ENVIRONMENT MAPPING

Programação 3D Simulação e Jogos
2014/2015

Prof. João Madeiras Pereira
MEIC/IST

Environment Mapping

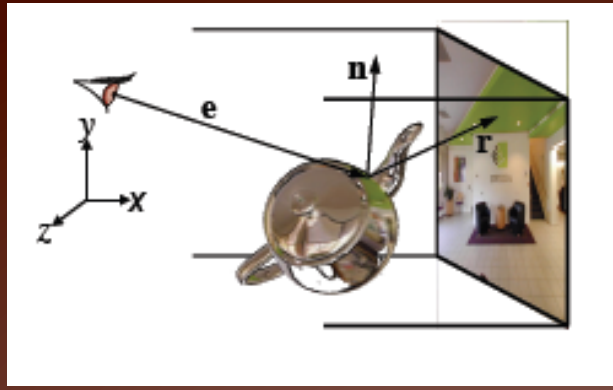
- The key to depicting a shiny-looking material is providing something for it to reflect
- proper reflection requires ray tracing, expensive
- It is simulated with a pre-rendered environment, stored as a texture

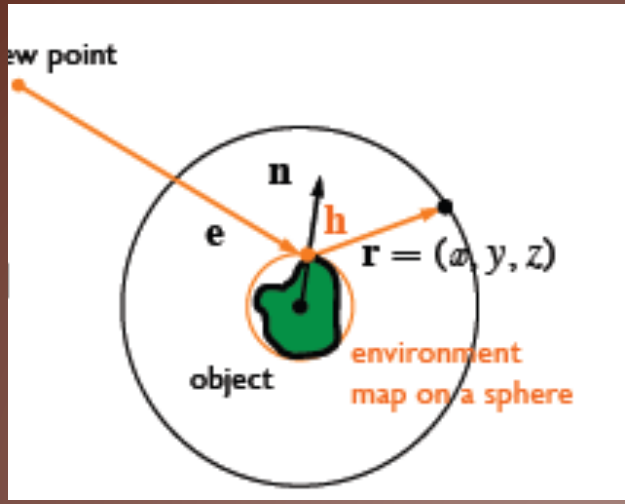
environment mapped

ray traced

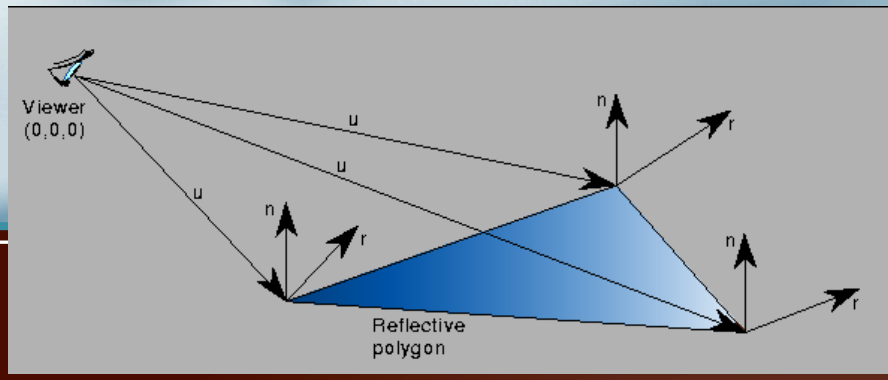
Environment Mapping

- **Environment mapping is essentially the process of pre-computing a texture map and then sampling texels from this texture during the rendering of a model**
- imagine object is enclosed in an infinitely large sphere or cube
- rays are bounced off object into environment to determine color
- Env map * surface color = reflection mapping



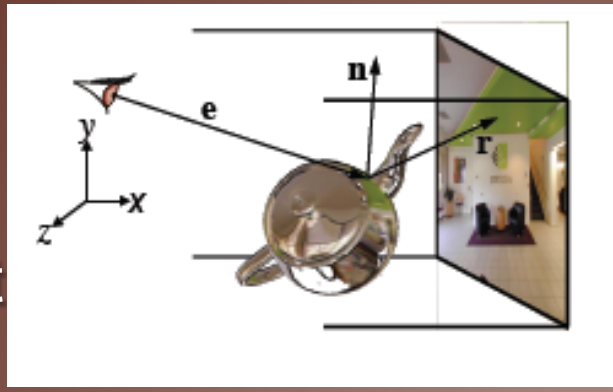


Environment Mapping



Steps:

- load environment map
- for each reflective pixel, compute its normal
- compute the reflection vector from the normal and view vectors
- use the reflection vector to compute an index into the environment map in the reflection direction
  - note: we're not computing ray intersection between the reflection vector and the environment
- use the texel at the index to color the pixel



Shortcomings:

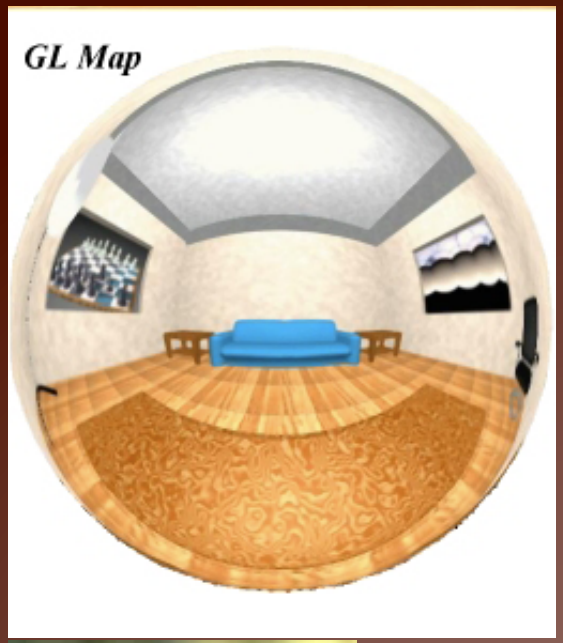
- no inter-object reflection
- works well when there's just a single object
- no self-reflection (convex ones)

Methods to Create EM

- Cube map: loading 6 disjoint textures, each one to be mapped to a cube face; the reflection vector is calculated in World space
- Spherical map
  - Photographing a gazing ball with fisheye lens
  - Pre-rendering program (it can be ray-tracing)
  - Photoshop to edit a texture and then use the filter spherize
  - the reflection vector is calculated in Eye space

