VGP352 – Week 8

- Agenda:
  - High Dynamic Range Imaging (HDR)

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

- A lot of hardware supports a `RGB9E5` mode
  - Hardware that can texture from it *should* be able to render to it too
    - `glCheckFramebufferStatus` will return `GL_FRAMEBUFFER_UNSUPPORTED` if it can't
  - Internal format is `GL_RGB9_E5`
    - 9-bits for each mantissa, 5-bits for exponent
      - Matches the bit partitions for 16-bit float
    - Requires OpenGL 3.0 or `GL_EXT_texture_shared_exponent`

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 possible but 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

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

  - $l_{min\ white}$ 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...

- Bilateral filtering
- Depth peeling
- More SSAO
  - Horizon Split AO
  - Multi-Layer Dual-Resolution SSAO
- Read:

Cass Everitt, “Interactive order-independent transparency”, Technical report, NVIDIA Corporation, 2001. http://developer.nvidia.com/object/Interactive_Order_Transparency.html

Tobias Ritschel, Thorsten Grosch, Hans-Peter Seidel. Approximating Dynamic Global Illumination in Screen Space. Proceedings ACM SIGRAPH Symposium on Interactive 3D Graphics and Games, Boston, MA, February 27—March 1, 2009. http://www.mpi-inf.mpg.de/~ritschel/SSDO/

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.