CS 380 - GPU and GPGPU Programming
Lecture 13: GPU Texturing 2

Markus Hadwiger, KAUST
Reading Assignment #6 (until Mar. 17)

Read (required):

• Programming Massively Parallel Processors book, Chapter 4 (*CUDA Threads*)

• GLSL book: material needed for programming assignment #3
Next Lecture Dates

Lecture 11: Mar. 10 16:00

Lecture 12: Mar. 11 16:00, seminar room GMSV

Lecture 13: Mar. 12 16:00, seminar room GMSV

Lecture 14: Mar. 17 16:00
(Peter Rautek)

Lecture 15: Mar. 24 16:00

Lecture 16: Mar. 27 16:00
GPU Texturing

Rage / id Tech 5 (id Software)
Texture Anti-Aliasing: MIP Mapping

- MIP Mapping ("Multum In Parvo")
  - Texture size is reduced by factors of 2 (downsampling = "many things in a small place")
  - Simple (4 pixel average) and memory efficient
  - Last image is only ONE texel
Texture Anti-Aliasing: MIP Mapping

- MIP Mapping Algorithm
  - \( D := \log_2(\max(d_1, d_2)) \)
  - \( T_0 := \text{value from texture} \quad D_0 = \text{trunc} (D) \)
  - Use \textit{bilinear interpolation}

![Diagram showing MIP Mapping algorithm](image.png)
Texture Anti-Aliasing: MIP Mapping

- Trilinear interpolation:
  - $T_1 := \text{value from texture } D_1 = D_0 + 1$ (bilin.interpolation)
  - Pixel value := $(D_1 - D) \cdot T_0 + (D - D_0) \cdot T_1$
    - Linear interpolation between successive MIP Maps
  - Avoids "Mip banding" (but doubles texture lookups)
Texture Anti-Aliasing: MIP Mapping

- Other example for bilinear vs. trilinear filtering
Anti-Aliasing: Anisotropic Filtering

- Anisotropic filtering
  - View-dependent filter kernel
  - Implementation: *summed area table*, "RIP Mapping", *footprint assembly*, *elliptical weighted average* (EWA)

Texture space
Anti-Aliasing: Anisotropic Filtering

Example
Texture Anti-aliasing

- Basically, everything done in hardware
  - `gluBuild2DMipmaps()` generates MIPmaps
- Set parameters in `glTexParameter()`
  - `GL_LINEAR_MIPMAP_NEAREST`
  - `GL_TEXTURE_MAG_FILTER`
- Anisotropic filtering is an extension:
  - `GL_EXT_texture_filter_anisotropic`
- Number of samples can be varied (4x, 8x, 16x)
  - Vendor specific support and extensions
Texture Coordinates

- Specified manually (glMultiTexCoord())
- Using classical OpenGL texture coordinate generation
  - Linear: from object or eye space vertex coords
  - Special texturing modes (env-maps)
  - Can be further modified with texture matrix
    - E.g., to add texture animation
  - Can use 3rd or 4th texture coordinate for projective texturing!
- Shader allows complex texture lookups!
Texture Mapping

2D (3D) Texture Space
| Texture Transformation

2D Object Parameters
| Parameterization

3D Object Space
| Model Transformation

3D World Space
| Viewing Transformation

3D Camera Space
| Projection

2D Image Space

Kurt Akeley, Pat Hanrahan
Texture Mapping Polygons

Forward transformation: linear projective map

\[
\begin{bmatrix}
x \\
y \\
w
\end{bmatrix} = \begin{bmatrix}
a & b & c \\
d & e & f \\
g & h & i
\end{bmatrix} \begin{bmatrix}
s \\
t \\
r
\end{bmatrix}
\]

Backward transformation: linear projective map

\[
\begin{bmatrix}
s \\
t \\
r
\end{bmatrix} = \begin{bmatrix}
a & b & c \\
d & e & f \\
g & h & i
\end{bmatrix}^{-1} \begin{bmatrix}
x \\
y \\
w
\end{bmatrix}
\]

Kurt Akeley, Pat Hanrahan
Incorrect attribute interpolation

Linear interpolation

$A' \neq A$!

