VGP352 – Week 8

- Agenda:
  - Post-processing effects
    - Filter kernels
    - Separable filters
    - Depth of field
    - HDR

2-March-2010

Filter Kernels

- Can represent our filter operation as a sum of products over a region of pixels
  - Each pixel is multiplied by a factor
  - Resulting products are accumulated
- Commonly represented as an $n \times m$ matrix
  - This matrix is called the *filter kernel*
  - m is either 1 or is equal to n

Filter Kernels

- Uniform blur over 3x3 area:
  - Larger kernel size results in more blurriness

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

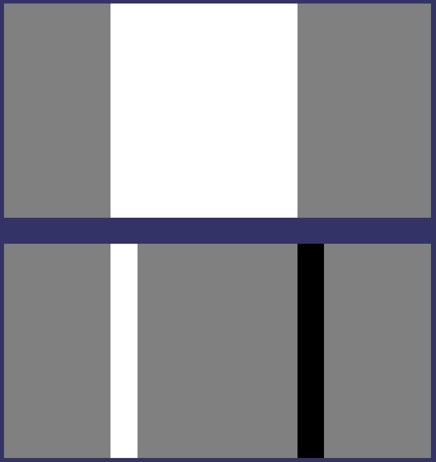
Filter Kernels – Edge Detection

- Edge detection

Filter Kernels – Edge Detection

- Edge detection
  - Take the difference of each pixel and its left neighbor

$$p(x, y) - p(x-1, y)$$



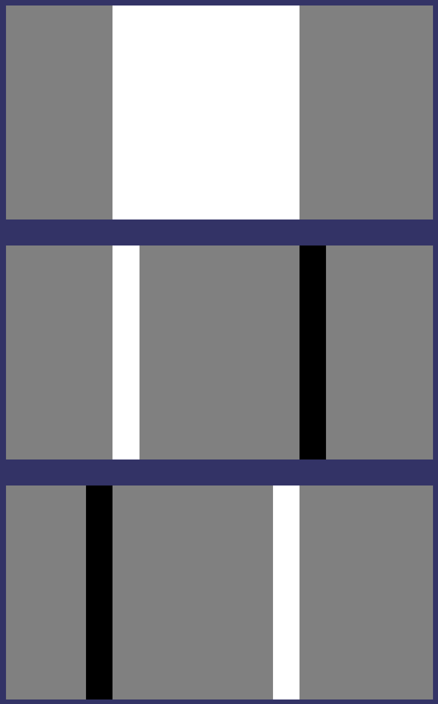
Filter Kernels – Edge Detection

- Edge detection
  - Take the difference of each pixel and its left neighbor

    $$p(x,y) - p(x-1,y)$$

  - Take the difference of each pixel and its right neighbor

    $$p(x,y) - p(x+1,y)$$



Filter Kernels – Edge Detection

- Edge detection
  - Take the difference of each pixel and its left neighbor

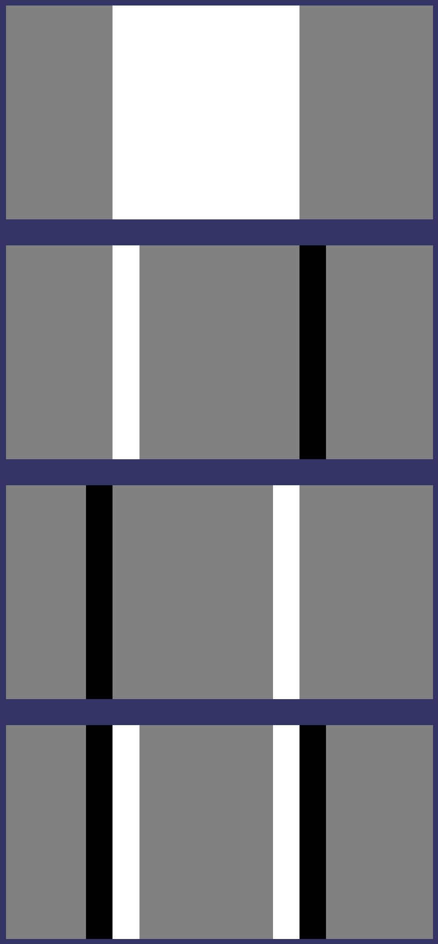
$$p(x,y)-p(x-1,y)$$

  - Take the difference of each pixel and its right neighbor

$$p(x,y)-p(x+1,y)$$

  - Add the two together

$$2p(x,y)-p(x-1,y)-p(x+1,y)$$



Filter Kernels – Edge Detection

- Rewrite as a kernel

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

Filter Kernels – Edge Detection

- Rewrite as a kernel

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

- Repeat in Y direction

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Filter Kernels – Edge Detection

- Rewrite as a kernel

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

- Repeat in Y direction

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

- Repeat on diagonals

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Sobel Edge Detection

- Uses two filter kernels
  - One in the Y direction

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

  - One in the X direction

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

Sobel Edge Detection

- Apply each filter kernel to the image

$$G_x = F_x * A$$
$$G_y = F_y * A$$

  – G_x and G_y are the gradients in the x and y directions
  – The combined magnitude of these gradients can be used to detect edges

$$G = \sqrt{G_x^2 + G_y^2}$$

Sobel Edge Detection

Images from http://en.wikipedia.org/wiki/Sobel_operator

2-March-2010

Filter Kernels

- Implement this easily on a GPU
  - Supply filter kernel as uniforms
  - Perform n^2 texture reads
  - Apply kernel and write result

Filter Kernels

- Implement this easily on a GPU
  - Supply filter kernel as uniforms
  - Perform n^2 texture reads
  - Apply kernel and write result
- Perform n^2 texture reads?!?

Filter Kernels

- Implement this easily on a GPU
  - Supply filter kernel as uniforms
  - Perform n^2 texture reads
  - Apply kernel and write result
- Perform n^2 texture reads?!?
  - n larger than 4 or 5 won't work on most hardware
  - Since the filter is a sum of products, it could be done in multiple passes

Filter Kernels

- Implement this easily on a GPU
  - Supply filter kernel as uniforms
  - Perform n^2 texture reads
  - Apply kernel and write result
- Perform n^2 texture reads?!?
  - n larger than 4 or 5 won't work on most hardware
  - Since the filter is a sum of products, it could be done in multiple passes
  - Or *maybe* there's a different way altogether...

Separable Filter Kernels

- Some 2D kernels can be re-written as the product of 2 1D kernels
  - These kernels are called *separable*
  - Applying each 1D kernel requires n texture reads per pixel, doing both requires $2n$
    - $2n \ll n^2$

