CS 380 - GPU and GPGPU Programming
Lecture 18: GPU Texturing 3

Markus Hadwiger, KAUST
Reading Assignments #10 (until Nov 4)

Read (required):

• **Brook for GPUs: Stream Computing on Graphics Hardware**  
  Ian Buck et al., SIGGRAPH 2004  

Read (optional):

• **The Imagine Stream Processor**  
  Ujval Kapasi et al.; IEEE ICCD 2002  

• **Merrimac: Supercomputing with Streams**  
  Bill Dally et al.; SC 2003  
  [https://dl.acm.org/citation.cfm?doid=1048935.1050187](https://dl.acm.org/citation.cfm?doid=1048935.1050187)
Programming Assignments: Schedule (tentative)

Assignment #1:
• Querying the GPU (OpenGL/GLSL and CUDA) due Sep 4

Assignment #2:
• Phong shading and procedural texturing (GLSL) due Sep 24

Assignment #3:
• Image Processing with GLSL due Oct 7

Assignment #4:
• Image Processing with CUDA
• Convolutional layers with CUDA due Oct 21

Assignment #5:
• Linear Algebra (CUDA) due Nov 11
Homogeneous Coordinates

Projective Geometry: projective spaces $\mathbb{RP}^n$: projective line $\mathbb{RP}^1$, projective plane $\mathbb{RP}^2$, ...

Examples of usage

- Translation
- Projection

\[
\begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 \\
0 & \frac{2}{\text{top} - \text{bottom}} & 0 \\
0 & 0 & \frac{-2}{\text{far} - \text{near}} \\
0 & 0 & 1
\end{bmatrix}
\]

orthographic

\[
\begin{bmatrix}
\frac{z_{\text{near}}}{\text{width}/2} & 0.0 & \frac{\text{left} + \text{right}}{\text{width}/2} & 0.0 \\
0.0 & \frac{z_{\text{near}}}{\text{height}/2} & \frac{\text{top} + \text{bottom}}{\text{height}/2} & 0.0 \\
0.0 & 0.0 & \frac{z_{\text{far}} + z_{\text{near}}}{z_{\text{far}} - z_{\text{near}}} & \frac{2z_{\text{far}}z_{\text{near}}}{z_{\text{far}} - z_{\text{near}}} \\
0.0 & 0.0 & -1.0 & 0.0
\end{bmatrix}
\]
perspective

Markus Hadwiger, KAUST
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
Linear Perspective

Correct Linear Perspective

Incorrect Perspective

Linear Interpolation, Bad

Perspective Interpolation, Good

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' ≠ 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
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

Ultima Underworld (Looking Glass, 1992)

Use key on what?
The key unlocks the lock.
You see a hostile stone golem.
You see a stone wall.
You see a hostile ghoul.
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:

$$A = \frac{aA_1/w_1 + bA_2/w_2 + cA_3/w_3}{a/w_1 + b/w_2 + c/w_3} \quad a + b + c = 1$$

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
Magnification (Bi-linear Filtering Example)

Original image

Nearest neighbor

Bi-linear filtering

Vienna University of Technology
Nearest-Neighbor vs. Bi-Linear Interpolation

nearest-neighbor

bi-linear

Bilinear patch (courtesy J. Han)
Consider area between 2x2 adjacent samples (e.g., pixel centers):

Given any (fractional) position

\[
\alpha_1 := x_1 - \lfloor x_1 \rfloor \quad \alpha_1 \in [0.0, 1.0)
\]
\[
\alpha_2 := x_2 - \lfloor x_2 \rfloor \quad \alpha_2 \in [0.0, 1.0)
\]

and 2x2 sample values

\[
\begin{bmatrix}
v_{01} & v_{11} \\
v_{00} & v_{10}
\end{bmatrix}
\]

Compute: \( f(\alpha_1, \alpha_2) \)
Consider area between 2x2 adjacent samples (e.g., pixel centers):

Given any (fractional) position

\[ \alpha_1 := x_1 - \lfloor x_1 \rfloor \quad \alpha_1 \in [0.0, 1.0) \]
\[ \alpha_2 := x_2 - \lfloor x_2 \rfloor \quad \alpha_2 \in [0.0, 1.0) \]

and 2x2 sample values

\[
\begin{bmatrix}
  v_{01} & v_{11} \\
  v_{00} & v_{10}
\end{bmatrix}
\]

Compute: \( f(\alpha_1, \alpha_2) \)
Bi-Linear Interpolation

Weights in 2x2 format:

\[
\begin{bmatrix}
\alpha_2 \\
(1 - \alpha_2)
\end{bmatrix}
\begin{bmatrix}
(1 - \alpha_1) \\
\alpha_1
\end{bmatrix}
= 
\begin{bmatrix}
(1 - \alpha_1)\alpha_2 \\
(1 - \alpha_1)(1 - \alpha_2)
\end{bmatrix}
\begin{bmatrix}
\alpha_1 \\
(1 - \alpha_2)
\end{bmatrix}
\]