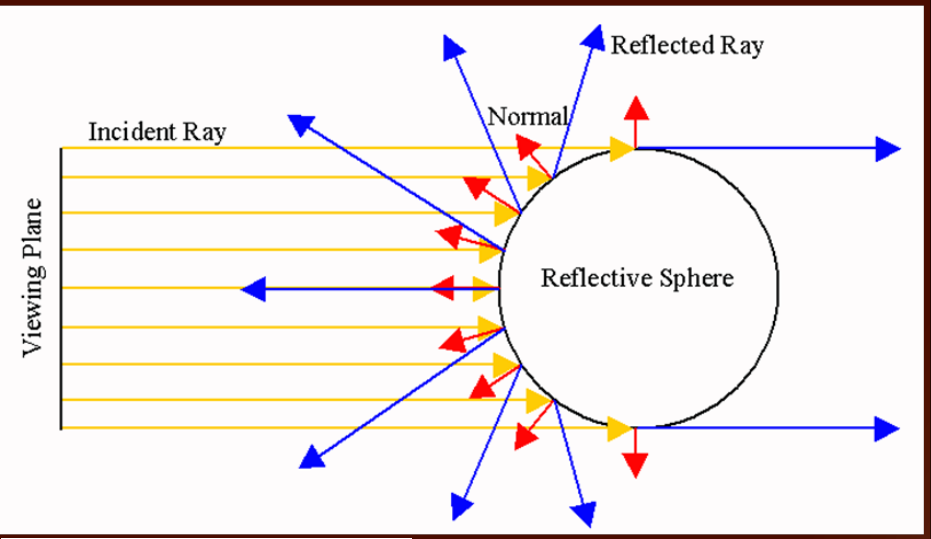
Gazing Ball (Light Probe)

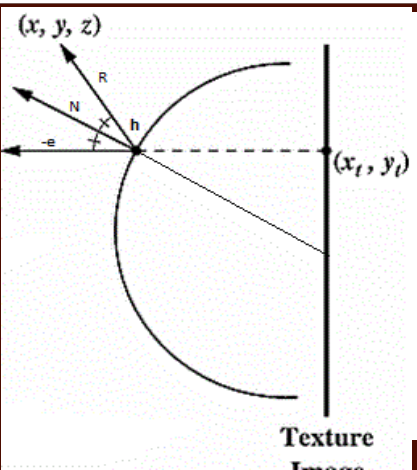
- Created by photographing a reflective sphere;
- or by ray-casting-based programming
- Maps all directions to a circle
- Reflection indexed by normal
- Resolution function of orientation
- View dependent: must regenerate EM when camera moves or will see the same thing!

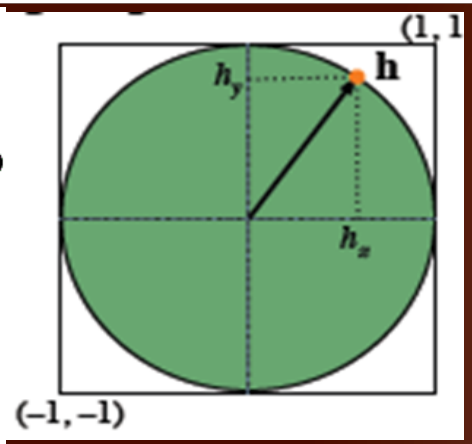


2D Sphere Map

- image of perfect reflective sphere seen from infinity
- 2D sphere map: orthogonal projection of a unit sphere reflecting the environment infinitely far away; will obtain a circle within a square texture



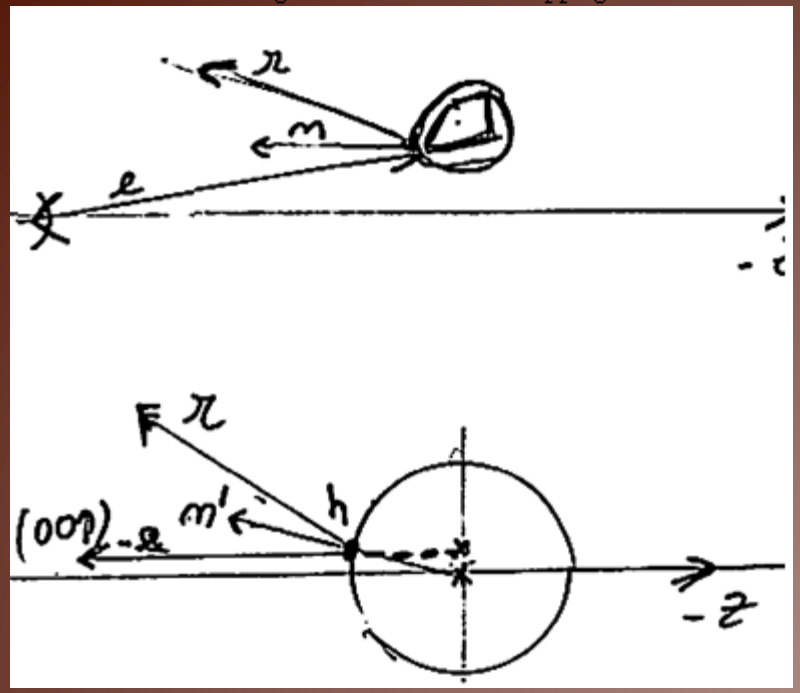




Sphere Mapping algorithm

- use a 2D sphere map (circle): orthogonal projection of a unit sphere reflecting the environment infinitely far away;
- object can be approximated as an infinitely small, perfectly mirroring ball concentric with the object;

Figure 7. Environment Mapping.

