VGP352 – Week 10

- Agenda:
  - Texture rectangles
  - Post-processing effects
    - Filter kernels
      - Simple blur
      - Edge detection
    - Separable filter kernels
      - Gaussian blur
    - Depth-of-field

Texture Rectangle

- Cousin to 2D textures

Texture Rectangle

- Cousin to 2D textures
  - Interface changes:
    - New target: `GL_TEXTURE_RECTANGLE_ARB`
    - New sampler: `sampler2DRect`
    - New sampler functions: `texture2DRect`, `texture2DRectProj`, etc.

Texture Rectangle

- Cousin to 2D textures
  - Interface changes:
    - New target: `GL_TEXTURE_RECTANGLE_ARB`
    - New sampler: `sampler2DRect`
    - New sampler functions: `texture2DRect`, `texture2DRectProj`, etc.
  - Minimization filter can only be `GL_LINEAR` or `GL_NEAREST`
  - Coordinate wrap mode can only be `GL_CLAMP`, `GL_CLAMP_TO_EDGE`, or `GL_CLAMP_TO_BORDER`

Texture Rectangle

- Cousin to 2D textures
  - Interface changes:
    - New target: `GL_TEXTURE_RECTANGLE_ARB`
    - New sampler: `sampler2DRect`
    - New sampler functions: `texture2DRect`, `texture2DRectProj`, etc.
  - Minimization filter can only be `GL_LINEAR` or `GL_NEAREST`
  - Coordinate wrap mode can only be `GL_CLAMP`, `GL_CLAMP_TO_EDGE`, or `GL_CLAMP_TO_BORDER`
  - Dimensions need not be power of two

Texture Rectangle

- Cousin to 2D textures
  - Interface changes:
    - New target: `GL_TEXTURE_RECTANGLE_ARB`
    - New sampler: `sampler2DRect`
    - New sampler functions: `texture2DRect`, `texture2DRectProj`, etc.
  - Minimization filter can only be `GL_LINEAR` or `GL_NEAREST`
  - Coordinate wrap mode can only be `GL_CLAMP`, `GL_CLAMP_TO_EDGE`, or `GL_CLAMP_TO_BORDER`
  - Dimensions need not be power of two
  - Texture accessed by coordinates on $[0, w\text{-}1] \times [0, h\text{-}1]$ instead of $[0, 1] \times [0, 1]$

Post-processing Effects

- Apply an *image space* effect to the rendered scene *after* it has been drawn
  - Examples:
    - Blur
    - Enhance contrast
    - Heat “ripple”
    - Color-space conversion (e.g., black & white, sepia, etc.)
    - Many, *many* more

Post-processing Effects

- Overview:
  - Render scene to off-screen target (framebuffer object)
    - Off-screen target should be same size as on-screen window
    - Additional information may need to be generated
  - Render single, full-screen quad to window
    - Use original off-screen target as source texture
    - Configure texture coordinates to cover entire texture
      - Texture rectangles are *really* useful here
    - Configure fragment shader to perform desired effect

Post-processing Effects

- Configure viewport, projection, and modelview matrices to 1-to-1 mapping

```
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, w, 0, h, -1, 1);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

- Draw full-screen quad with appropriate texture coordinates

```
glBegin(GL_QUADS);
glTexCoord2f(0.0, 0.0); glVertex3f(0.0, 0.0);
glTexCoord2f(1.0, 0.0); glVertex3f(1.0, 0.0);
glTexCoord2f(1.0, 1.0); glVertex3f(1.0, 1.0);
glTexCoord2f(0.0, 1.0); glVertex3f(0.0, 1.0);
glEnd();
```

Post-processing Effects

- Texture coordinate notes:
  - If a texture rectangle is used, texture coordinates will be $[0, w\text{-}1] \times [0, h\text{-}1]$ instead of $[0, 1] \times [0, 1]$
  - If neighbor texels will need to be accessed, the texel size $[1 / (w\text{-}1), 1 / (h\text{-}1)]$ must be supplied as a uniform
    - Not needed for texture rectangles! Texel size is always 1!
  - To access many neighbors, pre-calculate some in coordinates in vertex shader

