12 Advanced Texturing

19/02/2015
Texture Mapping
Brief Aside: Rasterization
void drawline(float x1, float y1, float x2, float y2) {
    float x=x1, y=y1, t;
    while(x<x2) {
        t = (x - x1) / (x2 - x1);
        y = (1-t) * y1 + t * y2;
        x = x + 1;
        point ( x, y);
    }
}
void drawline(float x1, float y1, float x2, float y2)
{
    float x=x1, y=y1, t;
    while(x<x2)
    {
        t = (x - x1) / (x2 - x1);
        y = (1-t)*y1 + t*y2;
        x = x + 1;
        point (x, y);
    }
}

N.B. this is only a partial solution;
- assumes that x1 < x2, y1 < y2.
- also leaves gaps if gradient is too steep

For details: look up DDA or Bresenham’s line drawing algorithm
Rasterizing Polygons

float[] v = { 0, 0, 100, 0, 100, 100, 0, 100 };

drawline(v[0], v[1], v[2], v[3]);
drawline(v[2], v[3], v[4], v[5]);
drawline(v[4], v[5], v[6], v[7]);
drawline(v[6], v[7], v[0], v[1]);

0,0

100,10
0
Rasterizing Filled Polygons (1)

- Polygon Scan-conversion:
  - First scan convert all edges: increment along y and don’t worry about gaps
  - For each scan line, “fill” between start and end pixel
1. RASTERIZE edges.
2. Interpolate across x values along each scan line
Rasterizing Filled Polygons (4)

\[ ta = 1 \]

Repeat until we get to a vertex

\[ tb = ? \]
Apply same procedure as before with one new edge and the remainder of the longer of the first two edges.
while(y<y2)
{
    ta = (y - y1) / (y2 - y1);
    tb = (y - y1) / (y3 - y1);

    xa = (1-ta) * x1 + ta * x2;
    xb = (1-tb) * x1 + tb * x3;

    y = y + 1;
    drawline(y, xa, y, xb);
}
Not only do we interpolate to find intermediate x values, BUT we also interpolate to find intermediate values of u and v.
Then we interpolate for uv values across the scan line
while(y<y2)
{
    ta = (y - y1) / (y2 - y1);
    tb = (y - y1) / (y3 - y1);

    xa = (1-ta) * x1 + ta * x2;
    xb = (1-tb) * x1 + tb * x3;

    ua = (1-ta) * u1 + ta * u2;
    ub = (1-tb) * u1 + ta * u3;
    va = (1-ta) * v1 + ta * v2;
    vb = (1-tb) * v1 + ta * v3;

    y = y + 1;
    drawTEXline(y,xa,ua,va,y,xb,ub, vb);
}

Image tex;
void drawTEXline(float x1, float y1, float ua, float va, float x2, float y2, float ub, float vb )
{
    float x=x1, y=y1, t;
    while(x<x2)
    {
        t = (x - x1) / (x2 - x1);
        y = (1-t) * x1 + t * x2;
        u = (1-t) * ua + t * ub;
        v = (1-t) * va + t * vb;
        x = x + 1;
        strokeColor(tex.get(u, v));
        point ( x, y);
    }
}

NB. This example is 2D. For 3D need to consider projection + transforms
Overview

- Texture Mapping
  - Aliasing
  - Filtering
  - Mip-Maps
  - Summed Area Table
  - Anisotropic Filtering
2D Texturing Issues

- 2D Texture made up of texels mapped on to 3D object
  - Texture object mapped on to 2d screen pixels
  - 2d – 2d warping: Sampling is not always uniform

- Texture **Minification**
  - Many pixels to few texels

- Texture **Magnification**
  - Few texels map to many pixels

- Can lead to **Aliasing**

- Sampling/filtering techniques need to account for this
Aliasing

- Leads to
  - Jaggies
  - Moire-patterns
  - Temporal aliasing:

- Nyquist law: max frequency displayable is half the sampling frequency
Discrete Sampling

- Take and example texture
Discrete Sampling

- Now suppose that the object moves away from us so that each square occupies 1 pixel.
Discrete Sampling

