ARLAB

ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

Environment Mapping

October 20th, 2015

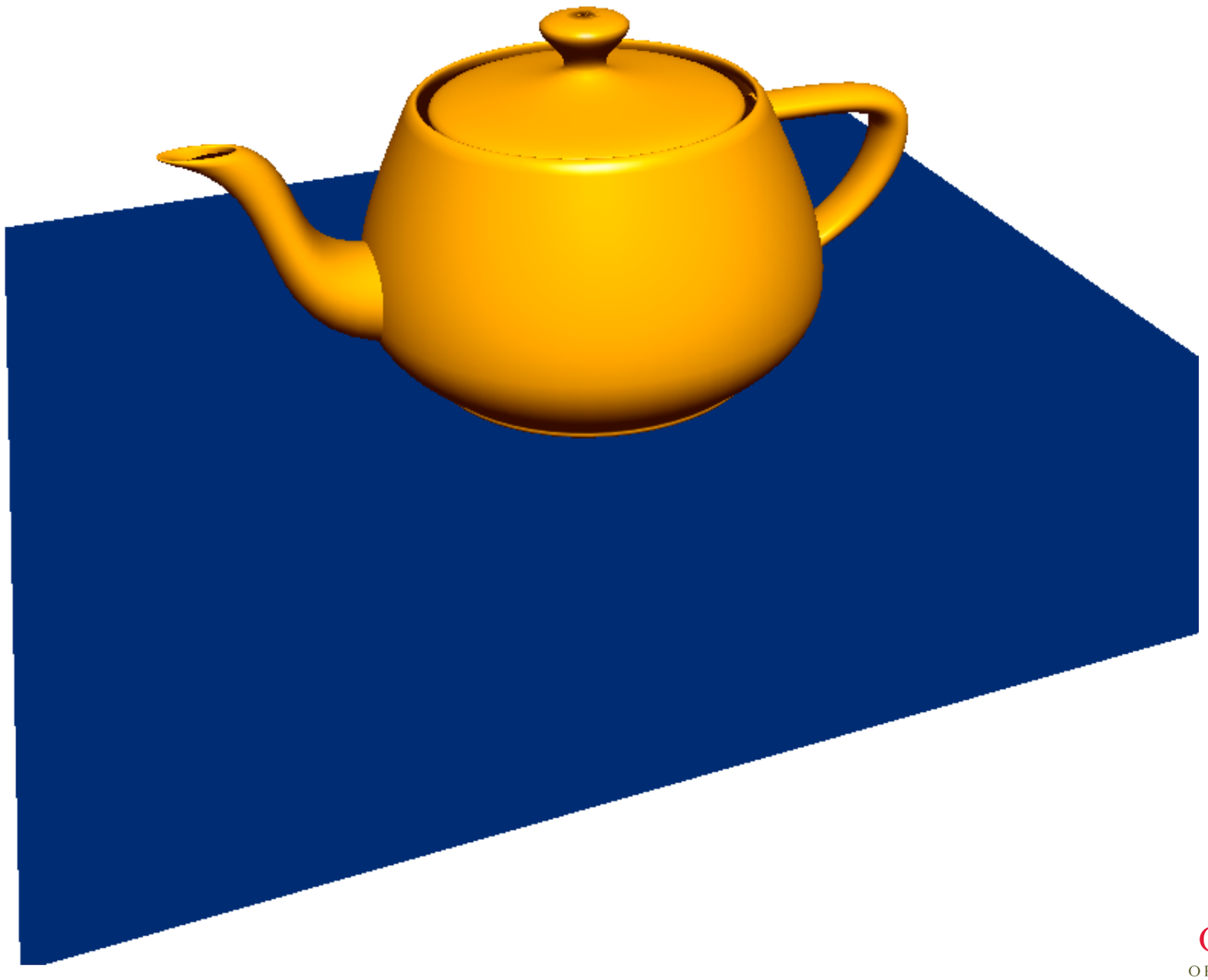
Rafael Radkowski