```
gl_TexCoord[0] = gl_MultiTexCoord0;
gl_TexCoord[1] = gl_MultiTexCoord0
                 + vec4(ts.x, 0.0, 0.0, 0.0);
gl_TexCoord[2] = gl_MultiTexCoord0
                 + vec4(0.0, ts.y, 0.0, 0.0);
gl_TexCoord[3] = ...
```

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

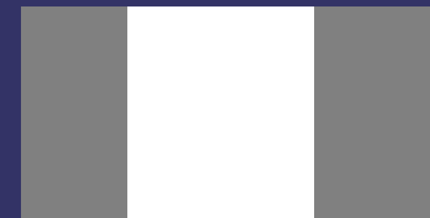
Filter Kernels

- 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

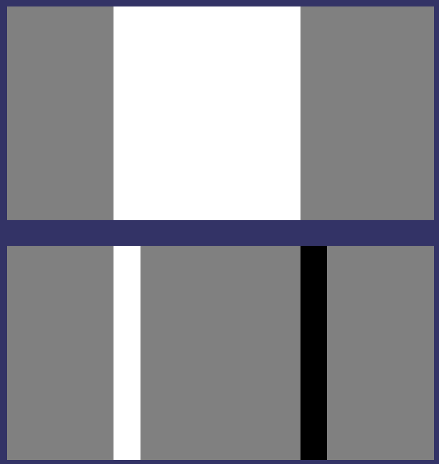
- Edge detection



Filter Kernels

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

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



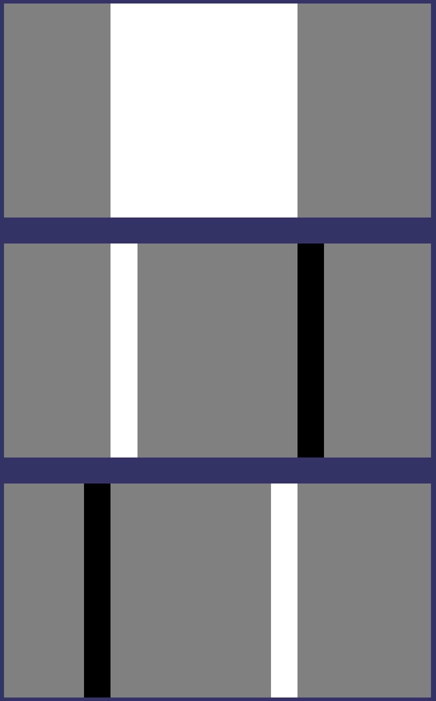
Filter Kernels

- 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 left neighbor

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



Filter Kernels

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