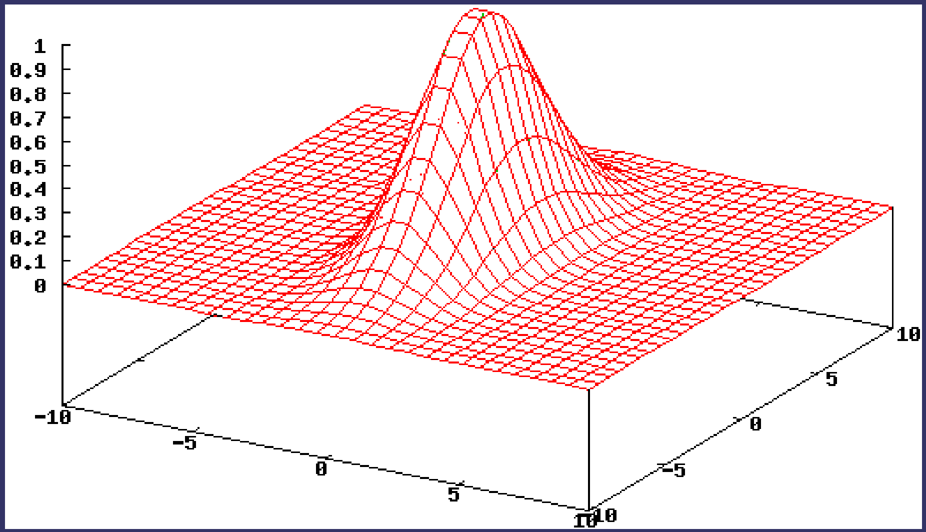
Separable Filter Kernels

- 2D kernel is calculated as the outer-product of the individual 1D kernels

$$\mathbf{A}=\mathbf{a}^{T}\mathbf{b}=\begin{bmatrix} \mathbf{a}_0\mathbf{b}_0 & \cdots & \mathbf{a}_0 b_n \\ \vdots & & \vdots \\ \mathbf{a}_n\mathbf{b}_0 & \cdots & \mathbf{a}_n\mathbf{b}_n \end{bmatrix}$$