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Content

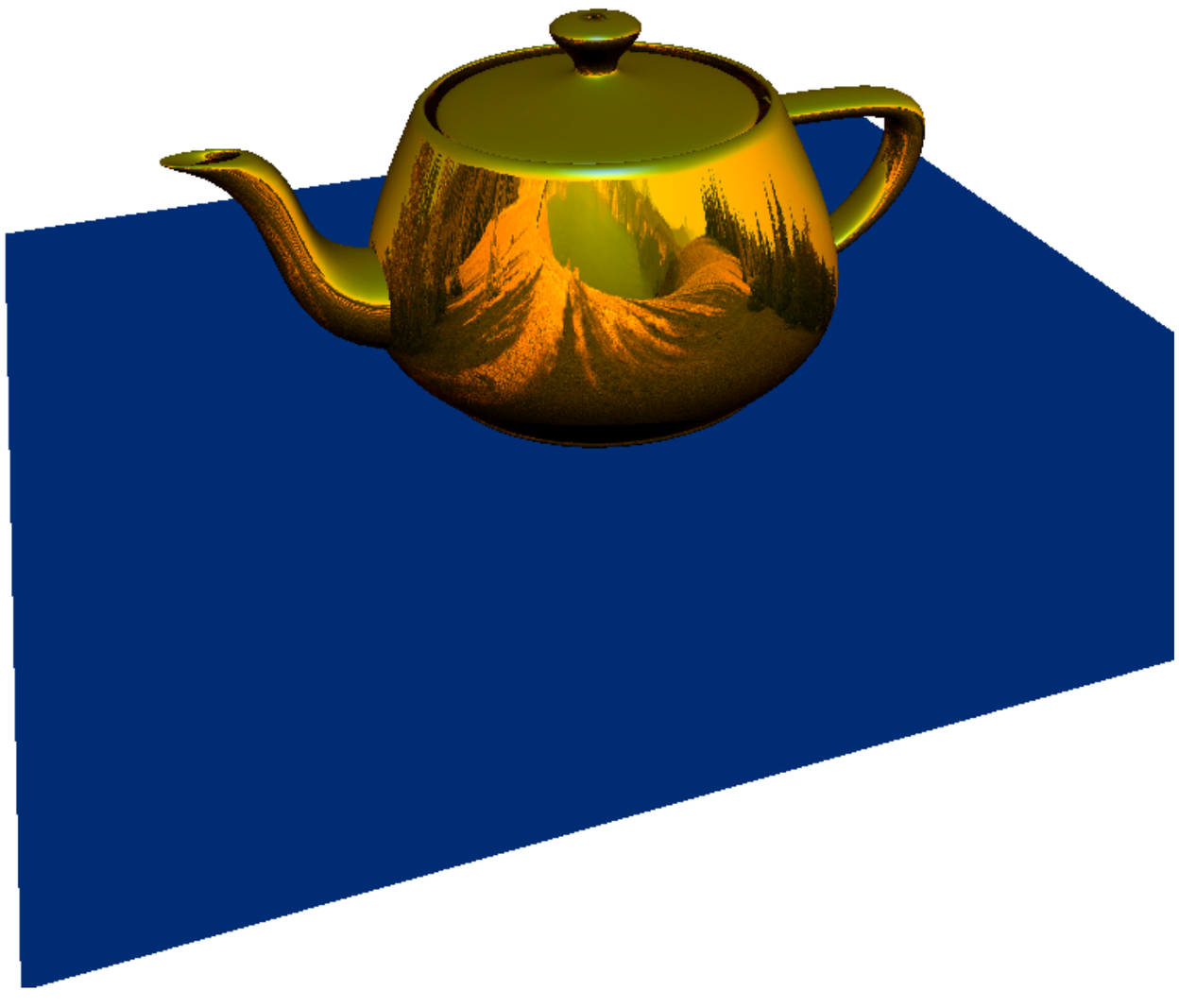
- Environment Mapping
- Sphere Maps in OpenGL

