left pixel
    texCoordSample.x = texCoord.x - dUV.x;
    texCoordSample.y = texCoord.y - dUV.y;
    cOut += tex2D(tSource, texCoordSample);

    // Average
    cOut *= 0.25f;

    return cOut;
}
Iterating Kawase’s Bloom Filter

- A few iterations are necessary for pleasing results

Real-Time 3D Scene Post-processing – Game Developer’s Conference Europe 2003
Kawase’s Light Streak Filter

- Caused by diffraction of incoming light
- Diffraction can happen in the eye’s lens or some external surface that scatters light such as a car’s windshield.
Kawase’s Light Streak Filter

Pixel being Rendered
Texture Sampling Point

sample weight = a^(b*s)

a = attenuation (about ~0.9 – 0.95)
b = 4^(n-1)  (where n = pass)

- Start by downsampling (¼ x ¼ = 1/16)
- Same idea as the bloom: repeatedly apply a small blur kernel
- Kernel’s shape is based on direction of streak
Kawase’s Light Streak Filter

- Every iteration of the filter expands the kernel in the direction of the streak
Kawase’s Light Streak Filter

Animating balls are light sources
Streak Filter Extension

- The direction of the streak can be specified per-pixel using an image-space map like those above
- Rotation vector \( <x,y,1> \)

Circular Pattern

Windshield groove pattern due to wiper blades
Streak Filter Extension

Streaks follow windshield groove map
float3 SiWrecklessStreakFilterRGB(sampler tSource, float2 texCoord, float2 pixelSize, float2 streakDirection, int streakSamples, float attenuation, int iteration)
{
    float2 texCoordSample = 0;
    float3 cOut = 0;

    float b = pow(streakSamples, iteration);

    for (int s = 0; s < streakSamples; s++)
    {
        // Weight = a^(b*s)
        float weight = pow(attenuation, b * s);

        // Streak direction is a 2D vector in image space
        texCoordSample = texCoord + (streakDirection * b * float2(s, s) * pixelSize);

        // Scale and accumulate
        cOut += saturate(weight) * tex2D (tSource, texCoordSample);
    }

    return saturate (cOut);
}
Depth Of Field

ATI Bacteria Screensaver

Real-Time 3D Scene Post-processing – Game Developer's Conference Europe 2003
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
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
Semantic Depth Of Field
Depth Of Field Shader

\[
\text{float4 hlsl_depth_of_field_loop (float2 centerTap : TEXCOORD0) : COLOR} \\
\begin{align*}
\text{float2 } & \text{tap[NUM_DOF_TAPS];} \\
\text{float4 } & \text{Color[NUM_DOF_TAPS];} \\
\text{float2 } & \text{Depth[NUM_DOF_TAPS];} \\
\end{align*}
\]

// 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;
}
Heat and Haze Effects

ATI Lava Caves Screensaver

Real-Time 3D Scene Post-processing – Game Developer’s Conference Europe 2003
Heat and Haze

- A natural atmospheric effect that everyone is familiar with
- Light bends as it passes through media of different densities
Heat Shimmering

- Hot air is less dense than cool air
- Density effects a medium’s index of refraction
- As hot air rises it is replaced by cooler air thus changing the way light bends into your line of sight
Heat Shimmering

- Render scene into RGBA off-screen buffer (renderable texture)
  - Color into RGB
  - Distortion weight into Alpha
- Draw full screen quad to backbuffer
  - Sample off-screen buffer to obtain distortion weight
  - Use perturbation map to determine perturbation vector, scale by distortion weight and offset original texture coordinates
  - Growable Poisson Disc filtering using perturbed texture coordinate (grow disc according to distortion weight)
Distortion Weights

- Per-pixel value that determines how much that pixel should be distorted
- The probability of refraction increases as light passes through more atmosphere
- Distortion increases with scene depth
  - Start by clearing render target’s alpha to 1.0 to indicate maximum depth
  - Pixel shader writes per-pixel depth value to alpha
Distortion Weights