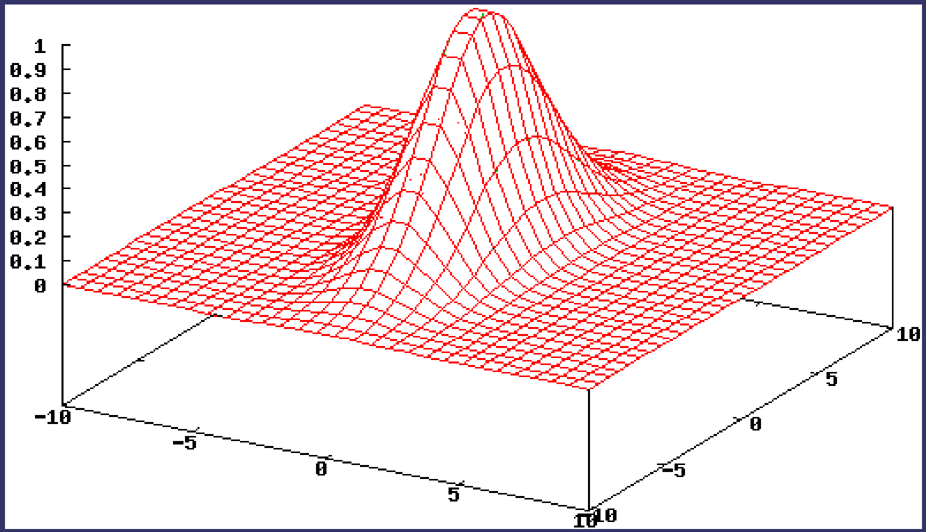
Separable Filter Kernels

- The 2D Gaussian filter is *the classic* separable filter

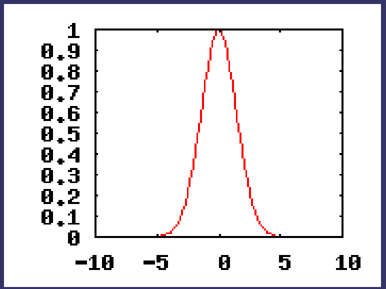


Separable Filter Kernels

- The 2D Gaussian filter is *the classic* separable filter



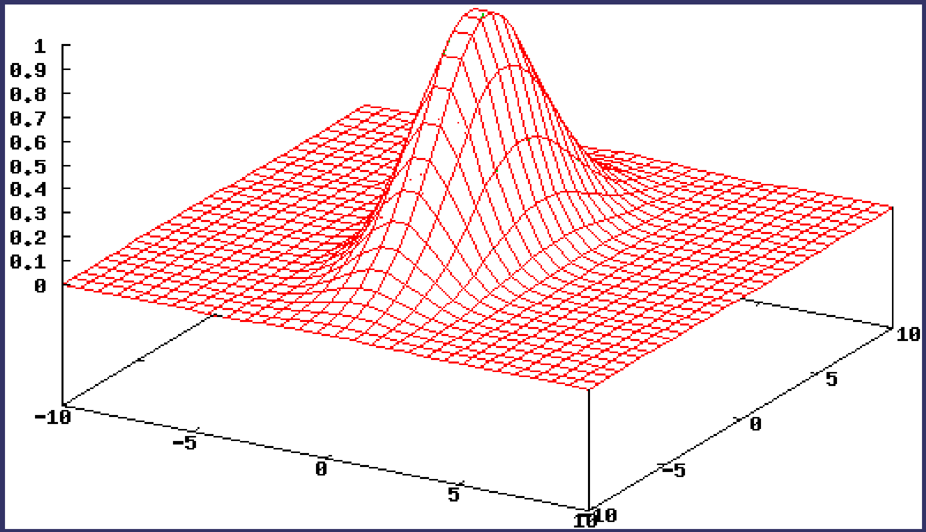
  - Product of a Gaussian along the X-axis



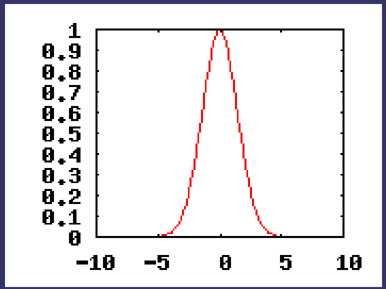
2-March-2010

Separable Filter Kernels

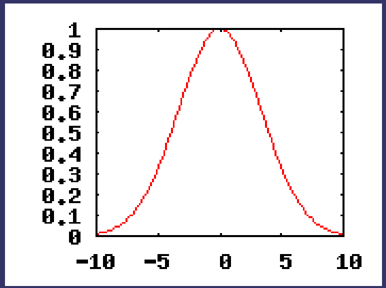
- The 2D Gaussian filter is *the classic* separable filter



  - Product of a Gaussian along the X-axis



  - ...and a Gaussian along the Y-axis



Separable Filter Kernels

- Implementing on a GPU:
  - Use first 1D filter on source image *to window*
  - Configure blending for $source \times destination$

    ```
    glBlendFunc(GL_DST_COLOR, GL_ZERO);
    ```
  - Use second 1D filter on source image *to window*

Separable Filter Kernels

- Implementing on a GPU:
  - Use first 1D filter on source image *to window*