Environment Mapping



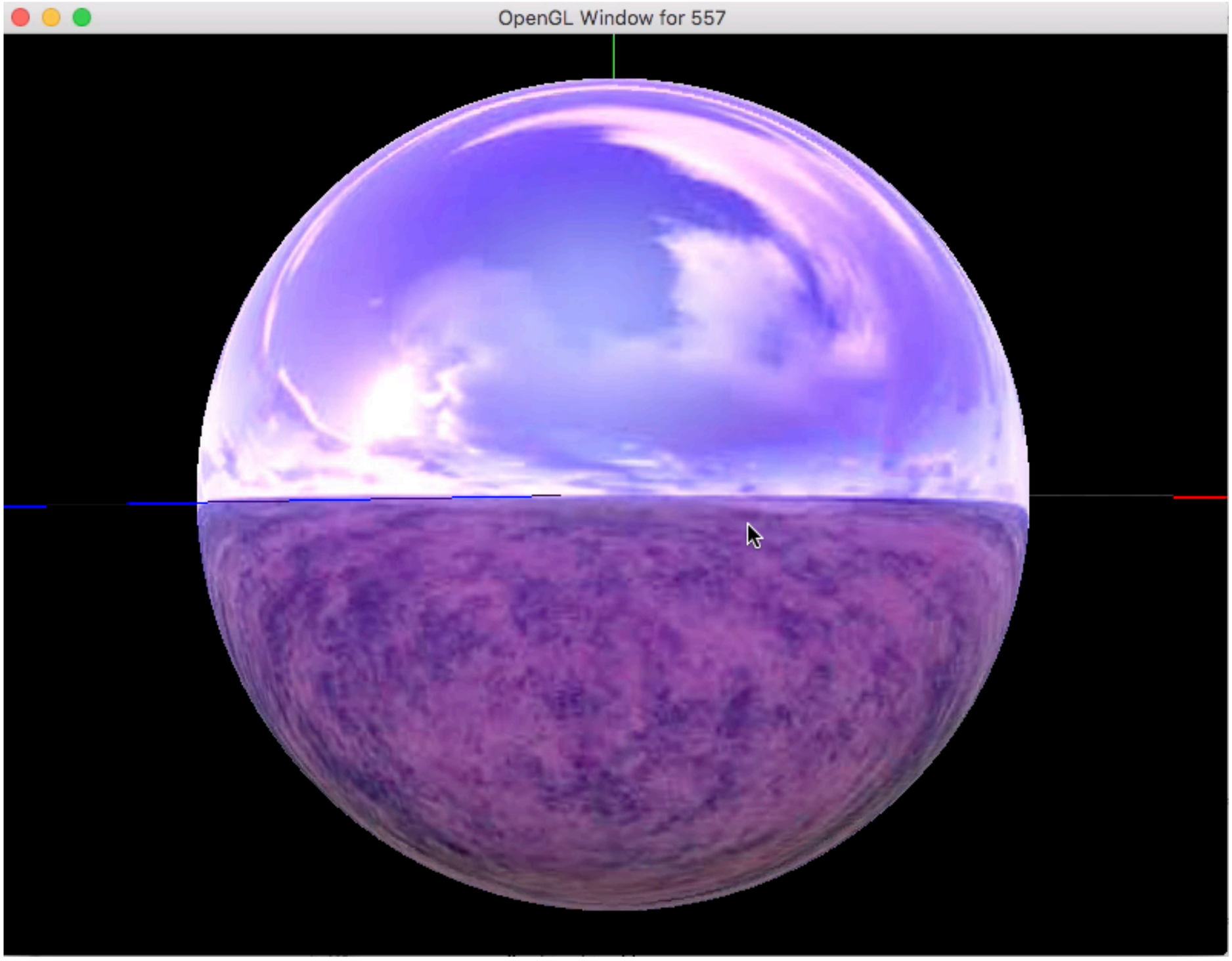
Environment Mapping

VRAC | HCI