Kurt Akeley, Pat Hanrahan
Linear interpolation

Compute intermediate attribute value

- Along a line: \( A = aA_1 + bA_2, \quad a+b=1 \)
- On a plane: \( A = aA_1 + bA_2 + cA_3, \quad a+b+c=1 \)

Only projected values interpolate linearly in screen space (straight lines project to straight lines)

- \( x \) and \( y \) are projected (divided by \( w \))
- Attribute values are not naturally projected

Choice for attribute interpolation in screen space

- Interpolate unprojected values
  - Cheap and easy to do, but gives wrong values
  - Sometimes OK for color, but
  - Never acceptable for texture coordinates
- Do it right

Kurt Akeley, Pat Hanrahan
Linear Perspective

Correct Linear Perspective

Incorrect Perspective

Linear Interpolation, Bad

Perspective Interpolation, Good

Kurt Akeley, Pat Hanrahan
Perspective Texture Mapping

linear interpolation in object space

\[
\frac{ax_1 + bx_2}{aw_1 + bw_2} \neq a \frac{x_1}{w_1} + b \frac{x_2}{w_2}
\]

linear interpolation in screen space

\[
a = b = 0.5
\]
Early Perspective Texture Mapping in Games

DOOM (id Software, 1993)
Early Perspective Texture Mapping in Games

Quake (id Software, 1996)
Perspective-correct linear interpolation

Only projected values interpolate correctly, so project $A$

- Linearly interpolate $A_1/w_1$ and $A_2/w_2$
- Also interpolate $1/w_1$ and $1/w_2$
  - These also interpolate linearly in screen space

Divide interpolants at each sample point to recover $A$

- $(A/w) / (1/w) = A$
- Division is expensive (more than add or multiply), so
  - Recover $w$ for the sample point (reciprocate), and
  - Multiply each projected attribute by $w$

Barycentric triangle parameterization:

$$A = \frac{aA_1/w_1 + bA_2/w_2 + cA_3/w_3}{a/w_1 + b/w_2 + c/w_3}$$

$a + b + c = 1$

Kurt Akeley, Pat Hanrahan
Perspective Texture Mapping

- Solution: interpolate \((s/w, t/w, 1/w)\)
- \((s/w) / (1/w) = s\) etc. at every fragment

Heckbert and Moreton
Perspective-Correct Interpolation Recipe

\[ r_i(x, y) = \frac{r_i(x, y) / w(x, y)}{1/w(x, y)} \]

1. Associate a record containing the \( n \) parameters of interest \((r_1, r_2, \cdots, r_n)\) with each vertex of the polygon.

2. For each vertex, transform object space coordinates to homogeneous screen space using \(4 \times 4\) object to screen matrix, yielding the values \((xw, yw, zw, w)\).

3. Clip the polygon against plane equations for each of the six sides of the viewing frustum, linearly interpolating all the parameters when new vertices are created.

4. At each vertex, divide the homogeneous screen coordinates, the parameters \(r_i\), and the number 1 by \(w\) to construct the variable list \((x, y, z, s_1, s_2, \cdots, s_{n+1})\), where \(s_i = r_i/w\) for \(i \leq n\), \(s_{n+1} = 1/w\).

5. Scan convert in screen space by linear interpolation of all parameters, at each pixel computing \(r_i = s_i/s_{n+1}\) for each of the \(n\) parameters; use these values for shading.

Heckbert and Moreton
Projective Texture Mapping

- Want to simulate a beamer
  - ... or a flashlight, or a slide projector
- Precursor to shadows
- Interesting mathematics:
  2 perspective projections involved!
- Easy to program!
Projective Texture Mapping
What about **homogeneous** texture coords?

Need to do perspective divide also for projector!

- \((s, t, q) \rightarrow (s/q, t/q)\) for every fragment

How does OpenGL do that?