  - Configure blending for $source \times destination$

    ```
    glBlendFunc(GL_DST_COLOR, GL_ZERO);
    ```
  - Use second 1D filter on source image *to window*
- Caveats:
  - Precision can be a problem in intermediate steps
  - May have to use floating-point output
  - Can also use 10-bit or 16-bit per component outputs as well
    - Choice ultimately depends on what the hardware supports

References

http://www.archive.org/details/Lectures_on_Image_Processing

Depth-of-field

- What is depth of field?
  "...the depth of field (DOF) is the portion of a scene that appears acceptably sharp in the image.[1]"

[1] http://en.wikipedia.org/wiki/Depth_of_field
Images also from http://en.wikipedia.org/wiki/Depth_of_field
2-March-2010

Depth-of-field

- Why is DOF important?

Images from http://en.wikipedia.org/wiki/Depth_of_field

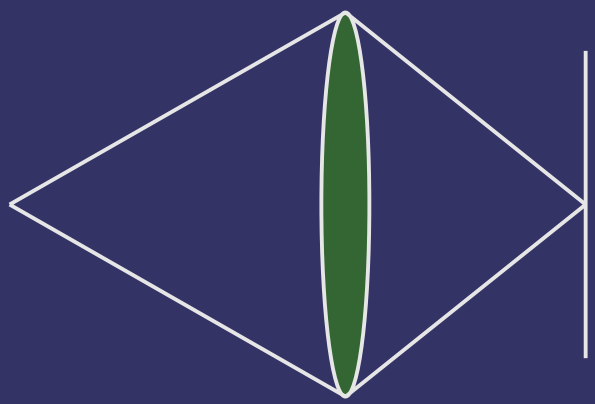
Depth-of-field

- Why is DOF important?
  - Draws viewer's attention
  - Gives added information about spatial relationships
  - etc.