- Consider one row of pixels and texels.

```
texels

[] [] [] [] [] [] [] [] [] [] [] [] [] [] []
pixels
```
Discrete Sampling

- Using the nearest texel, the pixels will be colored alternately black and white.
Discrete Sampling

- Now suppose the surface moves a little further away so that nearly 2 texels cover one pixel.
Discrete Sampling

- Using the nearest texel, there will be long stretches of black and white pixels.
Discrete Sampling

- What will happen when the texels are exactly half the width of a pixel?
- What will happen when the texels are exactly one third the width of a pixel?
- Exactly one fourth?
Texture Minification
Nyquist Frequency for Textures
Texture Minification
Texture Minification

Pixel centres

Aliasing: high frequency looks like low frequency
Texture Minification

Even at close to same frequency the image is distorted
Moire Patterns

- Interference pattern created by *almost-synchronised* signal
Nyquist Frequency for Textures

- Need at least twice as many pixels per texel in the same area
- But then this leads to texture magnification!
- **Filtering** required.

- Two main techniques in OpenGL: linear and nearest-neighbour
Nearest-Neighbour Filtering

- Use colour of texel closest to the pixel center
- Fast
- Results in a large number of artifacts
  - texture 'blockiness' during magnification,
  - and aliasing and shimmering during minification.
Bi-linear filtering

- Get values of four neighbouring texels and linearly interpolate to find a blended value
- removes the blockiness seen during magnification
Texture Magnification

Nearest neighbour  Bi-linear filtering
Texture Magnification

Nearest neighbour  Bi-linear filtering
Minification

- Several texels mapped to single pixels
Texture Minification

Nearest neighbour

Mip-mapping
Texture Minification

Nearest neighbour

Mip-mapping

Summed area tables
Summed Area Table

pixel space

pixel's cell

texture space

bounding box

A

B

x_{ul}

x_{ur}

y_{ul}

y_{ur}
Mip-Mapping

- **mip = multum in parvo = “many things in a small place.”**
- Create level of detail simplifications of the entire texture.

- **Basic technique**
  - take 2x2 squares and average
  - Box filter (not great)

- **Some are better:** gaussian, Lanczos, Kaiser

If the original texture is 64 × 64, then we should create copies at the scales of 32 × 32, 16 × 16, 8 × 8, 4 × 4, 2 × 2, and 1 × 1: This is why graphics API’s prefer power of 2 textures
Mip-map storage

10-level Mip Map

Memory format of a mip map
Mipmap

Level 0: 64 x 64

Level 1: 32 x 32

Level 2: 16 x 16

Level 3: 8 x 8

Level 4: 4 x 4

Level 5: 2 x 2

Level 6: 1 x 1
Level 1 Mipmap – 32 × 32
Level 2 Mipmap – 16 × 16
Level 3 Mipmap – 8 × 8
Level 4 Mipmap – 4 × 4
Level 5 Mipmap – $2 \times 2$
Level 6 Mipmap – 1 × 1
Level 0 Mipmap – $64 \times 64$
Level 1 Mipmap – 32 × 32
Level 2 Mipmap – 16 × 16
Level 3 Mipmap – 8 × 8
Level 4 Mipmap – $4 \times 4$
Level 5 Mipmap – $2 \times 2$
Level 6 Mipmap – $1 \times 1$
Using Mipmaps

- When using mipmaps, we have two separate choices.
  - Whether to use the nearest texel in a mipmap or to interpolate among the 4 nearest texels.
  - Whether to use the nearest mipmap or to interpolate between the nearest two mipmaps.