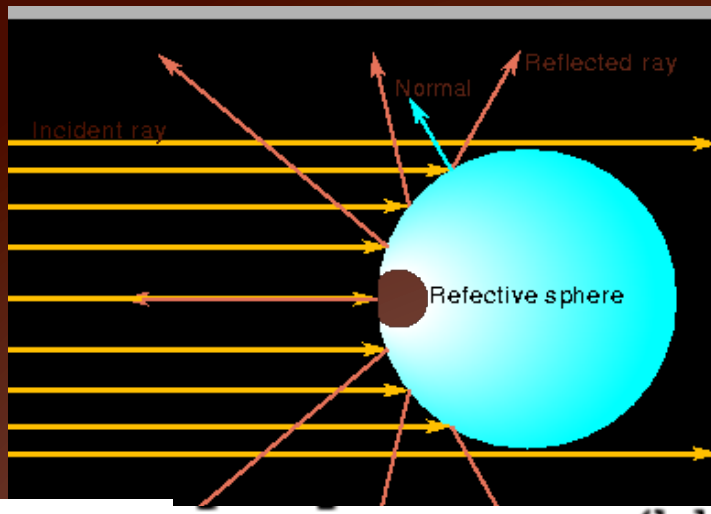


Want: compute texture coordinates (*s*, *t*) from reflection vectors calculated at the vertices of the model

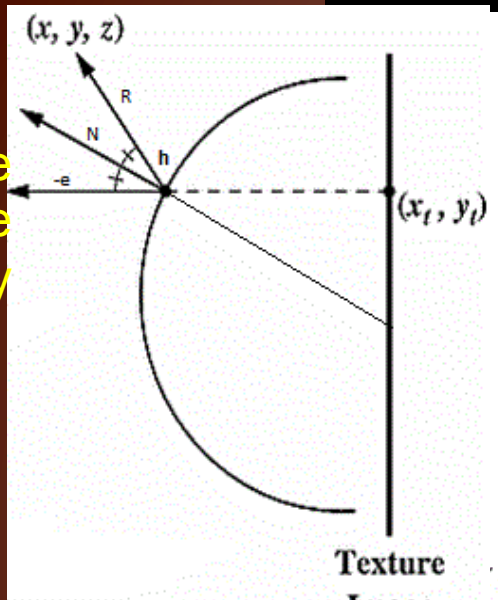
To index the 2D Sphere Map

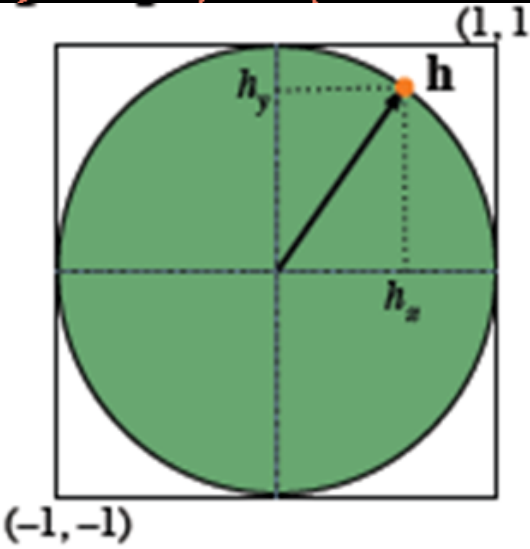
Remember:

- 2D texture built from the orthographic projection of a unit **sphere viewed from infinity**
- **viewed from infinity** means that the eye vectors are parallel; so
- the reflection vector at a point is determined only by the normal at that point or **the reciprocal**



Thus, to address a texel in the 2D sphere map, it suffices to find the point on the sphere surface which is determined by the direction of **r**





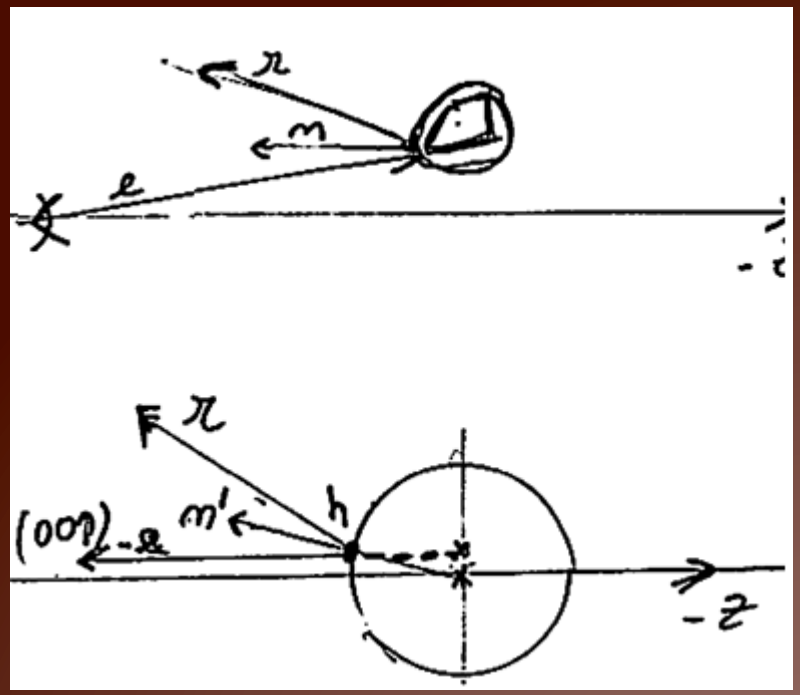
Calculating (s, t)

Calculate r vector at the vertex of the object:

r = e – 2n(n•e)

Compute the point h at the unit sphere, i.e. calculate the normal n' by using the direction of the vector r:

$$n' = \frac{r-e}{\| r-e \|} = \left[\frac{R_x}{p} \quad \frac{R_y}{p} \quad \frac{R_z+1}{p} \quad 0 \right]$$



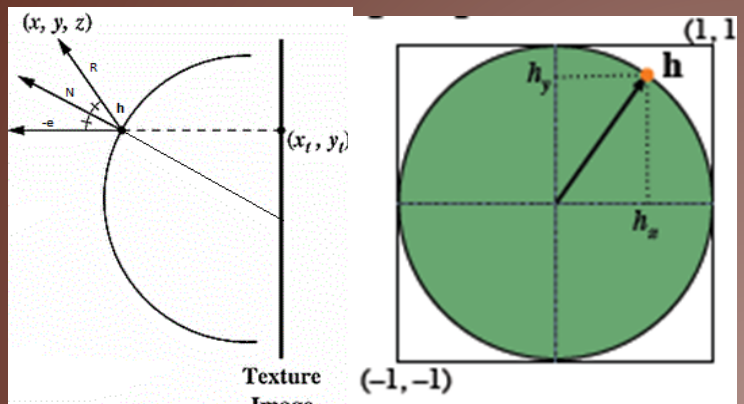
$$h_x = \frac{R_x}{p}, \quad h_y = \frac{R_y}{p}$$