Images from http://en.wikipedia.org/wiki/Depth_of_field

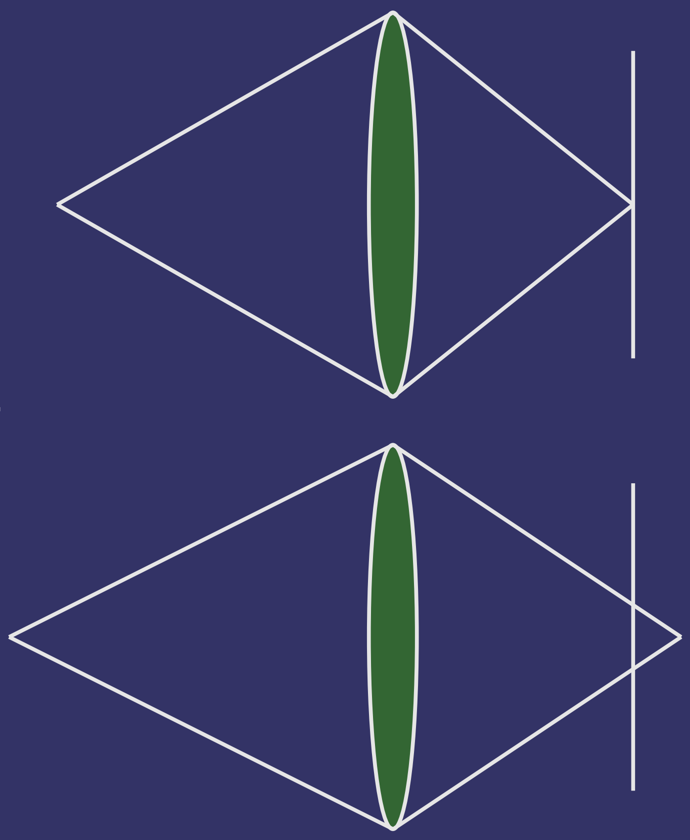
Depth-of-field

- Basic optics:
  - A point of light focused through a lens becomes a point on the object plane



