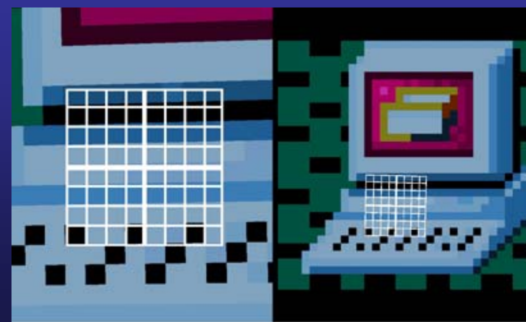
Texture Filtering

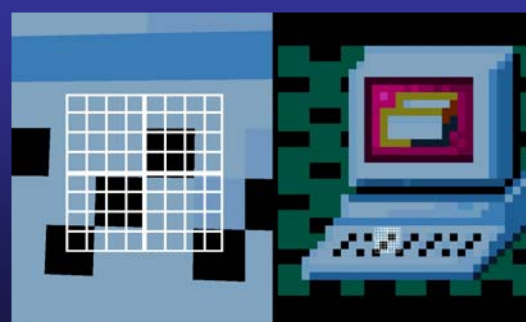
MipMaps

“Optimal” case



Minification

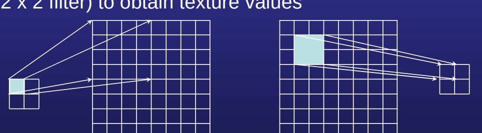
Magnification



Magnification and Minification

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

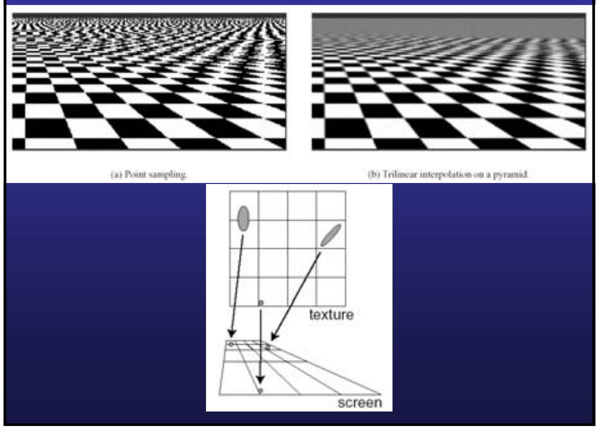
Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values



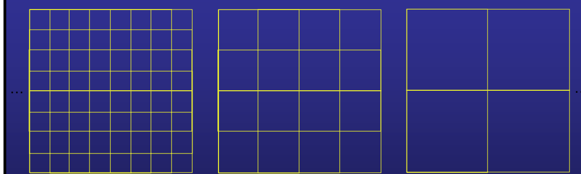
Texture Polygon
Magnification

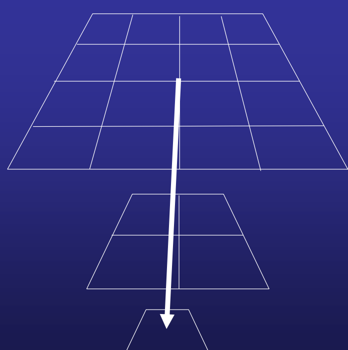
Texture Polygon
Minification

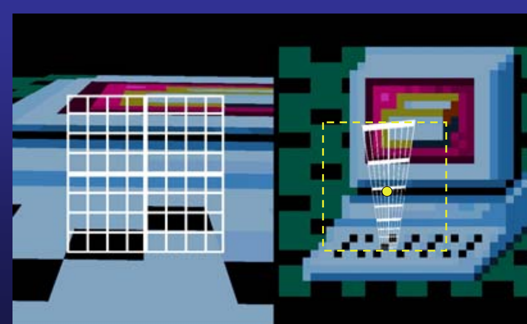
Pixel Footprint



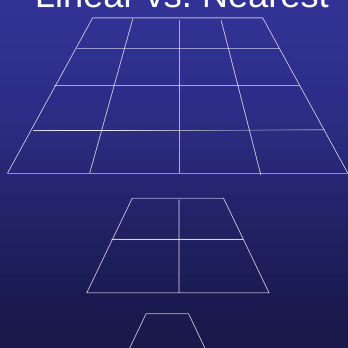
Pyramid Textures (Mipmapping)