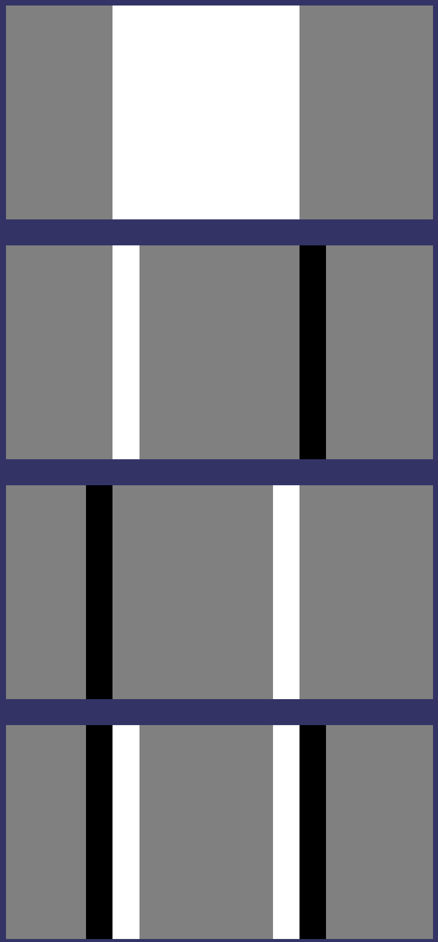
    $$p(x, y) - p(x-1, y)$$

  - Take the difference of each pixel and its left 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

- Rewrite as a kernel

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

Filter Kernels

- 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

- 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

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

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

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 $2n$
    - $2n \ll n^2$

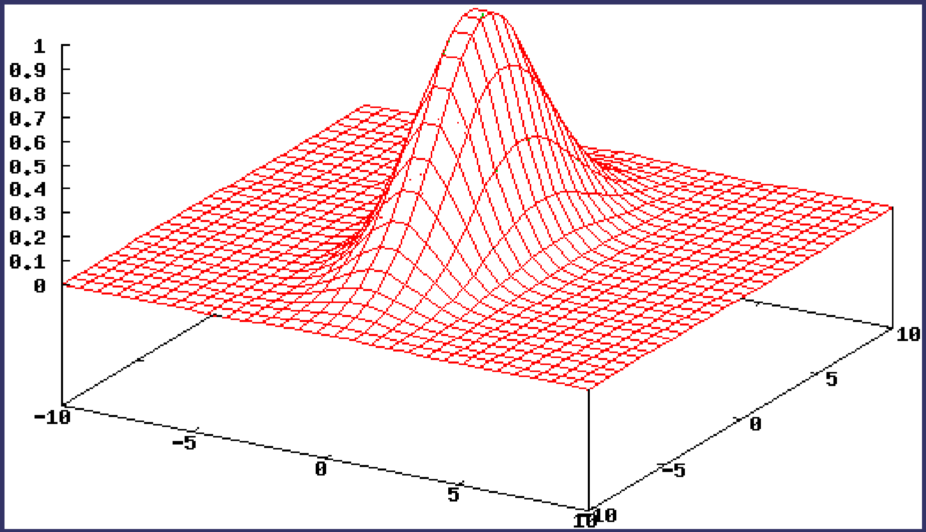
Separable Filter Kernels

- 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 $2n$
    - $2n \ll n^2$
- 2D kernel is calculated as the product of the 1D kernels

