Texture Filtering

MipMaps

“Optimal” case

Minification

Magnification

Magnification and Minification

More than one texel can cover a pixel (minification) or more than one pixel can cover a texel (magnification).

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values.

Pixel Footprint
Pyramid Textures (Mipmapping)

Linear vs. Nearest

Trilinear
Mipmapped Textures

- **Mipmapping** allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
  
  ```c
  glTexImage2D(GL_TEXTURE_*D, level, …)
  ```
- GL mipmap builder routines will build all the textures from a given image
  
  ```c
  glGenMipmap(GL_TEXTURE_*D)
  ```

---

Anisotropic Filtering

---

Anisotropic Filtering

---

Anisotropic Filtering
Light Mapping

- In order to keep the texture and light maps separate, we need to be able to perform multitexturing – application of multiple textures in a single rendering pass

- How do you create light maps?
  - Trying to create a light map that will be used on a non-planar object things get complex fast:
    - Need to find a divide object into triangles with similar orientations
    - These similarly oriented triangles can all be mapped with a single light map

Light Mapping

- Things for standard games are usually much easier since the objects being light mapped are usually planar:
  - Walls
  - Ceilings
  - Boxes
  - Tables
- Thus, the entire planar object can be mapped with a single texture map

Light Mapping

- Can dynamic lighting be simulated by using a light map?
- If the light is moving (perhaps attached to the viewer or a projectile) then the lighting will change on the surface as the light moves
  - Moving "flashlight" (use texture matrix)
  - The light map values can be partially updated dynamically as the program runs
  - Several light maps at different levels of intensity could be pre-computed and selected depending on the light’s distance from the surface

Lightmaps

- Creating local contribution

- Lightmap

- Unit Scene

- Lightmap Intensity

- Local Light Contribution
**Lightmaps**

- Adding local light to scene
  - Demo

**Lightmaps**

- Cached Lighting Results
  - Reuse lighting calculations
    - Multiple local lights (same type)
    - Static portion of scene’s light field
    - Sample region with texture instead of tessellating
  - Low resolution sampling
    - Local lighting: rapid change over small area
    - Global lighting: slow change over large area

**Lightmaps**

- Segmenting Scene Lighting
  - Static vs. dynamic light fields
  - Global vs. local lighting
  - Similar light shape

**Lightmaps**

- Segmenting the lighting

**Lightmaps**

- Moving Local Lights
  - Recreate the texture; simple but slow
  - Manipulate the lightmap
    - Translate to move relative to the surface
    - Scale to change spot size
    - Change base polygon color to adjust intensity
  - Projective textures ideal for spotlights
  - 3D textures easy to use (if available)

**Spotlights as Lightmap Special Case**

- Mapping Single Spotlight Texture Pattern

  Use texture matrix to perform spotlight texture coordinates transformations.
Lightmaps

• Creating a lightmap
  – Light white, tesselated surface with local light
  – Render, capture image as texture
  – Texture contains ambient and diffuse lighting
  – glLight() parameters should match light
  – Texture can also be computed analytically

Lightmaps

• Lightmap building tips
  – Boundary should have constant value
  – Intensity changes from light should be minimal near edge of lightmap

Lightmaps

• Lighting with a Lightmap
  – Local light is affected by surface color and texture
  – Two step process adds local light contribution:
    • Modulate textured, unlit surfaces with lightmap
    • Add locally lit image to scene
  – Can mix OpenGL, lightmap lighting in same scene

Lightmaps

• Creating local contribution
  – Unlit Scene
  – Lightmap Intensity
  – Local Light Contribution

Lightmaps

• Adding local light to scene
  – OpenGL Lighting
  – Combined Image
Lightmaps in Quake2

Lightmaps only \(\times\) decal only

= combined scene

Packing Many Lightmaps into a Single Texture

• Quake 2 light map texture image example

– Lightmaps typically heavily magnified.
– Permits multiple lightmaps packed into a single texture.
– Quake 2 computes lightmaps via off-line radiosity solver.

Lightmaps

• Lightmap considerations
  – Lightmaps are good:
    • Under-tessellated surfaces
    • Custom lighting
    • Multiple identical lights
    • Static scene lighting

• Lightmap considerations
  – Lightmaps less helpful:
    • Highly tessellated surfaces
    • Directional lights
    • Combine with other surface effects (e.g. bump-mapping)
      – eats a texture unit/access in fragment programs
      – may need to go to multi-pass rendering (fill-bound app)

Multitexturing

• Multitexturing allows the use of multiple textures at one time.
• It is a standard feature of OpenGL 1.3 and later.
• An ordinary texture combines the base color of a polygon with color from the texture image. In multitexturing, this result of the first texturing can be combined with color from another texture.
• Each texture can be applied with different texture coordinates.
Texture Units

- Multitexturing uses multiple texture units.
- A texture unit is a part of the rendering pipeline that applies one texture to whatever is being drawn.
- Each unit has a texture, a texture environment, and optional texgen mode. That is, its own complete and independent OpenGL texture state.
- Most current hardware has from 2 to 16 texture units.
- To get the number of units available:
  \[
  \text{glGetIntegerv(GL\_MAX\_COMBINED\_TEXTURE\_IMAGE\_UNITS)}
  \]

- Texture units are named GL\_TEXTURE0, GL\_TEXTURE1, etc.
- The unit names are used with two new functions.
  - \text{glActiveTexture(texture\_unit)}
    - selects the current unit to be affected by texture calls (such as \text{glBindTexture}, \text{glTexEnv}, \text{glTexGen}).
  - Use vertex attributes to set texture coordinates for each unit.