$$p = \sqrt{R_x^{\,2} + R_y^{\,2} + (R_z+1)^2} = \sqrt{2(R_z+1)}$$

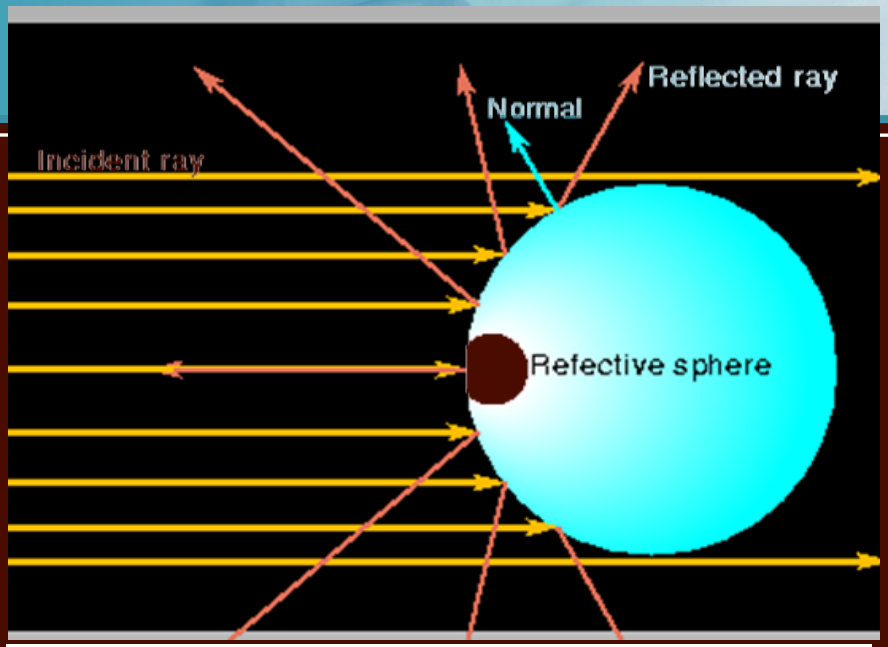
Texture coord -> Scale to [0 1]:

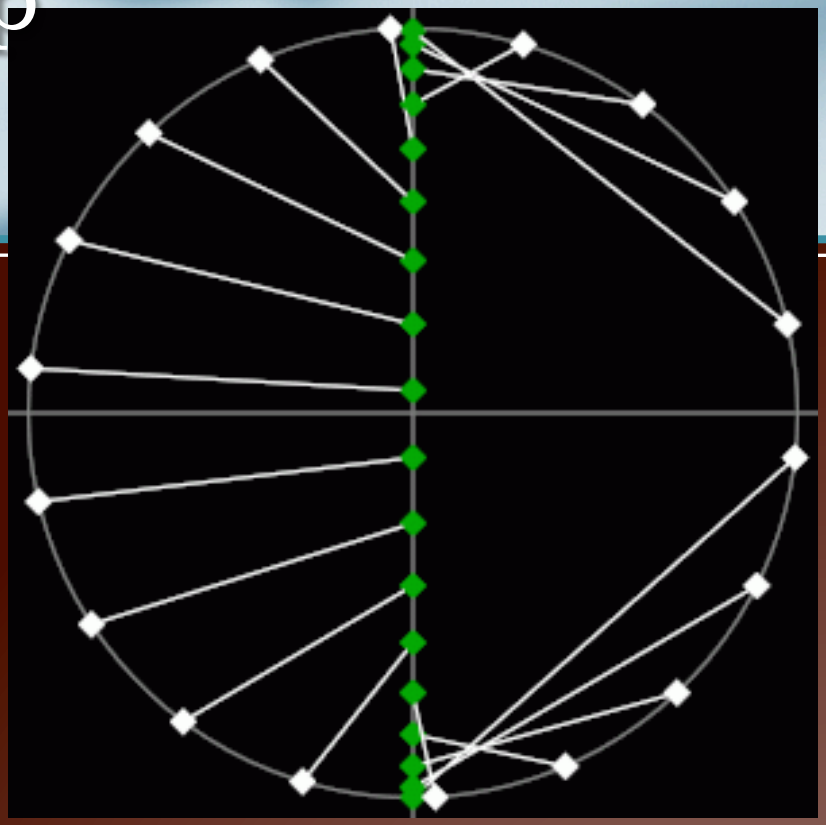
$$s = \frac{R_x}{2p} + \frac{1}{2} \qquad t = \frac{R_y}{2p} + \frac{1}{2}$$