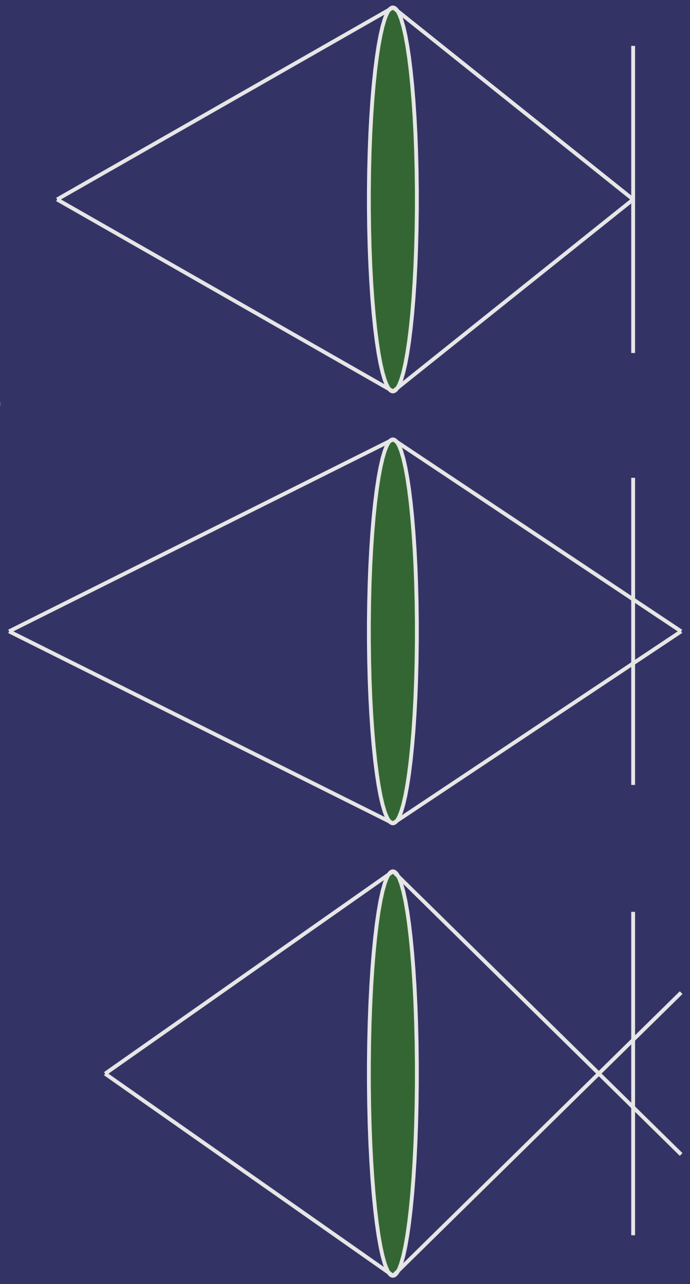
Depth-of-field

- Basic optics:
  - A point of light focused through a lens becomes a point on the object plane
  - A point farther than the focal distance becomes a blurry spot on the object plane



Depth-of-field

- Basic optics:
  - A point of light focused through a lens becomes a point on the object plane
  - A point farther than the focal distance becomes a blurry spot on the object plane
  - A point closer than the focal distance becomes a blurry spot on the object plane
- These blurry spots are called *circles of confusion* (CoC hereafter)



Depth-of-field

- In most real-time graphics, there is no depth-of-field
  - Everything is perfectly in focus all the time

Depth-of-field

- In most real-time graphics, there is no depth-of-field
  - Everything is perfectly in focus all the time
  - Most of the time this is okay
    - The player may want to focus on foreground and background objects in rapid succession. Without eye tracking, the only way this works is to have everything in focus.

Depth-of-field

- In most real-time graphics, there is no depth-of-field
  - Everything is perfectly in focus all the time
  - Most of the time this is okay
    - The player may want to focus on foreground and background objects in rapid succession. Without eye tracking, the only way this works is to have everything in focus.
  - Under some circumstances, DOF can be a *very* powerful tool
    - Non-interactive sequences
    - Special effects
      - Very effective use in the game Borderlands

Depth-of-field

- Straight-forward GPU implementation:
  - Render scene color *and* depth information to off-screen targets
  - Post-process:
    - At each pixel determine CoC size based on depth value
    - Blur pixels within circle of confusion
      - To prevent in-focus data from bleeding into out-of-focus data, do *not* use in-focus pixels that are closer than the center pixel

Depth-of-field

- Problem with this approach?

Depth-of-field

- Problem with this approach?
  - Fixed number of samples within CoC
    - Oversample for small CoC
    - Undersample for large CoC
  - Could improve quality with multiple passes, but performance would suffer

Depth-of-field

- Simplified GPU implementation:
  - Render scene color *and* depth information to off-screen targets
  - Post-process:
    - Down-sample image and Gaussian blur down-sampled image
      - Reduced size and filter kernel size are selected to produce maximum desired CoC size
    - Linearly blend between original image and blurred image based on per-pixel CoC size

Depth-of-field

- Simplified GPU implementation:
  - Render scene color *and* depth information to off-screen targets
  - Post-process:
    - Down-sample image and Gaussian blur down-sampled image
      - Reduced size and filter kernel size are selected to produce maximum desired CoC size
    - Linearly blend between original image and blurred image based on per-pixel CoC size
- Problems with this approach?