- Thus, the choices are
  - Nearest texel, nearest mipmap
  - Nearest texel, interpolate mipmaps
  - Interpolate texels, nearest mipmap
  - Interpolate texels, interpolate mipmaps
Which Mip Map Level To Use

- In essence: count number of changes in texels along length of pixel cell
Which Mip-map to use

- Assume that a single color has been selected from each of the nearest two mipmaps (from either the nearest texel or an average of texels).
- Compute the scale factor $r$ between the level 0 (original) mipmap and the polygon.
- Then compute $\lambda = \log_2 r$, which tells us which mipmap to use.
  - If $\lambda = 0$, use level 0.
  - If $\lambda = 1$, use level 1.
  - If $\lambda = 2$, use level 2, etc.
- What if $\lambda = 1.5$?
  - Then we interpolate between level 1 and level 2.
Example

- Suppose $\lambda = 1.3$ and the level 1 mipmap color is yellow $(1, 1, 0)$ and the level 2 mipmap color is cyan $(0, 1, 1)$.
- Then the interpolated color is

$$0.7(1, 1, 0) + 0.3(0, 1, 1) = (0.7, 1.0, 0.3).$$
Trilinear Interpolation

- If we interpolate bilinearly within mipmaps and then interpolate those values between mipmaps, we get *trilinear interpolation*. 
Setting up a new texture in OpenGL

```c
GLuint myTex=0; /* openGL "texture handle" */

glGenTextures(1,&myTex); /* make a new texture */
glBindTexture(GL_TEXTURE_2D,myTex); /* modify (or draw) our texture */

GLenum format=GL_RGBA8; /* color, 8 bits per channel */
GLenum format=GL_RGBA8; /* color, 8 bits per channel */

glTexImage2D(GL_TEXTURE_2D,0,format, w,h,0, GL_RGBA,GL_UNSIGNED_BYTE,0);
/* the texture is w x h pixels */

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_LINEAR);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

Texture Coordinate Wrapping

- You can change what happens outside normal texture coordinate bounds, on both the S (x axis) and T (y axis) texture coordinate axes.

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

- Texture coordinates wrap back around to 0.0 when they exceed 1.0, which causes the texture to repeat. This is the default and most common wrap mode.

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
```

- Texture coordinates outside the normal range are flattened to 0.0-1.0.

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
```

- The edge pixels linearly blend down to the "border color" (usually a transparent black), which then extends outward.

Texture Magnification in OpenGL

- Nearest Neighbour:

```c
GLfloat GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```  
Return the color of the nearest texel. Results in ugly boxy shapes, but it's really fast, and never blurry.

- Bi-Linear Filtering:

```c
GLfloat GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```  
Take a bilinear blend of neighboring texels to create output values. Gives nice smooth output, but may be strange around the edges without GL_CLAMP_TO_BORDER or GL_REPEAT.
Texture Minification in OpenGL

- Nearest Neighbour:
  
  ```
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST);
  ```

  - Return the color of the nearest texel. Results in ugly boxy shapes, but it's really fast, and never blurry.

- Bi-Linear Filtering:
  
  ```
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_LINEAR);
  ```

  - Take a bilinear blend of neighboring texels to create output values. Gives nice smooth output, but may be strange around the edges without GL_CLAMP_TO_BORDER or GL_REPEAT.
Texture Minification in OpenGL with Mip-maps

- Tri-linear:
  ```c
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_LINEAR);
  ```
  - Tri-linear: Linearly interpolate pixels in the mipmaps, and then linearly interpolate between mipmaps. The smoothest version by far, and the recommended one to avoid aliasing.

- Bi-linear texels Nearest-neighbour mip-map
  ```c
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_NEAREST);
  ```
  - Linearly interpolate texels in the mipmaps, but only use the closest mipmap level. Slightly faster than linear-linear on older hardware.

- Nearest-neighbour texels & nearest mipmap level:
  ```c
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST_MIPMAP_NEAREST);
  ```
  - Use nearest-neighbor texels in the mipmaps, and only use the closest mipmap level. The ugliest approach, but good for checking for mipmap problems.

- Nearest neighbour texels & bi-linear mip-maps
  ```c
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST_MIPMAP_LINEAR);
  ```
  - Use nearest-neighbor pixels in the mipmaps, but blend between mipmap levels. Rarely useful.

- Note: If you enable one of these mipmap filtering modes, but don’t set up all the mipmap levels in your texture, then OpenGL will IGNORE your texture.

Setting up Mip Maps in OpenGL

- You automatically set up mipmaps with:

  ```c
  glGenMipmap(GL_TEXTURE_2D)
  ```

  - N.B. gluBuild2DMipmaps is deprecated and no longer recommended according to OpenGL.org

- Alternatively, you can manually load the individual levels using:

  - a set of `glTexImage2D` calls (one per level).
  - Specifying how many levels you require using `glTexParameter` e.g.