- Needs to be perspective correct as well!
- Trick: interpolate \((s/w, t/w, r/w, q/w)\)
- \((s/w) / (q/w) = s/q\) etc. at every fragment

Remember: \(s, t, r, q\) are equivalent to \(x, y, z, w\) in projector space! \(\rightarrow r/q =\) projector depth!
Multitexturing

- Apply multiple textures in one pass
- *Integral* part of programmable shading
  - e.g. diffuse texture map + gloss map
  - e.g. diffuse texture map + light map
- Performance issues
  - How many textures are free?
  - How many are available
Example: Light Mapping

- Used in virtually every commercial game
- Precalculate diffuse lighting on static objects
  - Only low resolution necessary
  - Diffuse lighting is view independent!
- Advantages:
  - No runtime lighting necessary
    - VERY fast!
  - Can take global effects (shadows, color bleeds) into account
Light Mapping

Original LM texels  Bilinear Filtering
Light Mapping Issues

- Why premultiplication is bad…

Full Size Texture (with Lightmap)  Tiled Surface Texture plus Lightmap

→ use tileable surface textures and low resolution lightmaps
LightMapping

Original scene

Light-mapped
Example: Light Mapping

- Precomputation based on non-realtime methods
  - Radiosity
  - Ray tracing
    - Monte Carlo Integration
    - Path tracing
    - Photon mapping
Light Mapping

Lightmap

mapped
Light Mapping

Original scene

Light-mapped
Ambient Occlusion

- Special case of light mapping
- Cosine-weighted visibility to environment modulates intensity:

\[ A_p = \frac{1}{\pi} \int_{\Omega} V_{p,\omega}(N \cdot \omega) \, d\omega \]

- Darker where more occluded
- Option: “per object” lightmap
  - Allows to move object
Ambient Occlusion

Model/Texture: Rendermonkey
The OpenGL Framebuffer Object Extension

Simon Green
NVIDIA Corporation
Why Render To Texture?

- Allows results of rendering to framebuffer to be directly read as texture
- Better performance
  - avoids copy from framebuffer to texture (glCopyTexImage2D)
  - uses less memory – only one copy of image
  - but driver may sometimes have to do copy internally
    - some hardware has separate texture and FB memory
    - different internal representations
- Applications
  - dynamic textures – procedurals, reflections
  - multi-pass techniques – anti-aliasing, motion blur, depth of field
  - image processing effects (blurs etc.)
  - GPGPU – provides feedback loop
**Framebuffer Object Advantages**

- **Only requires a single OpenGL context**
  - switching between framebuffers is faster than switching between pBuffers \((\text{wglMakeCurrent})\)

- **No need for complicated pixel format selection**
  - format of framebuffer is determined by texture or renderbuffer format
  - puts burden of finding compatible formats on developer

- **More similar to Direct3D render target model**
  - makes porting code easier

- **Renderbuffer images and texture images can be shared among framebuffers**
  - e.g. share depth buffers between color targets
  - saves memory
**EXT_framebuffer_object**

- OpenGL framebuffer is a collection of logical buffers
  - color, depth, stencil, accumulation
- Framebuffer object extension provides a new mechanism for rendering to destinations other than those provided by the window system
  - window system independent
- Destinations known as “framebuffer-attachable images”. Can be:
  - off-screen buffers (Renderbuffers)
  - textures
Framebuffer Object Architecture

Framebuffer object

- Color attachment 0
- ... (omitted)
- Color attachment n
- Depth attachment
- Stencil attachment
- Other state

Texture objects

- Texture image

Renderbuffer objects

- Renderbuffer image

NVIDIA

GDC 2005

San Francisco, CA
Terminology

- Renderbuffer image - 2D array of pixels. Part of a renderbuffer object.
- Framebuffer-attachable image - 2D array of pixels that can be attached to a framebuffer. Texture images and renderbuffer images are examples.
- Attachment point - State that references a framebuffer-attachable image. One each for color, depth and stencil buffer of a framebuffer.
- Attach - the act of connecting one object to another. Similar to “bind”.
- Framebuffer attachment completeness
- Framebuffer completeness
Framebuffers and Renderbuffers

- Introduces two new OpenGL objects:
  - “Framebuffer” (FBO)
    - collection of framebuffer-attachable images (e.g. color, depth, stencil)
    - plus state defining where output of GL rendering is directed
    - equivalent to window system “drawable”
  - “Renderbuffer” (RB)
    - contains a simple 2D image
      - no mipmaps, cubemap faces etc.
    - stores pixel data resulting from rendering
    - cannot be used as textures
Framebuffer Objects