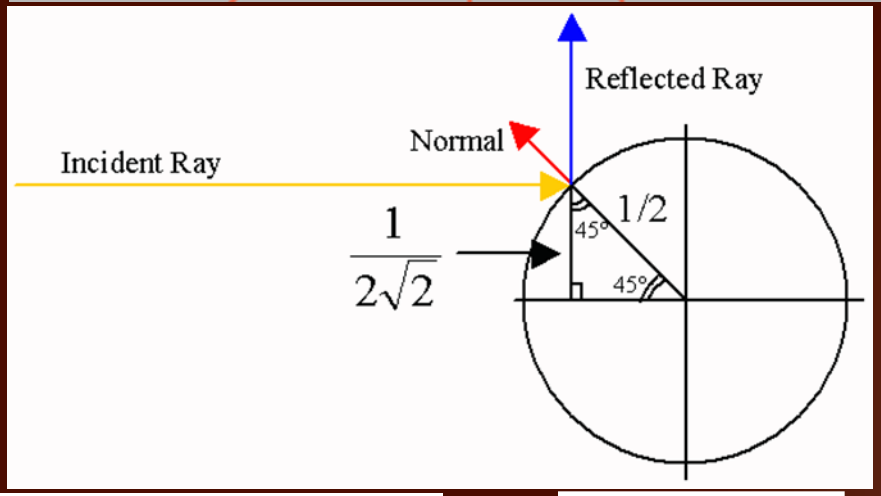
$$p = \sqrt{2(R_z+1)}$$

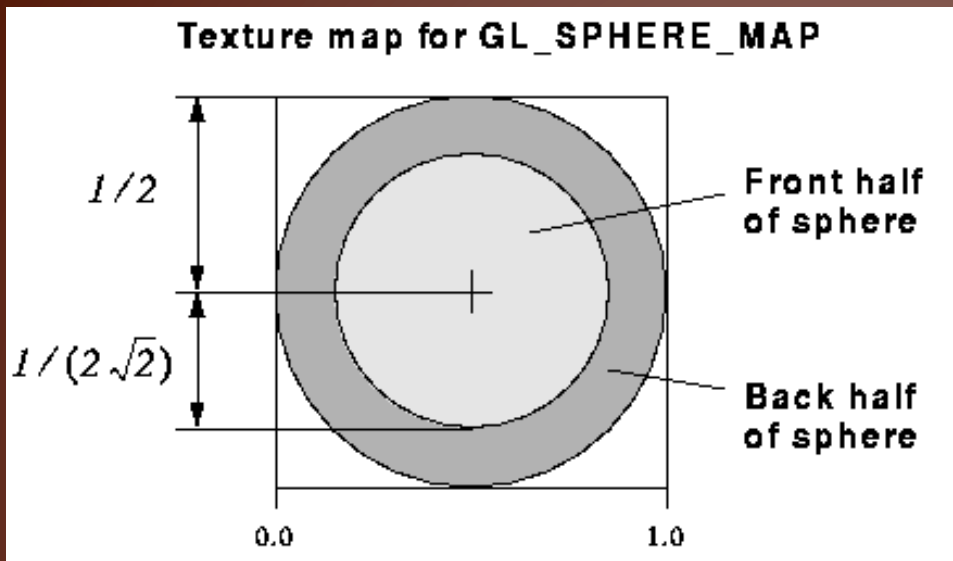


Advanced Sphere Map





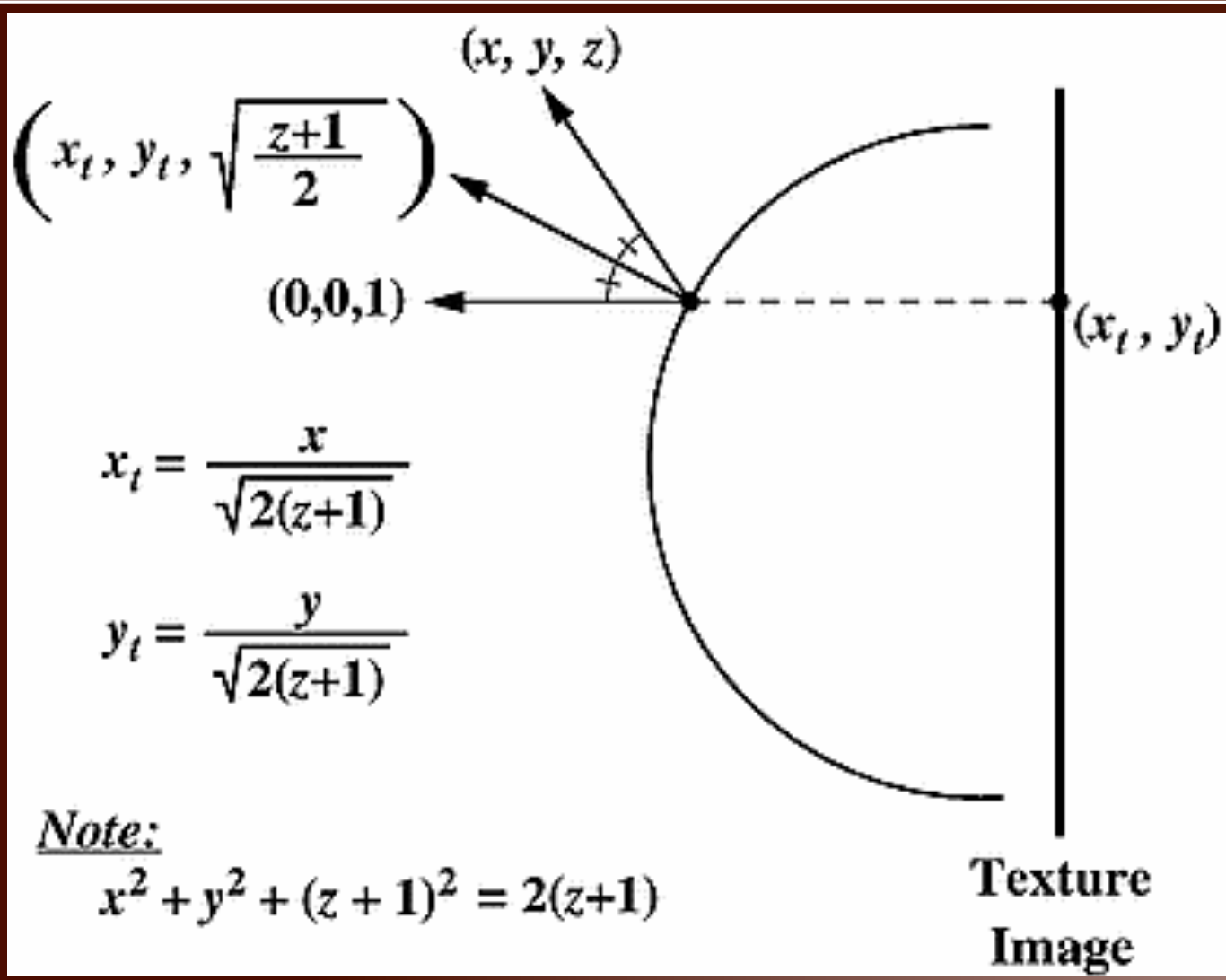




$$s = \frac{R_x}{2p} + \frac{1}{2} \qquad t = \frac{R_y}{2p} + \frac{1}{2}$$

$$p = \sqrt{2(R_z + 1)}$$

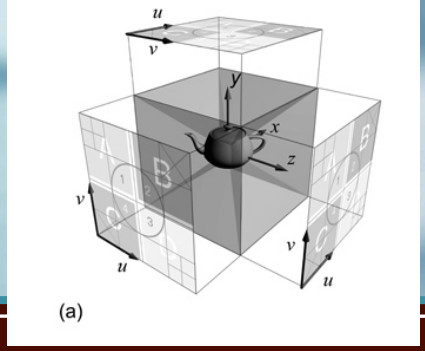
Sphere mapping computations summary



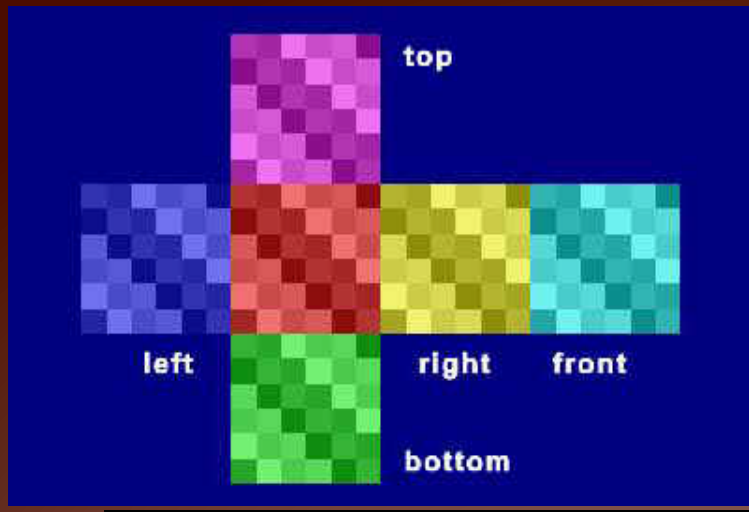
Sphere Mapping with GLSL

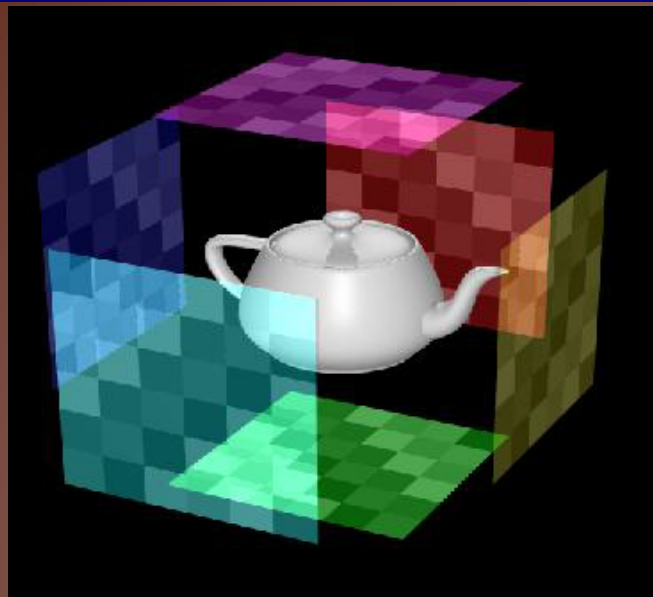
```
[Vertex_Shader]
in vec4 vPos, vNormal, vTexCoord;
out  vec2 tex_coord , sphere_coord;
uniform mat4 ModelViewMatrix, pvmMatrix;
uniform mat3 NormalMatrix;   //  inverse(transpose(mat3(ModelViewMatrix)))

void main()  {
                gl_Position = pvmMatrix * vPos;     /* Vertex Position to Clip Space */
                tex_coord = vTexCoord;
                vec3 e = normalize( vec3(ModelViewMatrix * vPos) );
                vec3 n = normalize( NormalMatrix * vNormal );
                vec3 r = reflect( e, n );
                float m = 2.0 * sqrt( r.x*r.x + r.y*r.y + (r.z+1.0)*(r.z+1.0) );
                sphere_coord.s = r.x/m + 0.5;
                sphere_coord.t = r.y/m + 0.5;
}
[Pixel_Shader]
in vec2 tex_coord , sphere_coord;
uniform sampler2D colorMap, envMap;
void main (void) {
                vec4 color = texture( colorMap, tex_coord.st);
                vec4 env = texture( envMap, sphere_coord.st);
                gl_FragColor = color + env*0.4;
}
```