$$k(x, y) = k_x(x) \times k_y(y)$$

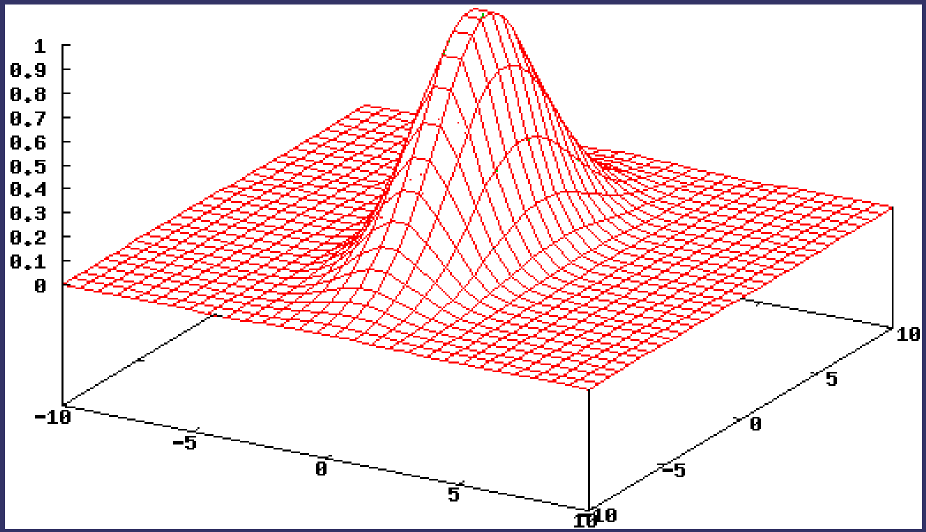
Separable Filter Kernels

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

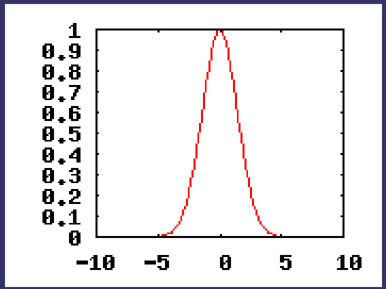


Separable Filter Kernels

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

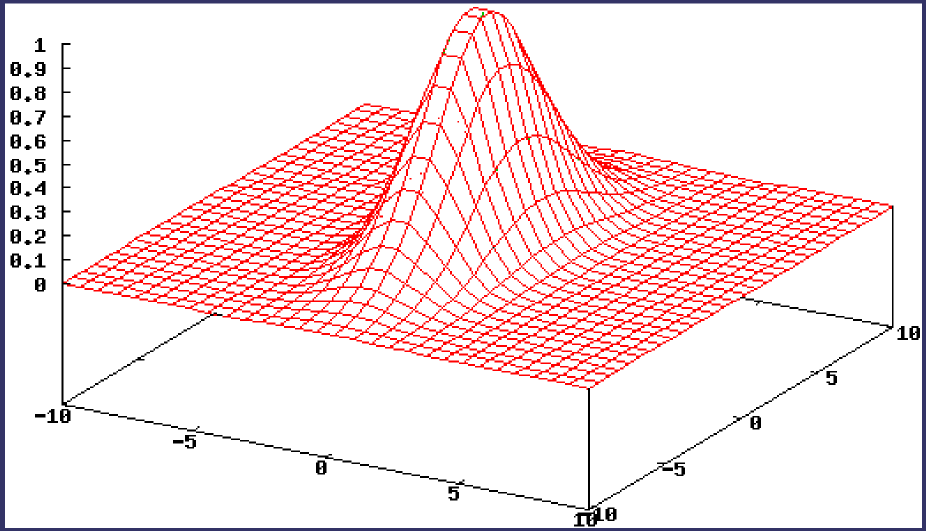


  - Product of a Gaussian along the X-axis

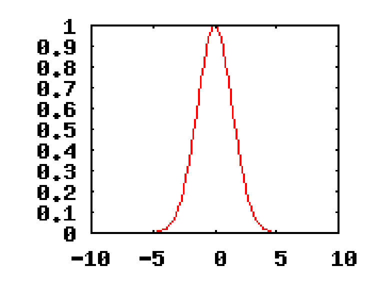


Separable Filter Kernels

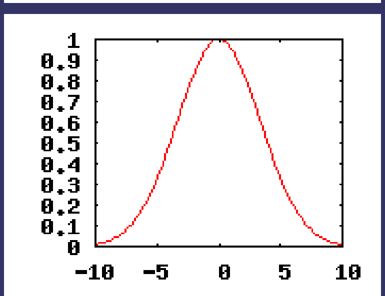
- The 2D Gaussian filter is *the classic* separable filter



  – 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

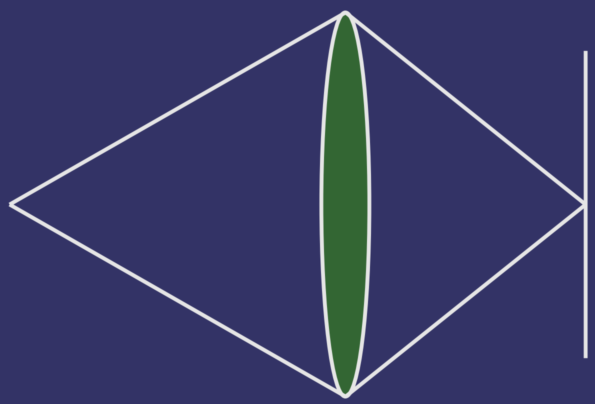
References

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

Break

Depth-of-field

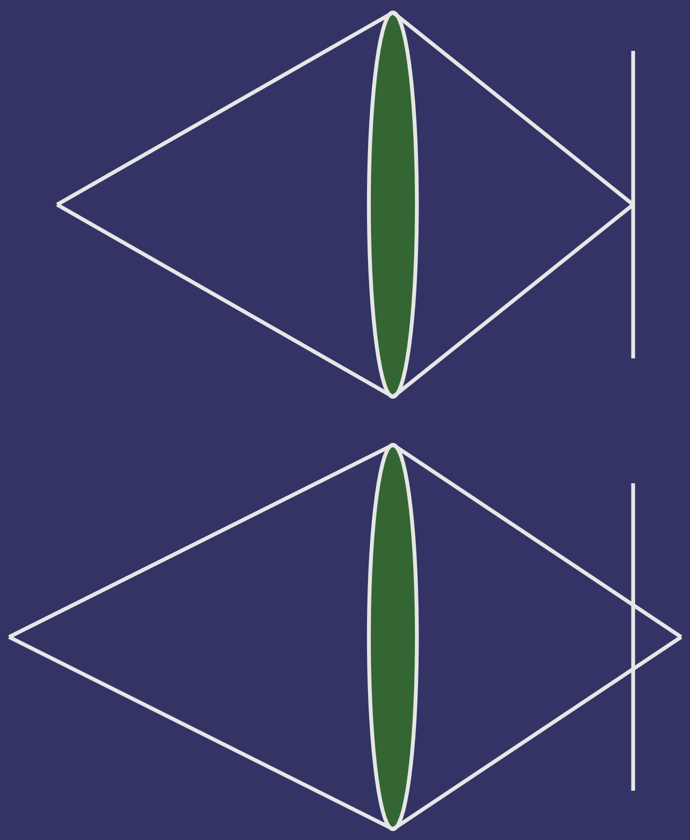
- A point of light focused through a lens becomes a point on object plane