- When a framebuffer object is bound its attached images are the source and destination for fragment operations
  - Color and depth textures
    - Supports multiple color attachments for MRT
  - Color, depth or stencil renderbuffers
Framebuffer Object API

```c
void GenFramebuffersEXT(sizei n, uint *framebuffers);
void DeleteFramebuffersEXT(sizei n,
                            const uint *framebuffers);

boolean IsFramebufferEXT(uint framebuffer);

void BindFramebufferEXT(enum target, uint framebuffer);

enum CheckFramebufferStatusEXT(enum target);
```
void FramebufferTexture1DEXT(enum target, enum attachment, enum textarget, uint texture, int level);
void FramebufferTexture2DEXT(enum target, enum attachment, enum textarget, uint texture, int level);
void FramebufferTexture3DEXT(enum target, enum attachment, enum textarget, uint texture, int level, int zoffset);

void FramebufferRenderbufferEXT(enum target, enum attachment, enum renderbuffertarget, uint renderbuffer);

void GetFramebufferAttachmentParameterivEXT(enum target, enum attachment, enum pname, int *params);

void GenerateMipmapEXT(enum target);
Managing FBOs and Renderbuffers

- Creating and destroying FBOs (and Renderbuffers) is easy - similar to texture objects
  
  void GenFramebuffersEXT(sizei n, uint *framebuffers);
  void DeleteFramebuffersEXT(sizei n, const uint *framebuffers);
  void BindFramebufferEXT(enum target, uint framebuffer);

- Can also check if a given identifier is a framebuffer object (rarely used)
  
  bool IsFramebufferEXT(uint framebuffer);
Binding an FBO

```c
void BindFramebufferEXT(enum target, uint framebuffer);
```

- **Makes given FBO current**
  - all GL operations occur on attached images
- **target** must be FRAMEBUFFER_EXT
- **framebuffer** is FBO identifier
  - if `framebuffer`==0, GL operations operate on window-system provided framebuffer (i.e. the window). This is the default state.
- Set of state that can change on a framebuffer bind:
  - AUX_BUFFERS, MAX_DRAW_BUFFERS, STEREO, SAMPLES, X_BITS, DOUBLE_BUFFER and a few more
void FramebufferTexture2DEXT(enum target, enum attachment, enum textarget, uint texture, int level);

- Attaches image from a texture object to one of the logical buffers of the currently bound framebuffer
- **target** must be FRAMEBUFFER_EXT
- **attachment** is one of:
  - COLOR_ATTACHMENT0_EXT ... COLOR_ATTACHMENTn_EXT,
    DEPTH_ATTACHMENT_EXT, STENCIL_ATTACHMENT_EXT
- **textarget** must be one of:
  - TEXTURE_2D, TEXTURE_RECTANGLE,
    TEXTURE_CUBE_MAP_POSITIVE_X etc.
- **level** is the mipmap level of the texture to attach
- **texture** is the texture object to attach
  - if texture==0, the image is detached from the framebuffer
- Other texture dimensions are similar
  - for 3D textures, z-offset specifies slice
Renderbuffer API

void GenRenderbuffersEXT(sizei n, uint *renderbuffers);
void DeleteRenderbuffersEXT(sizei n,
   const uint *renderbuffers);
boolean IsRenderbufferEXT(uint renderbuffer);

void BindRenderbufferEXT(enum target, uint renderbuffer);

void RenderbufferStorageEXT(enum target,
   enum internalformat, sizei width, sizei height);
void GetRenderbufferParameterivEXT(enum target,
   enum pname, int* params);
void RenderbufferStorageEXT (enum target,
    enum internalformat,sizei width,sizei height);

- Defines format and dimensions of a Renderbuffer
  - similar to TexImage call, but without image data
  - can read and write data using Read/DrawPixels etc.
- `target` must be `RENDERBUFFER_EXT`
- `internalformat` can be normal texture format (e.g. GL_RGBA8, GL_DEPTH_COMPONENT24)
  or:
  - STENCIL_INDEX1_EXT
  - STENCIL_INDEX4_EXT
  - STENCIL_INDEX8_EXT
  - STENCIL_INDEX16_EXT
Attaching Renderbuffers to a Framebuffer

```c
void FramebufferRenderbufferEXT(enum target,  
    enum attachment, enum renderbuffertarget,  
    uint renderbuffer);
```