Interpolate function at (fractional) position \((\alpha_1, \alpha_2)\):

\[
f(\alpha_1, \alpha_2) = \begin{bmatrix}
\alpha_2 \\
(1 - \alpha_2)
\end{bmatrix}
\begin{bmatrix}
v_{01} \\
v_{00}
\end{bmatrix}
\begin{bmatrix}
v_{11} \\
v_{10}
\end{bmatrix}
\begin{bmatrix}
(1 - \alpha_1) \\
\alpha_1
\end{bmatrix}
\]
Interpolate function at (fractional) position \((\alpha_1, \alpha_2)\):

\[
f(\alpha_1, \alpha_2) = \begin{bmatrix} \alpha_2 & (1 - \alpha_2) \end{bmatrix} \begin{bmatrix} v_{01} & v_{11} \\ v_{00} & v_{10} \end{bmatrix} \begin{bmatrix} (1 - \alpha_1) \\ \alpha_1 \end{bmatrix}
\]

\[
= \begin{bmatrix} \alpha_2 & (1 - \alpha_2) \end{bmatrix} \begin{bmatrix} (1 - \alpha_1)v_{01} + \alpha_1 v_{11} \\ (1 - \alpha_1)v_{00} + \alpha_1 v_{10} \end{bmatrix}
\]

\[
= \begin{bmatrix} \alpha_2 v_{01} + (1 - \alpha_2)v_{00} & \alpha_2 v_{11} + (1 - \alpha_2)v_{10} \end{bmatrix} \begin{bmatrix} (1 - \alpha_1) \\ \alpha_1 \end{bmatrix}
\]
Interpolate function at (fractional) position \((\alpha_1, \alpha_2)\):

\[
f(\alpha_1, \alpha_2) = \begin{bmatrix} \alpha_2 & (1 - \alpha_2) \end{bmatrix} \begin{bmatrix} v_{01} & v_{11} \\
 v_{00} & v_{10} \end{bmatrix} \begin{bmatrix} (1 - \alpha_1) \\
 \alpha_1 \end{bmatrix}
\]

\[
= (1 - \alpha_1)(1 - \alpha_2)v_{00} + \alpha_1(1 - \alpha_2)v_{10} + (1 - \alpha_1)\alpha_2v_{01} + \alpha_1\alpha_2v_{11}
\]

\[
= v_{00} + \alpha_1(v_{10} - v_{00}) + \alpha_2(v_{01} - v_{00}) + \alpha_1\alpha_2(v_{00} + v_{11} - v_{10} - v_{01})
\]
REALLY IMPORTANT:
this is a different thing (for a different purpose) than the linear (or, in perspective, rational-linear) interpolation of texture coordinates!!
Texture Aliasing: Minification

Problem: One pixel in image space covers many texels
Texture Aliasing: Minification

- Caused by *undersampling*: texture information is lost

Texture space

Image space
A good pixel value is the weighted mean of the pixel area projected into texture space.

Texture space

Image space

Pixel

Vienna University of Technology
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 bilinear interpolation

```
\text{Bilinear interpolation}   \quad \text{Trilinear interpolation}
```
• Use the partial derivatives of texture coordinates with respect to screen space coordinates
• This is the Jacobian matrix
\[
\begin{pmatrix}
\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\
\frac{\partial v}{\partial x} & \frac{\partial v}{\partial y}
\end{pmatrix} =
\begin{pmatrix}
s_x & s_y \\
t_x & t_y
\end{pmatrix}
\]
• Area of parallelogram is the absolute value of the Jacobian determinant (the Jacobian)
MIP-Map Level Computation (OpenGL)

• OpenGL 4.6 core specification, pp. 251-264

\[ \lambda_{\text{base}}(x, y) = \log_2[\rho(x, y)] \]

\[ \rho = \max \left\{ \sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2} \right\} \]

• Does not use area of parallelogram but greater hypotenuse [Heckbert, 1983]

• Approximation without square-roots

\[ m_u = \max \left\{ \left| \frac{\partial u}{\partial x} \right|, \left| \frac{\partial u}{\partial y} \right| \right\} \quad m_v = \max \left\{ \left| \frac{\partial v}{\partial x} \right|, \left| \frac{\partial v}{\partial y} \right| \right\} \quad m_w = \max \left\{ \left| \frac{\partial w}{\partial x} \right|, \left| \frac{\partial w}{\partial y} \right| \right\} \]

\[ \max\{m_u, m_v, m_w\} \leq f(x, y) \leq m_u + m_v + m_w \]
• Level of detail value is fractional!
• Use fractional part to blend (lin.) between two adjacent mipmap levels
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)
Anisotropic Filtering: Footprint Assembly

- MIP-map Sample Points
- Line of Anisotropy
- Initial Texture Coordinate
- Filter Kernels
  - Trilinear
  - Anisotropic
Anti-Aliasing: Anisotropic Filtering

Example
Texture Anti-aliasing

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