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
  ```
  glTexImage2D( GL_TEXTURE_*D, level, …
  )
  ```
- GLU mipmap builder routines will build all the textures from a given image
  ```
  gluBuild*DMipmaps( … )
  ```

Example

- point sampling
- mipmapped point sampling
- linear filtering
- mipmapped linear filtering

Demo
Anisotropic Filtering

Figure 21. Poor data in anisotropically scaled textures

Figure 22. Geometry orientation and texture aspect ratio

Anisotropic Filtering

Figure 23. Creating a set of anisotropically filtered images
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

[Images of texture, lightmap, and texture * lightmap]

Light Mapping

• 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

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
• 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:

\[
\mathbf{v}_{\text{eye}} = \mathbf{d} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}
\]

- Rotation about Y:

\[
\begin{align*}
\cos \theta &= \mathbf{v}_{\text{eye}} \cdot \mathbf{v}_{\text{front}} \\
\sin \theta &= \mathbf{v}_{\text{eye}} \cdot \mathbf{v}_{\text{right}}
\end{align*}
\]

Where:

\[
\mathbf{v}_{\text{front}} = (0, 0, 1)
\]

\[
\mathbf{v}_{\text{right}} = (1, 0, 0)
\]

- Build rotation matrix (R) with theta
- Transform geometry: \( \mathbf{M} \mathbf{R} = (\mathbf{M} \text{Modelview} * \mathbf{R} \text{Rotation}) \)

Billboards

\[
\begin{align*}
\text{look} &= \text{camera_pos} - \text{point_pos} \\
\text{right} &= \text{up} \times \text{look} \\
\text{up} &= \text{look} \times \text{right}
\end{align*}
\]
Billboards

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

Billboards Hack

- Trees don’t face camera
Billboards Hack

- Trees don’t face camera
- Use the Modelview
- Set rotation to identity

```c
void billboardCheatSphericalBegin() {
    float modelview[16];
    int i, j;

    // save the current modelview matrix
    glPushMatrix();

    // get the current modelview matrix
    glGetFloatv(GL_MODELVIEW_MATRIX, modelview);

    // undo all rotations
    // beware all scaling is lost as well
    for (i=0; i<3; i++)
        for (j=0; j<3; j++) {
            if (i==j)
                modelview[i*4+j] = 1.0;
            else
                modelview[i*4+j] = 0.0;
        }

    // set the modelview with no rotations and scaling
    glLoadMatrixf(modelview);
}

void billboardEnd() {
    // restores the modelview matrix
    glPopMatrix();
}
```

```c
billboardCheatSphericalBegin();
```

```c
drawObject();
```

```c
billboardEnd();
```

If scaling:

```c
billboardCheatSphericalBegin();
```

```c
glScalef(1.2, 1);
```

```c
drawObject();
```

```c
billboardEnd();
```
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 xform all vertices

\[
\begin{align*}
    a &= \text{center} - (\text{right} + \text{up}) \times \text{size}; \\
    b &= \text{center} + (\text{right} - \text{up}) \times \text{size}; \\
    c &= \text{center} + (\text{right} + \text{up}) \times \text{size}; \\
    d &= \text{center} - (\text{right} - \text{up}) \times \text{size};
\end{align*}
\]
Billboards Correct

- Trees face camera

- Need
  - Object in world coords
  - Target position in world coords

- Assume for the object
  
  Right = [1,0,0]
  Up = [0,1,0]
  LookAt = [0,0,1] {which is the normal}
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])
void billboardCylindricalBegin(
    float camX, float camY, float camZ,
    float objPosX, float objPosY, float objPosZ,
) {
    float lookAt[3], objToCamProj[3], aux[3];
    float modelView[16], angleCosine;
    glMatrixMode(GL_PROJECTION);
    // objToCamProj is the vector in world coordinates from the
    // local origin to the camera projected in the XZ plane
    objToCamProj[0] = camX - objPosX ;
    objToCamProj[1] = 0 ;
    objToCamProj[2] = camZ - objPosZ ;
    // This is the original lookAt vector for the object
    // in world coordinates
    lookAt[0] = 0;
    lookAt[1] = 0;
    lookAt[2] = 1;
    // normalize both vectors to get the cosine directly afterwards
    mathNormalize(objToCamProj);
    // easy fix to determine weather the angle is negative or positive
    // for positive angles aux will be a vector pointing in the
    // positive y direction, otherwise upAux will point downwards
    // effectively reversing the rotation.
    mathsCrossProduct(upAux, lookAt, objToCamProj);
    // compute the angle
    angleCosine = mathInnerProduct(lookAt, objToCamProj);
    // perform the rotation, the if statement is used for stability reasons
    // if the lookAt and objToCamProj vectors are too close together then
    // [angleCosine] could be bigger than 1 due to lack of precision
    if (angleCosine < 3.99999) {
        angleCosine = -0.99999;
    }
    glRotatef(acos(angleCosine) * 180 / 3.14, upAux[0], upAux[1], upAux[2]);
}

Billboards Correct

**objToCamProj** is the projection to the XZ plane (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])
Billboards Correct

objToCamProj is the projection to the XZ plane (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])

Object Position in world space

\[
\text{objPosWC} = \text{camPos} + (M_1^{-1}) \cdot V
\]

\[
V^T = [a_{12}, a_{13}, a_{14}]
\]