- Attaches given renderbuffer as one of the logical buffers  
  of the currently bound framebuffer  
- `target` must be FRAMEBUFFER_EXT  
- `attachment` is one of:  
  - `COLOR_ATTACHMENT0_EXT` ...  
    `COLOR_ATTACHMENTn_EXT`  
    - Maximum number of color attachments implementation  
      dependent - query `MAX_COLOR_ATTACHMENTn_EXT`  
  - `DEPTH_ATTACHMENT_EXT`  
  - `STENCIL_ATTACHMENT_EXT`  
- `renderbuffertarget` must be RENDERBUFFER_EXT  
- `renderbuffer` is a renderbuffer id
Generating Mipmaps

```c
void GenerateMipmapEXT(enum target);
```

- Automatically generates mipmaps for texture image attached to `target`
- Generates same images as `GL_SGIS_generate_mipmap` extension
  - filtering is undefined, most likely simple 2x2 box filter
- Implemented as new entry point for complicated reasons (see spec).
**Framebuffer Completeness**

- **Framebuffer** is complete if all attachments are consistent
  - Texture formats make sense for attachment points
    - i.e. don’t try and attach a depth texture to a color attachment
  - All attached images must have the same width and height
  - All images attached to `COLOR_ATTACHMENT0_EXT` - `COLOR_ATTACHMENTn_EXT` must have the same format

- If not complete, `glBegin` will generate error `INVALID_FRAMEBUFFER_OPERATION`
Checking Framebuffer Status

```c
enum CheckFramebufferStatusEXT(enum target);
```

- Should always be called after setting up FBOs
- Returns enum indicating why framebuffer is incomplete:
  - `FRAMEBUFFER_COMPLETE`
  - `FRAMEBUFFER_INCOMPLETE_ATTACHMENT`
  - `FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT`
  - `FRAMEBUFFER_INCOMPLETE_DUPLICATE_ATTACHMENT`
  - `FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT`
  - `FRAMEBUFFER_INCOMPLETE_FORMATS_EXT`
  - `FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT`
  - `FRAMEBUFFER_INCOMPLETE_READ_BUFFER_EXT`
  - `FRAMEBUFFER_UNSUPPORTED`
  - `FRAMEBUFFER_STATUS_ERROR`
- Completeness is implementation-dependent
  - if result is “FRAMEBUFFER_UNSUPPORTED”, application should try different format combinations until one succeeds
FBO Performance Tips

- Don’t create and destroy FBOs every frame
- Try to avoid modifying textures used as rendering destinations using TexImage, CopyTexImage etc.
FBO Usage Scenarios

- FBO allows several ways of switching between rendering destinations
- In order of increasing performance:
  - Multiple FBOs
    - create a separate FBO for each texture you want to render to
    - switch using `BindFramebuffer()`
      - can be 2x faster than `wglMakeCurrent()` in beta NVIDIA drivers
  - Single FBO, multiple texture attachments
    - textures should have same format and dimensions
    - use `FramebufferTexture()` to switch between textures
  - Single FBO, multiple texture attachments
    - attach textures to different color attachments
    - use `glDrawBuffer()` to switch rendering to different color attachments
#define CHECK_FRAMEBUFFER_STATUS() \ 
{ \ 
  GLenum status; \ 
  status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT); \ 
  switch(status) { \ 
  case GL_FRAMEBUFFER_COMPLETE_EXT: \ 
    break; \ 
  case GL_FRAMEBUFFER_UNSUPPORTED_EXT: \ 
    /* choose different formats */ \ 
    break; \ 
  default: \ 
    /* programming error; will fail on all hardware */ \ 
    assert(0); \ 
  } \ 
}
Simple FBO Example

GLuint fb, depth_rb, tex;

// create objects
glGenFramebuffersEXT(1, &fb);        // frame buffer
glGenRenderbuffersEXT(1, &depth_rb); // render buffer
glGenTextures(1, &tex);              // texture
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

// initialize texture
glBindTexture(GL_TEXTURE_2D, tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, NULL);
// (set texture parameters here)

// attach texture to framebuffer color buffer
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, tex, 0);
Simple FBO Example

// initialize depth renderbuffer
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depth_rb);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
    GL_DEPTH_COMPONENT24, width, height);

// attach renderbuffer to framebuffer depth buffer
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
    GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT,
    depth_rb);
CHECK_FRAMEBUFFER_STATUS();
...

// render to the FBO
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
// (draw something here, rendering to texture)

// render to the window, using the texture
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
glBindTexture(GL_TEXTURE_2D, tex);
Thank you.