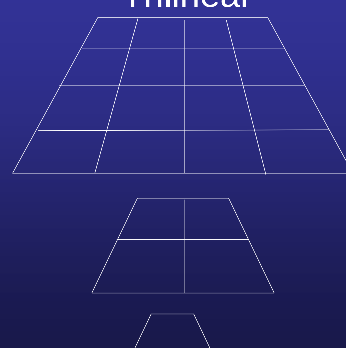




Linear vs. Nearest



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, …
)
```

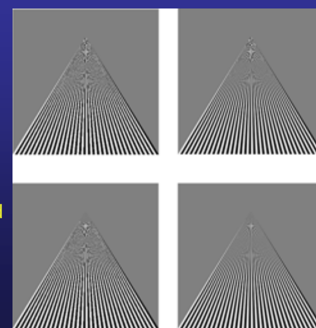
- GLU mipmap builder routines will build all the textures from a given image

```
gluBuild*DMipmaps( … )
```

Example

point sampling

linear filtering

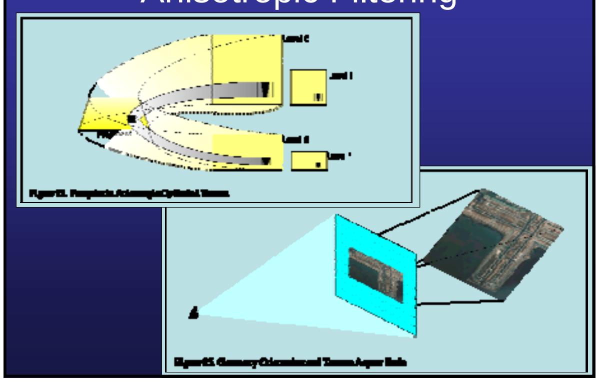


mipmapped point sampling

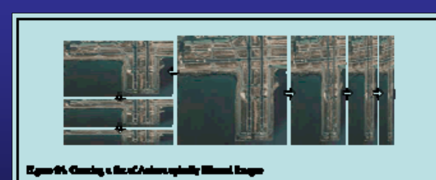
mipmapped linear filtering

Demo

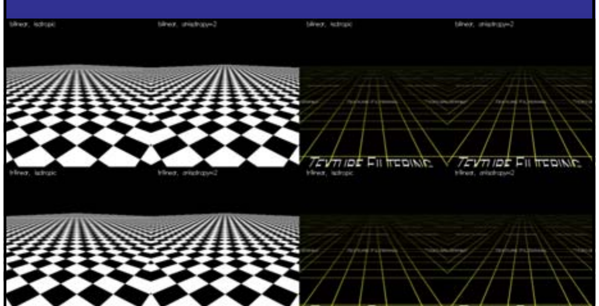
Anisotropic Filtering

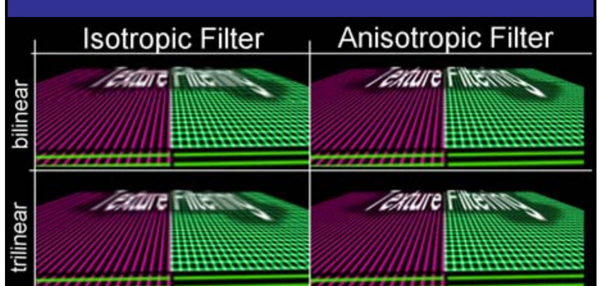


Anisotropic Filtering



Anisotropic Filtering



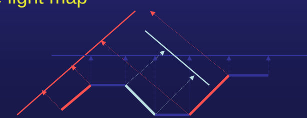


Light Mapping

- 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

Light Mapping

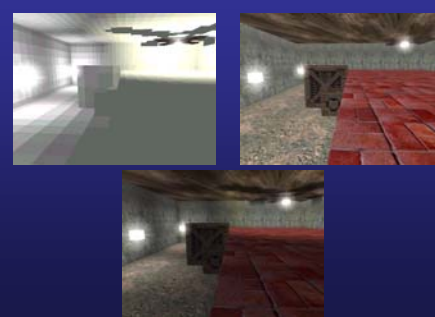
- How do you create light maps?
- Trying to create a light map that will be used on a non-planar object things get complex fast:
  - Need to find a divide object into triangles with similar orientations
  - These similarly oriented triangles can all be mapped with a single light map



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

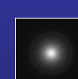


Light Mapping

- 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


Unlit Scene | Lightmap Intensity | Local Light Contribution

Lightmaps

- Adding local light to scene


OpenGL Lighting


Combined Image

- 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


Dominant Lighting


Local 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


Original | Translate Spotlight Texture Coordinates | Scale Spotlight Texture Coordinates | Change Base Polygon Intensity

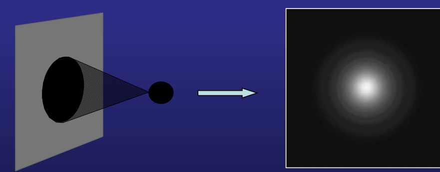
Use texture matrix to perform spotlight texture coordinates transformations.

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

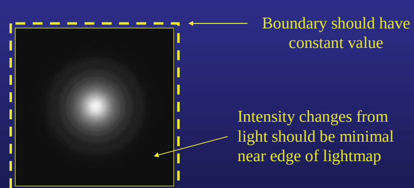
- Creating a lightmap



Render surface lit by local light — Create a Texture Map from Image