Depth-of-field

- Simplified GPU implementation:
  - Render scene color *and* depth information to off-screen targets
  - Post-process:
    - Down-sample image and Gaussian blur down-sampled image
      - Reduced size and filter kernel size are selected to produce maximum desired CoC size
    - Linearly blend between original image and blurred image based on per-pixel CoC size
- Problems with this approach?
  - No way to prevent in-focus data from bleeding into out-of-focus data

References

J. D. Mulder, R. van Liere. *Fast Perception-Based Depth of Field Rendering*, In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology* (Seoul, Korea, October 22 - 25, 2000). VRST '00. ACM, New York, NY, 129-133. http://homepages.cwi.nl/~mullie/Work/Pubs/publications.html

Guennadi Riguer, Natalya Tatarchuk, John Isidoro. *Real-time Depth of Field Simulation*, In *ShaderX2*, Wordware Publishing, Inc., October 25, 2003. http://developer.amd.com/documentation/reading/pages/ShaderX.aspx

M. Kass, A. Lefohn, J. Owens. 2006. *Interactive Depth of Field Using Simulated Diffusion on a GPU.* Technical Memo #06-01, Pixar Animation Studios. http://graphics.pixar.com/library/DepthOfField/

High Dynamic Range

- Until now, our rendering has had a contrast ratio of 256:1
  - As noted in [Green 2004]:
    - Bright things can be really bright
    - Dark things can be really dark
    - And the details can be seen in both

High Dynamic Range

- Several possible solutions depending on hardware support / performance:
  - Render multiple “exposures” and composite results
    - This is how HDR images are captured with a camera
    - Yuck!
  - Render to floating-point buffers
    - Best quality
    - Even fp16 buffers are large / expensive
    - Differing levels of hardware support (esp. on mobile devices)
  - Render to RGBe
    - Smaller / faster
    - Lower quality
    - Issues with blending / multipass

Floating-Point Render Targets

- Create drawing surface with a floating-point internal format
  - Surface is either a texture or a renderbuffer
  - GL_RGB32F, GL_RGBA32F, GL_RGB16F, and GL_RGBA16F are most common
    - Requires GL_ARB_texture_float (and GL_ARB_half_float_pixel for 16F formats) and GL_ARB_color_buffer_float or OpenGL 3.0

Floating-Point Render Targets

- Disable [0, 1] clamping of fragments

```
glClampColorARB(GLenum target, Glenum clamp);
```

  - `target` is one of `GL_CLAMP_VERTEX_COLOR`, `GL_CLAMP_FRAGMENT_COLOR`, or `GL_CLAMP_READ_COLOR`
  - `clamp` is one of `GL_FIXED_ONLY`, `GL_TRUE`, or `GL_FALSE`
  - OpenGL 3.x version drops `ARB` from name

Floating-Point Render Targets

- Common hardware limitations:
  - May not be supported at all!
    - Almost universal on desktop, not so much on mobile
    - Intel GMA950 in most netbooks lacks support
  - May not support blending to floating-point targets
    - `RGBA32F` blending not supported on Geforce6 and similar generation chips
    - May also be *really* slow
  - May not support all texture filtering modes
    - Some hardware can't do mipmap filtering from FP textures
    - Many DX9 era cards can't do any filtering on `RGBA32F` textures

RGBe

- Store R, G, and B mantissa values with a single exponent
  - Exponent store in alpha component
  - Trades precision for huge savings on storage
    - Keeps most of the useful range of FP32

RGBe

- Convert floating-point RGB in shader to RGBe:

```
vec4 rgb_to_rgbe(vec3 color)
{
    const float max_component =
        max(color.r, max(color.g, color.b));
    const float e = ceil(log(max_component));

    return vec4(color / exp(e),
                (e + 128.0) / 255.0);
}
```

RGBe

- Limitations / problems:
  - The `log` and `exp` calls in the shader aren't free
    - May be a problem for compute bound vs. bandwidth bound shaders
  - Blending is still possible, but it is rather painful
  - Can't store components with vastly different magnitudes
    - {10000, 0.1, 0.1 } becomes {10000, 0, 0}
    - *Usually* fine for color data because the final display can't reproduce that much range anyway