Cube Map

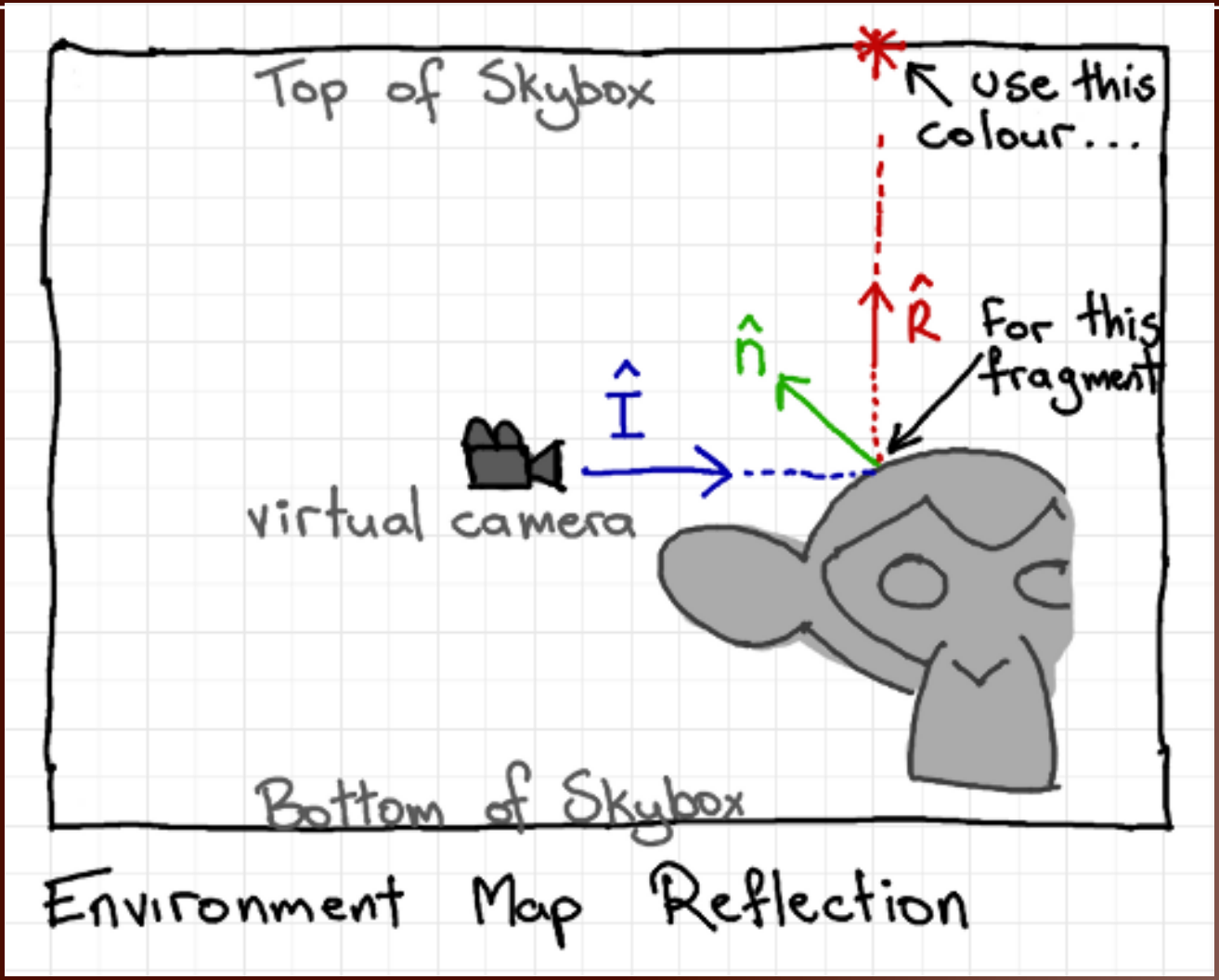


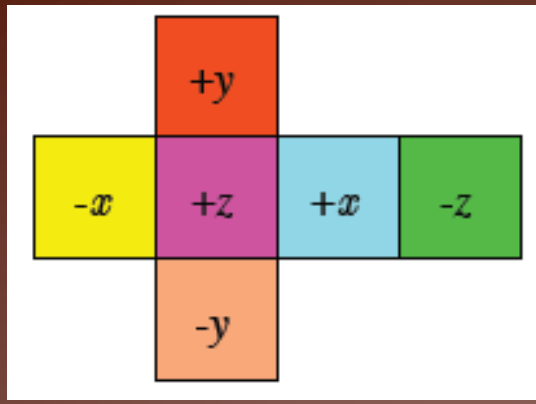
- Most popular and fastest: easy to produce with rendering system or by photography from center of object, once for each side of cube
- Simple texture-coordinates calculation
- View independent!
- “Uniform” sampling/resolution!
- Supports bilinear filtering and mipmapping!
- Get examples of cube textures at Humus' (Emil Persson's) site: http://www.humus.name/index.php?page=Textures





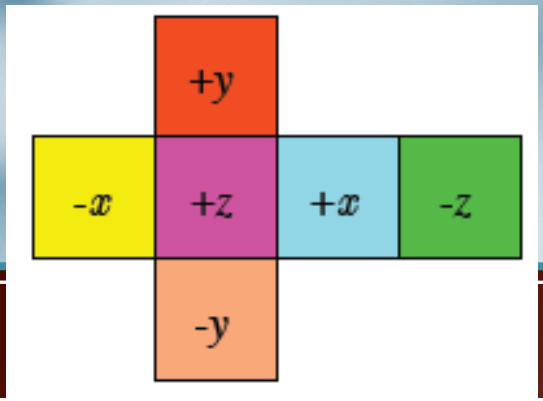
Computing Reflection





from http://antongerdelan.net/opengl/cubemaps.html

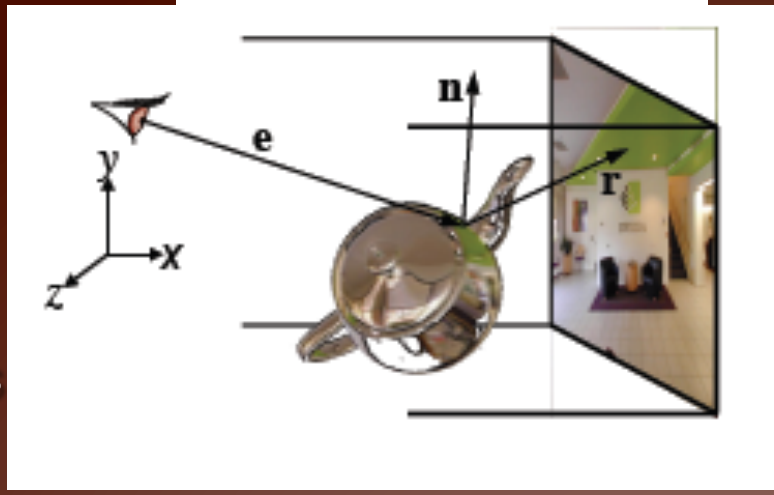
Computing Reflection

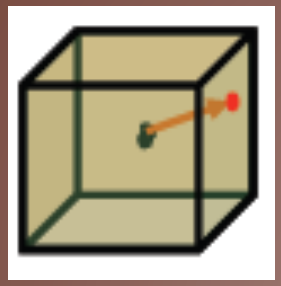


Steps:

- compute reflection vector, **r**
  - **e** from eye to vertex
  - **n** normal in eye coordinates
  - **r** = **e** – 2**n**(**n**•**e**)
- reflection is a function of one direction: largest absolute value of **r**'s components determines the **cube face** to reflect
  - example: **r** = (5, −1, 2) gives +*x* as the reflected face
- divide **r** by the largest absolute value (5) and map to [0,1] to get the texture coord in the selected face:

  (*s*, *t*) = ((*y* /*x* + *1*)/2, (*z*/*x* + *1*)/2) = ((−1/5 + 1)/2, (2/5 + 1)/2) = (0.6, 0.8)





Setting Cube Mapping in OpenGL

```
//Defining the texture object
glGenTextures(1, &cube_texture_id);
glBindTexture(GL_TEXTURE_CUBE_MAP, cube_texture_id);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGBA, texture_size,
texture_size, 0, GL_RGBA, GL_UNSIGNED_BYTE, texture[0]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGBA, texture_size,
texture_size, 0, GL_RGBA, GL_UNSIGNED_BYTE, texture[1]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGBA, texture_size,
texture_size, 0, GL_RGBA, GL_UNSIGNED_BYTE, texture[2]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGBA, texture_size,
texture_size, 0, GL_RGBA, GL_UNSIGNED_BYTE, texture[3]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGBA, texture_size,
texture_size, 0, GL_RGBA, GL_UNSIGNED_BYTE, texture[4]);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, GL_RGBA, texture_size,
texture_size, 0, GL_RGBA, GL_UNSIGNED_BYTE, texture[5]);


// using the cube texture object
glActiveTexture (GL_TEXTURE0);
glBindTexture(GL_TEXTURE_CUBE_MAP, cube_texture_id);
glUniform1i(tex_loc, 0);
```