- Depth serves as a good global distortion metric but your artists will want more local control
- Heat geometry may be used to define areas of high distortion such as the air above a hot gas vent or behind a jet’s engine
- Heat textures may be used to animate distortion weights across heat geometry
Heat Geometry & Heat Textures

- Per-pixel weights come from Heat Texture
- Weights are Scaled by depth
- Then further scaled by Height (tex coords) and N.V to avoid hard edges
- Distortion weights are written to Alpha

Real-Time 3D Scene Post-processing – Game Developer’s Conference Europe 2003
• Full-Screen quad is drawn using off-screen buffer (renderable texture) and perturbation map as textures
Perturbation Maps

- A 2D Vector stored in Red and Green channels
- Scroll Map in two different directions across full-screen quad and sample twice
- Average both samples then scale and bias to get into \([-1.0, 1.0]\) range
- Scale vector by distortion weight
- Result is the perturbation vector
Perturbation Vector

- Perturbation vector is used to offset original texture coordinate
- Vector’s magnitude is determined by distortion weight
- This new perturbed texture coordinate is used for dependant read into off-screen buffer
Growable Poisson Disc

- Center blur kernel at perturbed texture coordinate
- Grow disc based on distortion weight
float4 main (PsInput i) : COLOR
{
    // fetch from perturbation map with scrolling texture coords
    float3 vPerturb0 = tex2D (tPerturbationMap, i.texCoord1);
    float3 vPerturb1 = tex2D (tPerturbationMap, i.texCoord2);

    // scale and bias: (color - 0.5f)*2.0f
    vPerturb0 = SiConvertColorToVector(vPerturb0);
    vPerturb1 = SiConvertColorToVector(vPerturb1);

    // average perturbation vectors
    float2 offset = (vPerturb0.xy + vPerturb1.xy) * 0.5f;

    // get distortion weight from renderable texture (stored in alpha)
    float4 cDistWeight = tex2D (tRBFullRes, i.texCoord0);

    // square distortion weight
    cDistWeight.a *= cDistWeight.a;

    // compute distorted texture coords
    offset.xy = ((offset.xy * cDistWeight.a) * fPerturbScale) + i.texCoord0;

    // fetch the distorted color
    float4 o;
    o.rgb = SiPoissonDisc13RGB(tRBFullRes, offset, 1.0f/screenRes.xy, cDistWeight.a);
    o.a = 1.0f;
    return o;
}
Growable Poisson Disc Shader

```c
float3 SiGrowablePoissonDisc13FilterRGB(sampler tSource, float2 texCoord, float2 pixelSize, float discRadius) {
    float3 cOut;
    float2 poisson[12] = {float2(-0.326212f, -0.40581f),
                          float2(-0.840144f, -0.07358f),
                          float2(-0.695914f, 0.457137f),
                          float2(-0.203345f, 0.620716f),
                          float2(0.96234f, -0.194983f),
                          float2(0.473434f, -0.480026f),
                          float2(0.519456f, 0.767022f),
                          float2(0.185461f, -0.893124f),
                          float2(0.507431f, 0.064425f),
                          float2(0.519456f, 0.767022f),
                          float2(0.185461f, -0.893124f),
                          float2(0.507431f, 0.064425f)};

    // Center tap
    cOut = tex2D (tSource, texCoord);
    for (int tap = 0; tap < 12; tap++) {
        float2 coord = texCoord.xy + (pixelSize * poisson[tap] * discRadius);

        // Sample pixel
        cOut += tex2D (tSource, coord);
    }
    return (cOut / 13.0f);
}
```

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

• Image Space Post Processing
  – High Dynamic Range Rendering
  – Blooms, Tone Mapping and Vignettes
  – Depth Of Field
  – Heat and Haze
Acknowledgments

• Thanks to...
  – Jason Mitchell for contributing slides used in this presentation
  – John Isidoro for the HDR shaders
  – Guennadi Riguer for helping with Depth of Field
For More information

• ATI Developer Relations
  – http://www.ati.com/developer
• High Dynamic Range Rendering
  – http://www.debevec.org
• Masaki Kawase’s GDC03 Presentation
  – http://bunkasha-games.com
  – http://www.daionet.gr.jp/~masa