Lightmaps

- Lightmap building tips

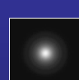


Lightmaps

- 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



Lightmap

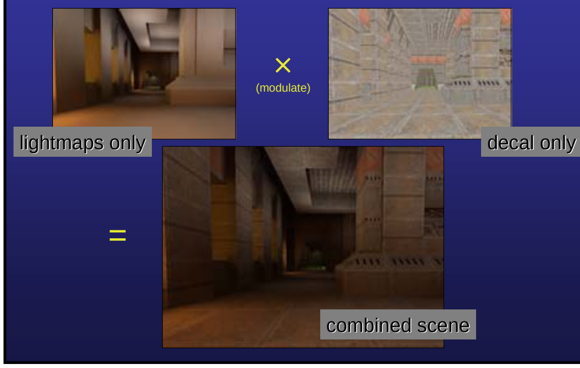
Unlit Scene — Lightmap Intensity — Local Light Contribution

Lightmaps

- Adding local light to scene

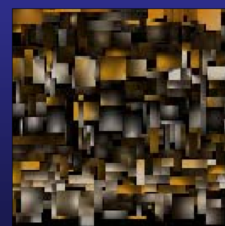
OpenGL Lighting — Combined Image

Lightmaps in Quake2



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

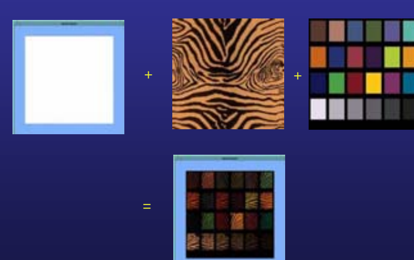
- Lightmap considerations
  - Lightmaps are good:
    - Under-tessellated surfaces
    - Custom lighting
    - Multiple identical lights
    - Static scene lighting

Lightmaps

- 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)*

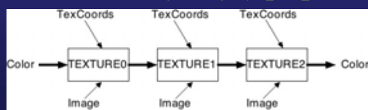
Multitexturing

- **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: glGetIntegerv(GL_MAX_TEXTURE_UNITS)



Texture Units

- Texture units are named GL_TEXTURE0, GL_TEXTURE1, etc.
- The unit names are used with two new functions.
- glActiveTexture(texture_unit)
  - selects the current unit to be affected by texture calls (such as glBindTexture, glTexEnv, glTexGen).
- glMultiTexCoord2f(texture_unit, s, t)
  - Sets texture coordinates for one unit

OpenGL Multitexture Quick Tutorial

– Configuring up a given texture unit:

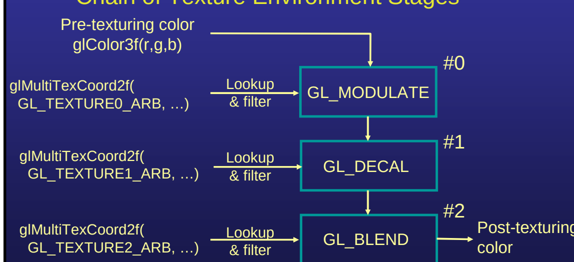
```
tex1_uniform_loc = glGetUniformLocation(prog, “tex1”);
glUniform1i(tex1_uniform_loc, 1);
glActiveTexture(GL_TEXTURE1);            ← Sets active texture unit
glBindTexture(GL_TEXTURE_2D, texObject);   ┐
glTexImage2D(GL_TEXTURE_2D, …);            │ update
glTexParameterfv(GL_TEXTURE_2D, …);        │ state of
glTexEnvfv(GL_TEXTURE_ENV, …);             │ active
glTexGenfv(GL_S, …);                       │ texture
glMatrixMode(GL_TEXTURE);                  │ unit
glLoadIdentity();                          ┘
```

– Setting texture coordinates for a vertex:

```
glMultiTexCoord4f(GL_TEXTURE0, s0, t0, r0 ,q0);
glMultiTexCoord2f(GL_TEXTURE1, s1, t1);
glMultiTexCoord3f(GL_TEXTURE2, s2, t2, r2);
glVertex3f(x, y, z);
```

OpenGL Multitexture Texture Environments (old way)

- Chain of Texture Environment Stages

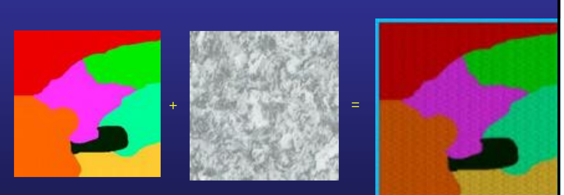


OpenGL Multitexture Texture Environments (new way)

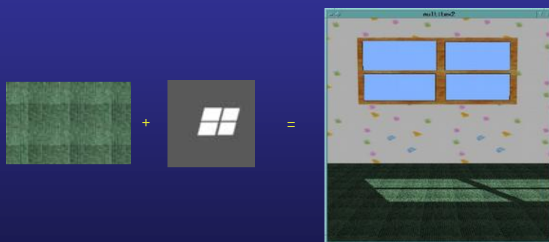
- Chain of Texture Environment Stages: put it in the shaders!

```
varying vec3 lightDir, normal;
void main(){
	gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;
	gl_TexCoord[1] = gl_TextureMatrix[1] * gl_MultiTexCoord1;
	gl_Position = ftransform();
}

varying vec3 lightDir, normal;
uniform sampler2D tex0, tex1;
void main(){
	vec3  ct, cf;
	vec4 texel;
	float intensity, at, af;
	intensity = max(dot(lightDir, normalize(normal)),0.0);
	cf = intensity * gl_FrontMaterial.diffuse.rgb + gl_FrontMaterial.ambient.rgb;
	af = gl_FrontMaterial.diffuse.a
	texel = texture2D(tex0,gl_TexCoord[0].st) + texture2D(tex0,gl_TexCoord[1].st);
	ct = texel.rgb;
	at = texel.a;
	gl_FragColor = vec4(ct*cf, at*af);
}
```

Detail Texture



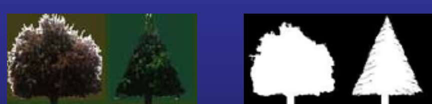
Multitexture Lightmapping

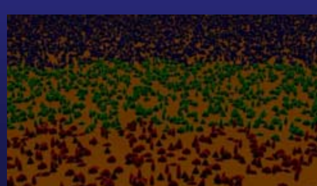


Alpha Mapping

- 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





Alpha Mapping

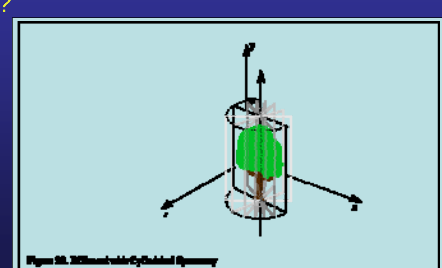
- In the previous tree example, all the trees are texture mapped onto flat polygons
- The illusion breaks down if the viewer sees the tree from the side
- Thus, this technique is usually used with another technique called "billboarding"
  - Simply automatically rotating the polygon so it always faces 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

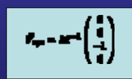
- How?



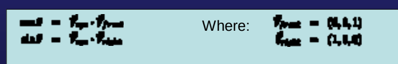
Demo

Bill Boarding

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

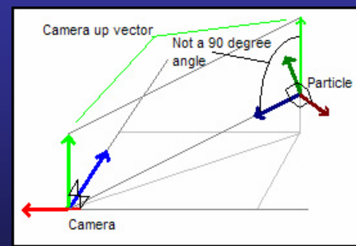


- Rotation about Y:



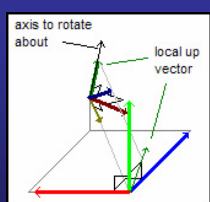
- Build rotation matrix (R) with theta
- Transform geometry: MR (Modelview * Rotation)

Billboards



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

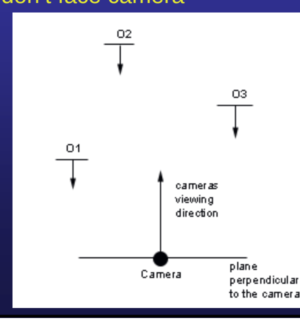
Billboards



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

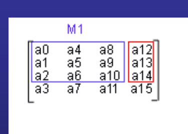
Billboards Hack

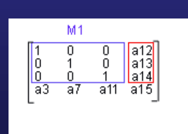
- Trees don't face camera



Billboards Hack

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