OpenGL Multitexture Quick Tutorial

- Configuring multitextures:
  - \text{GLuint\ textures[3];}
  - \text{glGenTextures(3, &textures);}
  - \text{glActiveTexture(GL\_TEXTURE0);}
  - \text{glBindTexture(GL\_TEXTURE\_2D, textures[0]);}
  - \text{glActiveTexture(GL\_TEXTURE\_1;)
  - \text{glBindTexture(GL\_TEXTURE\_2D, textures[1]);}
  - \text{glActiveTexture(GL\_TEXTURE2);}
  - \text{glBindTexture(GL\_TEXTURE\_2D, textures[2]);}

  
  \text{tex0\_uniform\_loc = glGetUniformLocation(prog, \text{"tex0"});}
  \text{glUniform1i(tex0\_uniform\_loc, 0);}
  \text{tex1\_uniform\_loc = glGetUniformLocation(prog, \text{"tex1"});}
  \text{glUniform1i(tex1\_uniform\_loc, 1);}
  \text{tex2\_uniform\_loc = glGetUniformLocation(prog, \text{"tex2"});}
  \text{glUniform1i(tex2\_uniform\_loc, 2);}

OpenGL Multitexture Texture Environments (old way)

- Chain of Texture Environment Stages:
  - Pre-texturing color
    - TexCoord[0] = TextureMatrix[0] * MultiTexCoord0
  - TexCoord[2]
  - \text{void main(){}

  OpenGL Multitexture Texture Environments (new way)

- Chain of Texture Environment Stages:
  - Put it in the shaders!

  \text{varying vec3 lightDir, normal, TexCoord[0];}
  \text{void main(){}
  \text{TexCoord[0] = TextureMatrix[0] * MultiTexCoord0;}
  \text{vec3  ct, cf;}
  \text{vec4 texel;}
  \text{float intensity, at, af;}
  \text{intensity = max(dot(lightDir, normalize(normal)),0.0);}
  \text{cf = intensity * gl\_FrontMaterial\_diffuse\_rgb + gl\_FrontMaterial\_ambient\_rgb;}
  \text{af = gl\_FrontMaterial\_diffuse\_a;}
  \text{texel = texture2D(tex0, TexCoord[0].st) + texture2D(tex0, TexCoord[1].st);}
  \text{ct = texel.rgb;}
  \text{at = texel.a;}
  \text{gl\_FragColor = vec4(ct*cf, at*af);}
  \text{}}
  \text{}}
Detail Texture

Multitexture Lightmapping

Alpha Mapping
• An Alpha Map contains a single value with transparency information
  – 0 → fully transparent
  – 1 → fully opaque
• Can be used to make sections of objects transparent
• Can be used in combination with standard texture maps to produce cutouts
  – Trees
  – Torches

Alpha Mapping
• In the previous tree example, all the trees are texture mapped onto flat polygons
• The illusion breaks down if the viewer sees the tree from the side
• Thus, this technique is usually used with another technique called "billboarding"
  – Simply automatically rotating the polygon so it always faces the viewer
• Note that if the alpha map is used to provide transparency for texture map colors, one can often combine the 4 pieces of information (R,G,B,A) into a single texture map

Alpha Mapping
• The only issue as far as the rendering pipeline is concerned is that the pixels of the object made transparent by the alpha map cannot change the value in the z-buffer
  – We saw similar issues when talking about whole objects that were partially transparent → render them last with the z-buffer in read-only mode
  – However, alpha mapping requires changing z-buffer modes per pixel based on texel information
  – This implies that we need some simple hardware support to make this happen properly
Bill Boarding

• How?

Bill Boarding

• Eye looking down –Z axis, UP = +Y axis
• Compute eye-vector from ModelView:

• Rotation about Y:

Build rotation matrix (R) with theta
Transform geometry: MR (Modelview * Rotation)

Billboards

look = camera_pos - point_pos;
right = up x look;
up = look x right;

Billboards Hack

• Trees don’t face camera

Billboards Hack

• Trees don’t face camera
• Use the Modelview
• Set rotation to identity
• Spherical Billboarding
Billboards Hack 2

• Trees don’t face camera
• Use the Modelview
• Make billboard cylindrical
• Set part of rotation to identity

Billboards Hack 3

• Modify the vertices of the Billboard quad
• Reverses the orientations in the Modelview Matrix
• Draw quad using right/up offsets
• Only get modelview once
• Must transform all vertices

How to do cylindrical?

Billboards Correct

• Trees face camera
• Need
  – Object in world coords
  – Target position (camera) in world coords
• Assume for the object (billboard)
  Right = [1,0,0]
  Up = [0,1,0]
  LookAt = [0,0.1] (which is the normal)
Billboards Correct

objToCamProj is the projection to the XZ plane (set y=0)

1. Normalize objToCamProj
2. aux = LookAt dot objToCamProj
3. Up" = lookAt X objToCamProj
4. glRotate(acos(aux), Up"[0], Up"[1], Up"[2])

Tilt towards Camera

1. Aux = objToCamProj dot objToCam
2. glRotate(acos(aux'), right[0], right[1], right[2])
Billboards

Billboard Clouds