Per-Face Texture Mapping for Realtime Rendering

“Realtime Ptex”
Quick Note

• For CUDA/Compute folks:
  – Ptex != PTX
Goals

- Render native Ptex datasets in real-time on commodity hardware
- Remove texture seams from textured models
- Remove expensive, manual model unwrap step from art pipeline
- Support arbitrary resolutions on a per-face basis
Video Recap

Stored as 8Kx8K
Rendered as 2Kx2K
Model Statistics

- ~800 FPS on GTX 460 with no optimization
- 278M of Color Texture Data
- 5812 Patches
General Steps

- Load Model
- Bucket and Sort
- Generate Mipmaps
- Fill Borders
- Pack Texture Arrays

Preprocess

- Reorder Index Buffer
- Pack Patch Constants

Draw Time

- Render

Colors:
- Red: Vertex and Index data
- Green: Patch Constant information
- Blue: Texel data
- Orange: Adjacency information
Load Model

- Load Model
- Bucket and Sort
- Generate Mipmaps
- Fill Borders
- Pack Texture Arrays

Preprocess

- Reorder Index Buffer
- Pack Patch Constants

Draw Time

- Render

Color Codes:
- Red: Vertex and Index data
- Green: Patch Constant information
- Blue: Texel data
- Orange: Adjacency information
Load Model

- **Vertex Data**
  - Any geometry arranged as a quad-based mesh
  - Example: Wavefront OBJ
- **Patch Texture**
  - Power-of-two texture images
- **Adjacency Information**
  - 4 Neighbors of each quad patch
- **Easily load with library available from** [http://ptex.us/](http://ptex.us/)
Load Model (cont’d)

• **Texel Data**
  
  — Per face, load the largest available mipmap level from the source files
  
  — In memory, place the loaded texel data into a memory buffer that has a fixed-size texel border region
  
  — The borders must be big enough for the largest filter kernel you will support (Border size = $\frac{1}{2}$ filter kernel size)
    
    • Bilinear (2x2 kernel): 1 pixel border
    
    • 16x Aniso (16x16 kernel): 8 pixel border
Load Model

VB: …
IB:
Bucket and Sort

Load Model

Bucket and Sort

Bucket and Sort

Generate Mipmaps

Fill Borders

Pack Texture Arrays

Preprocess

Draw Time

Render

Red: Vertex and Index data
Green: Patch Constant data
Blue: Texel data
Orange: Adjacency information
Bucket and Sort

• Bucket ptex surfaces into groups by Aspect Ratio
  – Each Aspect Ratio will be one bucket
  – 1:1 bucket, 2:1 bucket, 4:1 bucket, etc

• Then, within each bucket, sort by decreasing surface size and assign IDs.
  – This allows us to densely pack texture arrays, avoiding “empty” surfaces.
Reorder Index Buffer

Load Model → Bucket and Sort → Reorder Index Buffer → Preprocess

Preprocess → Pack Patch Constants

Pack Patch Constants → Generate Mipmaps

Generate Mipmaps → Fill Borders

Fill Borders → Pack Texture Arrays

Pack Texture Arrays → Render

Red: Vertex and Index data
Green: Patch Constant information
Blue: Texel data
Orange: Adjacency information
Reorder Index Buffer

• Original

IB:

IB(1:1):

• Post Sort, we have 2!

IB(1:1):

IB(4:1):
Generate Mipmaps

• Walk through surfaces, and generate a mipmap chain from native size to \((\text{MinFilterSize} \times \text{MinFilterSize})\) for each surface.
  
  — Bilinear stops at 2x2, 8xAniso stops at 8x8

• Ptex data files do not guarantee complete mipmap chains—although the library can generate all levels for you—with pre-multiplied alpha.

• Mipmap chains stop to allow for unique pinning values in the corners
Generate Mipmaps

- Done for every surface, but only inside the surface—the border is not touched.
Fill Borders

Load Model
Bucket and Sort
Reorder Index Buffer
Generate Mipmaps
Pack Patch Constants
Fill Borders
Pack Texture Arrays

Preprocess

Draw Time

Render

Red: Vertex and Index data
Green: Patch Constant information
Blue: Texel data
Orange: Adjacency information
Fill Borders

• Copy neighbor texels into border area of this surface’s mip level
  – Match source and destination number of pixels when possible
• Bordered textures are the heart of the logical realtime ptex solution
• Allows 1-2 texture lookups per ptex sample request
  – 1 if not performing tween-mip-level interpolation, 2 otherwise
Fill Borders
Fill Borders
Pack Texture Arrays

Load Model → Bucket and Sort → Reorder Index Buffer → Generate Mipmaps → Pack Patch Constants → Fill Borders → Pack Texture Arrays

Preprocess

Draw Time

Render

Red: Vertex and Index data
Green: Patch Constant information
Blue: Texel data
Orange: Adjacency information
Texture Arrays