```
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();
}
```

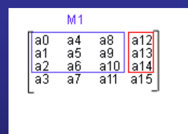
```
billboardCheatSphericalBegin();
	drawObject();
billboardEnd();
```

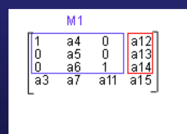
If scaling:

```
billboardCheatSphericalBegin();
    glScalef(1,2,1);
    drawObject();
billboardEnd();
```

Billboards Hack 2

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

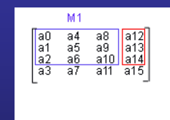


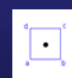


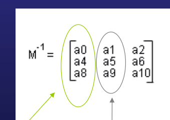
Billboards Hack 3

- Modify the verticies of the Billboard quad
- Reverses the orientations in the Modelview Maxtrix
- Draw quad using right/up offsets

+ only get modelview once
- Must xform all verticies







a = center - (right + up) * size;
b = center + (right - up) * size;
c = center + (right + up) * size;
d = center - (right - up) * size;

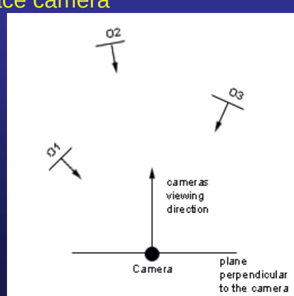
right

up

How to do cylindrical?

Billboards Correct

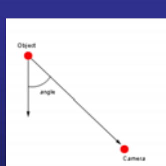
- Trees face camera

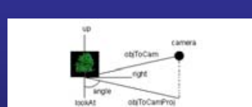


Billboards Correct

- 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}

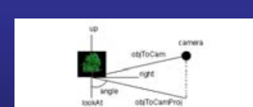
Billboards Correct





Billboards Correct

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



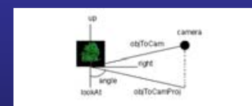
1. **Normalize objToCamProj**
2. **aux=LookAt dot objToCamProj**
3. **Up'= lookAt X objToCamProj**
4. **glRotate(acos(aux), Up'[0], Up'[1], Up'[2]**

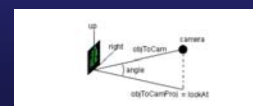
```
void billboardCylindricalBegin(
                        float camX, float camY, float camZ,
                        float objPosX, float objPosY, float objPosZ) {

        float lookAt[3],objToCamProj[3],upAux[3];
        float modelview[16],angleCosine;

        glPushMatrix();

// 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
        mathsNormalize(objToCamProj);

// easy fix to determine wether the angle is negative or positive
// for positive angles upAux 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 = mathsInnerProduct(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 < 0.99990) && (angleCosine > -0.9999))
      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]

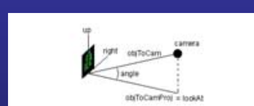




Billboards Correct

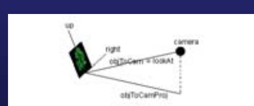
objToCamProj is the projection to the XZ plane (y=0)

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

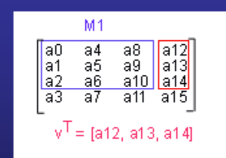
1. Aux'= objToCamProj dot objToCam
2. glRotate(acos(aux'), right[0], right[1], right[2])



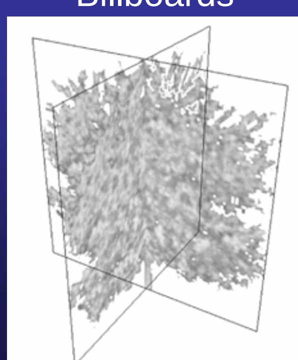
Billboards Correct

Object Position in world space

objPosWC = camPos + ($M1^{-1}$) * V



Billboards



Billboard Clouds