  ```c
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 4);
  ```
LOD Bias

- Level of detail bias is a value added on to \( l \) to affect the perceived sharpness of a textures
  - Instructs system to use a higher/lower mip-map level
  - e.g. use negative bias for blurry images; positive bias for aliased texture images

- In OpenGL set this using `glTexEnv`:
  ```c
  glTexEnvf(GL_TEXTURE_FILTER_CONTROL, GL_TEXTURE_LOD_BIAS, -1.5);
  ```
Anisotropic Filtering

pixel space

pixel's cell

texture

texture space

mipmap samples

line of anisotropy
Anisotropic Filtering

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, 16);
```
N.B. Texture wrapping and filtering modes apply to the currently bound texture--if you switch textures, your wrapping and filtering modes change too.
Texture Compression

color + luminance = final image
Multi-texture

- Two or more textures are applied at once on an object
  - Hardware supported to render in a single pass
  - Fixed function or Fragment shader
  - Draw object once with several textures,
Multi-texturing

- Some of which can be dynamically modified
  - Light map may be updated by a dynamic light and the fog map may by the moving camera.
  - The texture coordinate and vertex correspondence can be different for each texture map

\[
\text{Gloss Map} = (\text{+}) \times \text{(Gloss Map)}
\]
GLSL Example: Multi-texturing

**Vertex Program:**

```glsl
void main(void)
{
    gl_TexCoord[0] = gl_MultiTexCoord0;
    gl_TexCoord[1] = gl_MultiTexCoord1;
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

**Fragment Program:**

```glsl
uniform sampler2D myTexture1;
uniform sampler2D myTexture2;
void main (void)
{
    vec4 texval1 = texture2D(myTexture, vec2(gl_TexCoord[0]));
    vec4 texval2 = texture2D(myTexture2, vec2(gl_TexCoord[1]));
    gl_FragColor = 0.5*(texval1 + texval2);
}
```
Defining Texture Coordinates

```c
void drawBox(float size)
{
    // Texture Coordinate (0,0) is top left.
    glBegin(GL_QUADS);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
    glVertex3f(0.0, 0.0, 0.0);
    glMultiTexCoord2f(GL_TEXTURE0_ARB, 0.0, 0.0);
    glVertex3f(0.0, size*1.0, 0.0);
    glMultiTexCoord2f(GL_TEXTURE0_ARB, 1.0, 0.0);
    glVertex3f(size*1.0, 0.0, 0.0);
    glMultiTexCoord2f(GL_TEXTURE0_ARB, 1.0, 1.0);
    glVertex3f(size*1.0, size*1.0, 0.0);