• Like 3D / Volume Textures, except:
  – No filtering between 2D slices
  – Only X and Y decrease with mipmap level (Z doesn’t)
  – Z indexed by integer index, not [0,1]
    • E.g. (0.5, 0.5, 4) would be (0.5, 0.5) from the 5th slice

• API Support
  – Direct3D 10+: Texture2DArray
  – OpenGL 3.0+: GL_TEXTURE_2D_ARRAY
Pack Texture Arrays

- Copy all generated data into Texture2DArray
- Each Texture2DArray represents a single mipmap level
  - Texture2DArrays present a view of the data that is efficient for GPU layout
  - Logical Textures cut across the same page index of every Texture2DArray
Pack Texture Arrays

Logical Texture Layout
### Pack Texture Arrays

#### GPU Layout

<table>
<thead>
<tr>
<th>Texture2DArray</th>
<th>gColor[0]</th>
<th>gColor[1]</th>
<th>gColor[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10x10x3</td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
</tr>
<tr>
<td>$(1+8+1) \times (1+8+1) \times 3$</td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
</tr>
<tr>
<td>6x6x4</td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
</tr>
<tr>
<td>$(1+4+1) \times (1+4+1) \times 4$</td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
</tr>
<tr>
<td>4x4x4</td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
</tr>
<tr>
<td>$(1+2+1) \times (1+2+1) \times 4$</td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
<td><img src="Image" alt="Texture" /></td>
</tr>
</tbody>
</table>
Pack Patch Constants

- Load Model
- Bucket and Sort
- Generate Mipmaps
- Fill Borders
- Pack Texture Arrays

Preprocess

Draw Time

Render

Red: Vertex and Index data
Green: Patch Constant information
Blue: Texel data
Orange: Adjacency information
• Each primitive has a “PatchInfo” struct:
  – TextureId – which array slice contains our data
  – TopMipLevel – the index of the top-most mipmap level for this texture
  – FlipUVs – whether or not to flip UVs, allows 1:2 and 2:1 to be grouped into same bucket
  – MaxMipLevels – Maximum mipmap level for each edge
Fill Borders

Load Model  →  Bucket and Sort  →  Generate Mipmaps  →  Fill Borders  →  Pack Texture Arrays

Reorder Index Buffer  →  Pack Patch Constants

Preprocess

Draw Time

Render

Red: Vertex and Index data
Green: Patch Constant information
Blue: Texel data
Orange: Adjacency information
• Brief Recap of D3D11 Pipe stages

New Direct3D 11 Stages

Programmable (Shader)
Fixed Function
In the Hull Shader

- Store pre-expansion PrimitiveID to output control points
- This is used everywhere to determine which set of Patch Constants are owned by the currently running thread (in Domain, Geometry or Pixel Shaders)
• **In the Domain Shader**
  - Vertices belonging to a quad meshes are evaluated with a domain location, which is (0,0)-(1,1) for each patch
  - Use this value to store our UV location

New Direct3D 11 Stages

IA ➔ VS ➔ HS ➔ TS ➔ **DS** ➔ GS ➔ RAS ➔ PS ➔ OM
Texture lookups in Domain or Pixel Shader are replaced with a “ptex” sample function.

- Determines which logical texture to work from
- Compute mipmap level(s) to access
- Scale and bias computed (u,v) by mipmap size
- Lookup texels, return weighted average
Texture Lookup Shader Code

• Traditional (D3D11)

```c
return gTxDiffuse.Sample(gSampler, I.fTextureUV);
```

• Ptex

```c
return ptex( gTxDiffuse, gSampler, I.uPrimitiveId, I.fTextureUV );
```

Complex logic hidden in single function call
Questions?

- jmcdonald at nvidia dot com
- Brent dot Burley at disneyanimation dot com
- http://ptex.us/
- http://groups.google.com/group/ptex
- Or visit our studio session
  - Monday, 8 August @ 4:30 PM – 5:00 PM
  - The Studio / West Building, Ballroom A
Additional Considerations

- Filtering across edges with differing numbers of pixels
- Filtering across corners
Filtering across edges
Filtering across edges
Filtering across corners

- Corners with valence > 4 cannot be exactly matched with a bilerp (4 samples)
Filtering across corners

- The solution is a bit more involved.
  - First, walk the mesh and determine which corners are shared
  - For each shared group, determine the correct value for when we’re exactly at that corner. E.g. Simple Average.
  - Then, modify every mipmap level of every surface of that group s.t. the shared corner has the same value
  - When you’re in the corner, everyone will perform the same lookup—regardless of mipmap level—and continuity prevails
Filtering across corners

- For Realtime Ptex, we apply pinning to all corners, regardless of valence.
Questions redux?

- jmcdonald at nvidia dot com
- Brent dot Burley at disneyanimation dot com
- http://ptex.us/
- http://groups.google.com/group/ptex
- Or visit our studio session
  - Monday, 8 August @ 4:30 PM – 5:00 PM
  - The Studio / West Building, Ballroom A