Cube Mapping with GLSL (I)

```
[Vertex_Shader]
in vec4 vPos, vNormal, vTexCoord;
out  vec2 tex_coord ,
out vec3  reflected;
uniform mat4 ModelMatrix, ViewMatrix, ModelViewMatrix, pvmMatrix;
uniform mat3 NormalMatrix;   //  inverse(transpose(mat3(ModelViewMatrix)))

void main()  {
                gl_Position = pvmMatrix * vPos;     /* Vertex Position to Clip Space */
                 tex_coord =  vTexCoord.st;
                vec3 e = normalize( vec3(ModelViewMatrix * vPos) );
                vec3 n = normalize( NormalMatrix * vNormal.xyz );   //
                reflected = reflect( e, n );  //reflection vector in eye coord
                reflected = vec3 (inverse (ViewMatrix) * vec4 (reflected, 0.0)); //reflection vector in world coord
}
```

You can reduce the number of instructions in the shader if you **compute the reflection in world space**, but then either you need a camera world position uniform or you can get it from:

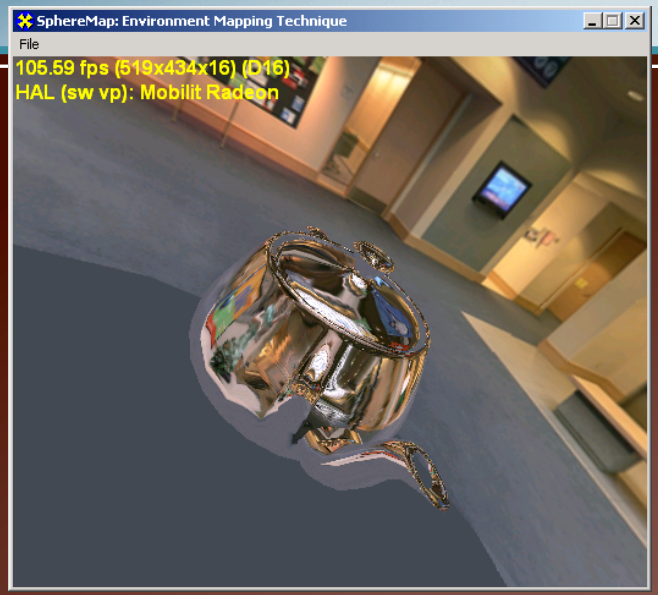
```
vec3 CameraPosition = -ViewMatrix[3].xyz * mat3(ViewMatrix);  //  Word Coord

/* Both E and N in World Coordinates so we use only the uniform ModelMatrix for the vertices and normals*/
vec3 E = normalize(vec3(ModelMatrix * vPos) - CameraPosition);
vec3 N = normalize(inverse(transpose(mat3(ModelMatrix))) * vec3(Vertex_Normal));
wReflected = reflect(E,N);
```

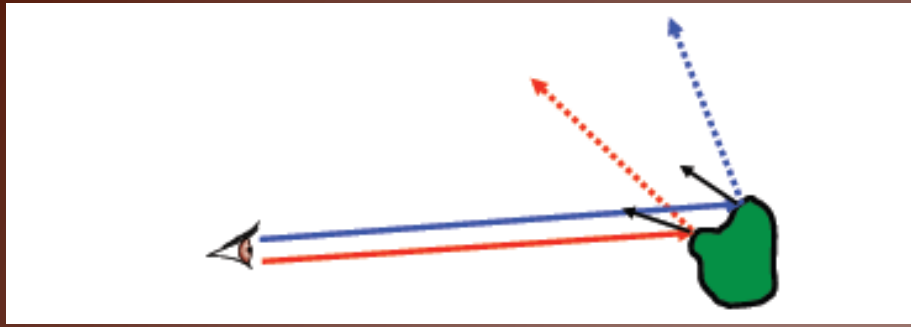
Cube Mapping with GLSL (II)

```
[Pixel_Shader]
in  vec3 reflected;
in vec2 tex_coord;

uniform samplerCube cubeMap;
uniform sampler2D colorMap;
const float reflect_factor = 0.8;

void main (void) {
        // Perform a simple 2D texture look up.
        vec3 base_color = texture2D(colorMap, tex_coord).rgb;

        // Perform a cube map look up.
        vec3 cube_color = textureCube(cubeMap, reflected);

        // Write the final pixel.
        gl_FragColor = vec4( mix(base_color, cube_color, reflect_factor), 1.0);
}
```

EM Summary

- environment is infinitely far away
- all reflections as seen from the same view point:
  - object approximated as an infinitely small,
  - perfectly mirroring ball concentric with the object
- reflected color computed only from the **direction** of reflection
  - not from the position on surface
  - no ray-env intersection computation!





Generating the Sphere Map

- Consider the circle of the environment map within the square texture to be a unit circle. For each **point** (*s*, *t*) in the unit circle, you can compute a point on the sphere (Attention: s and t varies between [-1 1]) :

$$\begin{aligned} P_x &= s \\ P_y &= t \\ P_z &= \sqrt{1.0 - P_x^2 - P_y^2} \end{aligned}$$

- $\mathbf{r} = \mathbf{e} - 2\mathbf{n}(\mathbf{n} \cdot \mathbf{e})$ where $\mathbf{e} = (0\ 0\ -1)$
- Then :

$$\begin{aligned} R_x &= 2N_xN_z \\ R_y &= 2N_yN_z \\ R_z &= 2N_zN_z - 1 \end{aligned}$$

Generating the Sphere Map

```
void gen_sphere_map(GLsizei width, GLsizei height, GLfloat pos[3],
                    GLfloat (*tex)[3])
{
  GLfloat ray[3], color[3], p[3], s, t;
  int i, j;

  for (j = 0; j < height; j++) {
    t = 2.0 * ((float)j / (float)(height-1) - .5);
    for (i = 0; i < width; i++) {
      s = 2.0 * ((float)i / (float)(width - 1) - .5);

      if (s*s + t*t > 1.0) continue;

      /* compute the point on the sphere (aka the normal) */
      p[0] = s;
      p[1] = t;
      p[2] = sqrt(1.0 - s*s - t*t);

      /* compute reflected ray */
      ray[0] = p[0] * p[2] * 2;
      ray[2] = p[1] * p[2] * 2;
      ray[3] = p[2] * p[2] * 2 - 1;
      fire_ray(pos, ray, tex[j*width + i]);
    }
  }
}
```

References

- http://antongerdelan.net/opengl/cubemaps.html
- http://web.eecs.umich.edu/~sugih/courses/eecs487/lectures/28-EM.pdf
- http://www.cse.ohio-state.edu/~hwshen/781/Site/Slides_files/env.pdf