    glMultiTexCoord2f(GL_TEXTURE1_ARB, 0.0, 0.0);
    glVertex3f(0.0, 0.0, 0.0);
    glMultiTexCoord2f(GL_TEXTURE1_ARB, 0.0, 0.0);
    glVertex3f(0.0, size*1.0, 0.0);
    glMultiTexCoord2f(GL_TEXTURE1_ARB, 1.0, 0.0);
    glVertex3f(size*1.0, 0.0, 0.0);
    glMultiTexCoord2f(GL_TEXTURE1_ARB, 1.0, 1.0);
    glVertex3f(size*1.0, size*1.0, 0.0);
    glEnd();
}
```

Full details: http://www.clockworkcoders.com/oglsl/tutorial8.htm
Multi-pass Rendering

An example of a high quality image generated at 60 Hz using multipass rendering techniques

Multi-pass Rendering

- Render the same scene multiple times and combine the results.
- This is the predominant approach for realistic imagery in real-time applications.
- Many advanced rendering techniques fall into this category.
  - E.g. shadow, light map, etc.
Multi-pass Rendering

- E.g. Quake III engine use 10 passes

- accumulate bump map
- diffuse lighting
- base texture
- specular lighting
- emissive lighting
- volumetric/atmospheric effects
- screen flashes
Light Mapping

- Stores the result of the diffuse component computation on static lighting (view independent)
- By multiplying the map with the underlying surface’s material, brightness or global intensity decreases
- Basically precomputed per-pixel lighting

![Diagram showing light mapping process]
Light Mapping
Light Mapping
Dynamic Lightmaps

1. Translate Texture
2. Scale Texture
3. Change Base Polygon Intensity
Light map example
Multipass Techniques

- Many cool effects are easier to achieve by rendering the scene multiple times:
  - Render the scene
  - Read the scene into a texture
  - Render that texture on the screen in some interesting way

- Main Benefits:
  - Speed and lower algorithm complexity:
    - Each pass is a specialised simplification of the rendering problem
    - Combine passes for apparent generality
Planar Reflections

- Simulate mirror reflection using multi-pass rendering
  - Reflect the eye about the plane of the mirror (OR reflect the scene)
  - Render a view from that point into texture
  - Render the scene as normal, placing that texture over the mirror
Sample Algorithm

- Render the scene excluding the mirror object as normal (stencil test off, depth test enabled)
  - Clear the stencil buffer and render the mirror polygon into the stencil and into Z buffer
  - Stencil buffer is set to draw only when depth test passes
  - Create a stencil mirror mask with visible mirror polygon tagged
  - Draw the mirror polygon into Z buffer using stencil mask so that visible mirror pixels are reset to maximum depth
  - Render the reflected scene using stencil testing
    - Objects that are behind the reflector plane should not be reflected
Shadow Mapping

- E.g. Shadow buffers:
  - Render scene from light’s viewpoint
  - Store depth values
  - Render scene from eye
  - Compare depths with shadow buffer

- More on Shadows in later lectures
Dynamic Environment Mapping

- Generate environment map on the fly by rendering objects onto six faces of cube map
- Problems with multiple reflecting objects:
  - Cubemap for each centre OR approximate from centre of scene
  - Reflecting objects reflecting each other
Image Based Effects

Depth-of field

Bloom

Motion Blur
Motion Blur

1. \( T = \) empty texture
2. Render a slightly dimmed \( T \)
3. Render the scene
4. Copy the framebuffer into \( T \)
5. Goto 2
Image Convolution

1. Render the scene
2. Save it as a texture
3. Render the texture, using a shader to convolve the image

- Convolution:

\[ H(x, y) = \sum_{j=0}^{\text{height}-1} \sum_{i=0}^{\text{width}-1} F(x + i, y + j) \cdot G(i, j) \]

- F is the base image
- G is the convolution kernel
Image Convolution

const int MaxKernelSize = 25;
uniform vec2 Offset[MaxKernelSize];
// size of kernel (width * height) for this execution

uniform int KernelSize; // final scaling value
uniform vec4 ScaleFactor; // image to be convolved
uniform sampler2D BaseImage;

void main(void)
{
    int i;
    vec4 sum = vec4(0.0);

    for (i = 0; i < KernelSize; i++)
    {
        sum += texture2D(BaseImage, gl_TexCoord[0].st + Offset[i]);
    }

    gl_FragColor = sum * ScaleFactor;
}

<table>
<thead>
<tr>
<th></th>
<th>Gaussian Blur</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0</td>
<td>4/273 16/273 26/273 16/273 4/273</td>
</tr>
<tr>
<td>1 −4 1</td>
<td>7/273 26/273 41/273 26/273 7/273</td>
</tr>
<tr>
<td>0 1 0</td>
<td>4/273 16/273 26/273 16/273 4/273</td>
</tr>
</tbody>
</table>

Edge Detection

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>−4</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Sharpening

<table>
<thead>
<tr>
<th>0</th>
<th>−1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1</td>
<td>4</td>
<td>−1</td>
</tr>
<tr>
<td>0</td>
<td>−1</td>
<td>0</td>
</tr>
</tbody>
</table>