11-March-2008

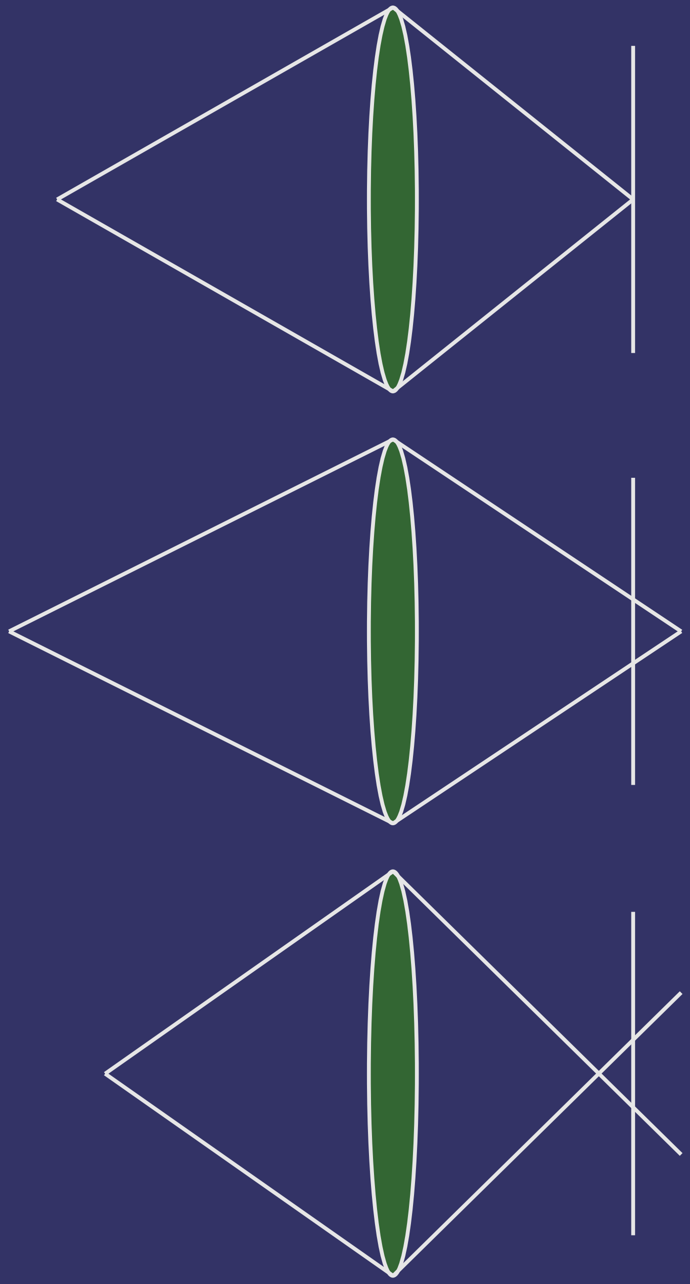
Depth-of-field

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



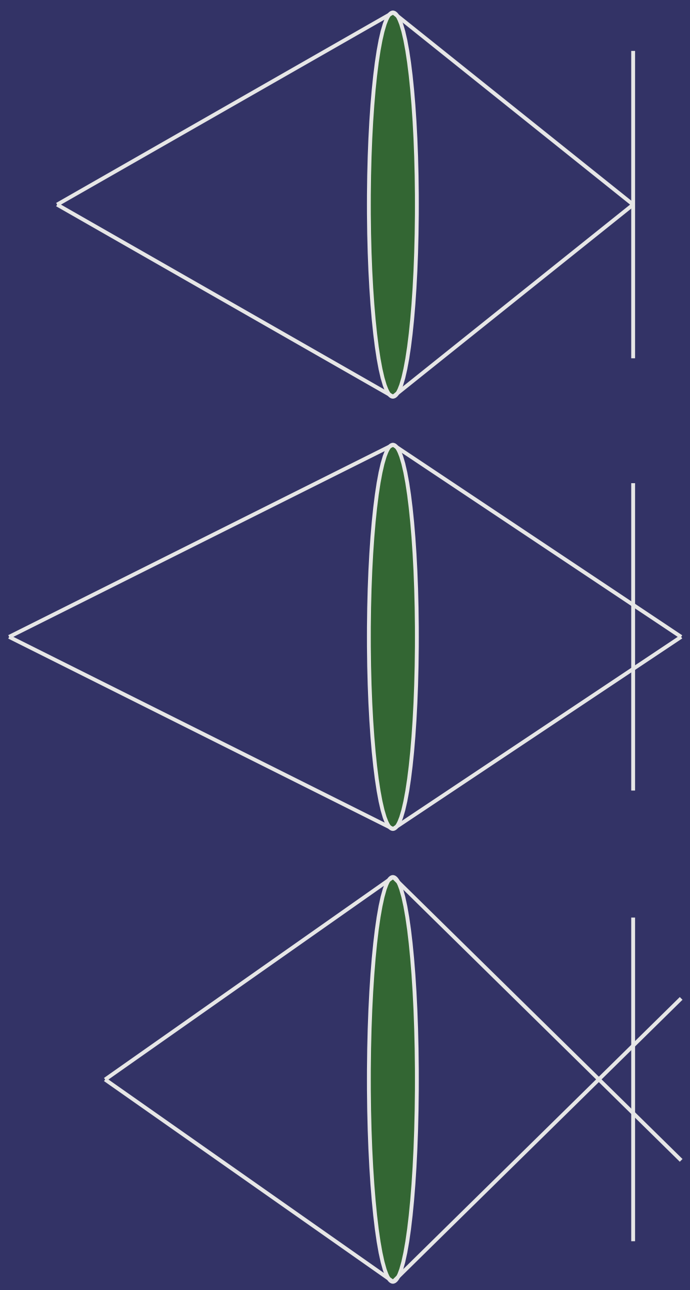
Depth-of-field

- A point of light focused through a lens becomes a point on 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



Depth-of-field

- A point of light focused through a lens becomes a point on 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
    - In a game, the player may want to focus on foreground and background objects in rapid succession. Until we can track where the player is looking on the screen, 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
    - In a game, the player may want to focus on foreground and background objects in rapid succession. Until we can track where the player is looking on the screen, the only way this works is to have everything in focus.
  - For non-interactive sequences, DoF can be a *very* powerful tool!
    - Film makers use this all the time to draw the audience's attention to certain things
    - Note the use of DoF in *Citizen Kane*

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

- 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
- Problem with this approach?

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
- 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://ati.amd.com/developer/shaderx/

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/DepthOfField/

Next week...

- Projects due at the start of class
- Oh yeah...the final!

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM 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.