Tone Mapping

- Remap HDR rendered image to LDR displayable image
  - Display still limited to [0,1] with only 8-bit precision
- Remap using Reinhard's tone reproduction operator in 5 steps:
  - Convert RGB image to luminance
  - Calculate log-average luminance
    - Used to calculate key value
  - Scale luminance by key value
  - Remap scaled luminance to [0, 1]
  - Scale RGB values by remapped luminance

Tone Mapping

- Standard luminance calculation:

$$l = \begin{bmatrix} 0.2125 & 0.7154 & 0.0721 \end{bmatrix}^T \cdot \mathbf{C}$$

  - If using RGBe, the color must be mapped back from RGBe to floating-point

Tone Mapping

- Image key:

$$k = \frac{1}{n} e^{\sum_{\text{all pixels}} \ln(\partial + l_{x,y})}$$

- Does this pixel averaging operation remind you of anything?

Tone Mapping

- Image key:

$$k = \frac{1}{n} e^{\sum_{\text{all pixels}} \ln(\partial + l_{x,y})}$$

- Does this pixel averaging operation remind you of anything?
  - It's like calculating the lowest-level mipmap!
  - ...but with some other math and emitting HDR

Tone Mapping

- Scaled luminance:

$$l_{scaled} = l_{x,y}\left(\frac{l_{mid\ zone}}{k}\right)$$

  - $l_{mid\ zone}$ is the mid zone reference reflectance value
    - 0.18 is a “common” value... see references
- Remapped luminance:

$$l_{final} = \frac{l_{scaled}}{1 + l_{scaled}}$$

- Final pass modulates l_{final} with original RGB
  - Output in plain old 8-bit RGB, naturally

Tone Mapping

- Can alternately map based on the dimmest value that should be full intensity

$$l_{final} = \frac{l_{scaled}\left(1 + \frac{l_{scaled}}{l_{min\ white}}\right)}{1 + l_{scaled}}$$

  - $l_{minwhite}$ is the minimum HDR intensity that should be mapped to fully bright

Tone Mapping

- Tone map operation is performed each frame

Tone Mapping

- Tone map operation is performed each frame
  - Ouch!
  - Common practice is to only recompute k every few frames
    - Once every half second is common
    - Has the realistic side-effect of not immediately responding to dramatic changes in scene brightness

Bloom

- Overly bright areas leak brightness into neighboring areas

Bloom

- Overly bright areas leak brightness into neighboring areas
  - Apply “bright pass” filter to image
    - Pixels above a certain threshold keep their luminance, everything else becomes black
  - Apply Gaussian blur
  - Add blurred image to final LDR image

Bloom

- Overly bright areas leak brightness into neighboring areas
  - Apply “bright pass” filter to image
    - Pixels above a certain threshold keep their luminance, everything else becomes black
  - Apply Gaussian blur
  - Add blurred image to final LDR image

This step can be very expensive!

Bloom

- Blur optimization:
  - Make multiple down-scaled images (i.e., mipmaps)
    - Largest image should be 1/8th the size of the original
  - Blur each down-scaled image
    - This approximates a doubling of the filter kernel size
  - Apply small filter kernel
    - [Kalogirou 2006] suggests 5x5 is sufficient

References

Simon Green and Cem Cebenoyan (2004). "High Dynamic Range Rendering (on the GeForce 6800)." GeForce 6 Series. nVidia. http://download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_HDR.pdf

Adam Lake, Cody Northrop, and Jeff Freeman. “High Dynamic Range Environment Mapping On Mainstream Graphics Hardware.” 2005. http://www.gamedev.net/reference/articles/article2485.asp

Harry Kalogirou (2006). “How to do good bloom for HDR rendering.” http://harkal.sylphis3d.com/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/

Next week...

- Beyond bumpmaps:
  - Relief textures
  - Parallax textures
  - Interior mapping

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.