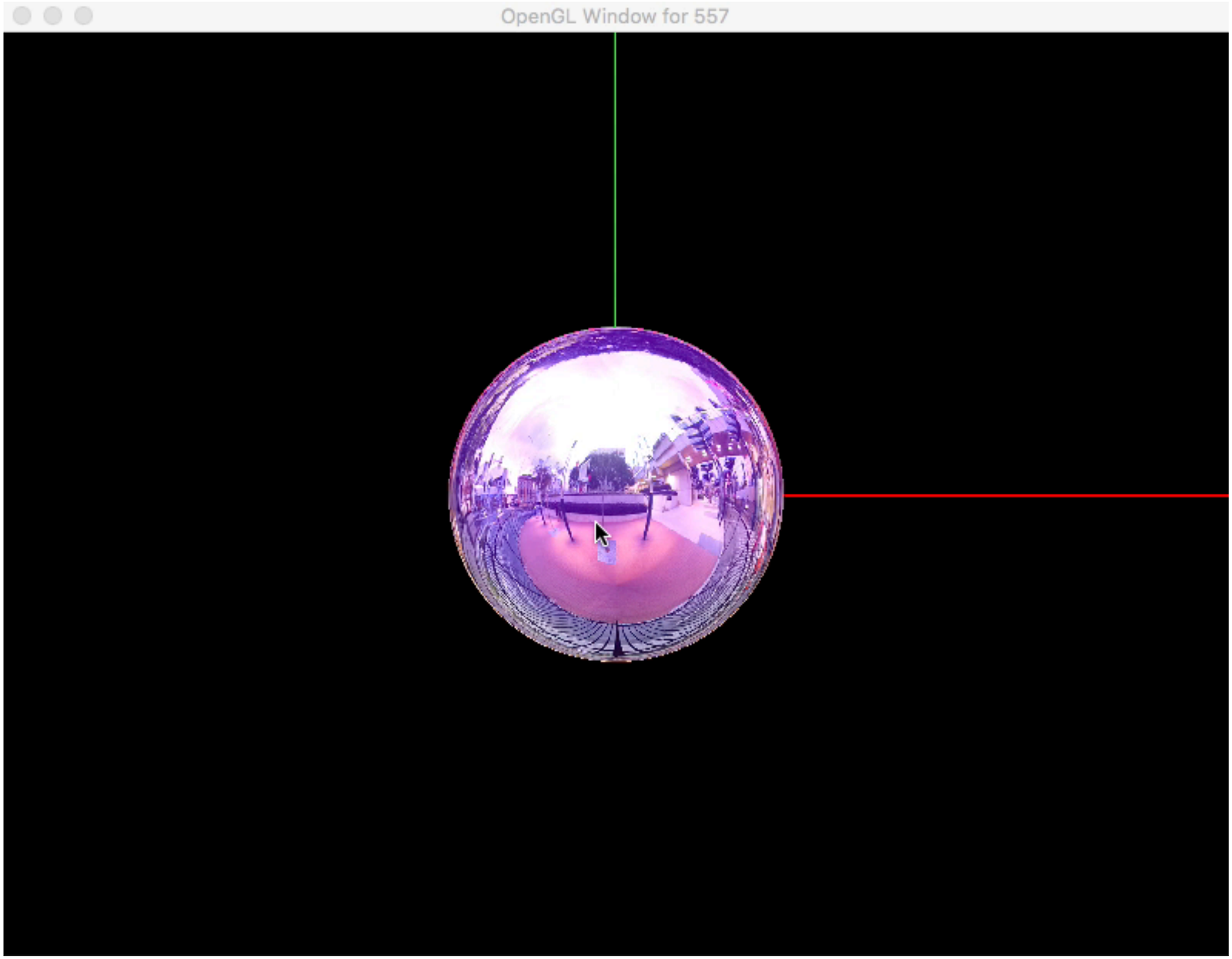


Video

ARLAB



Video



Concept of Environment Mapping

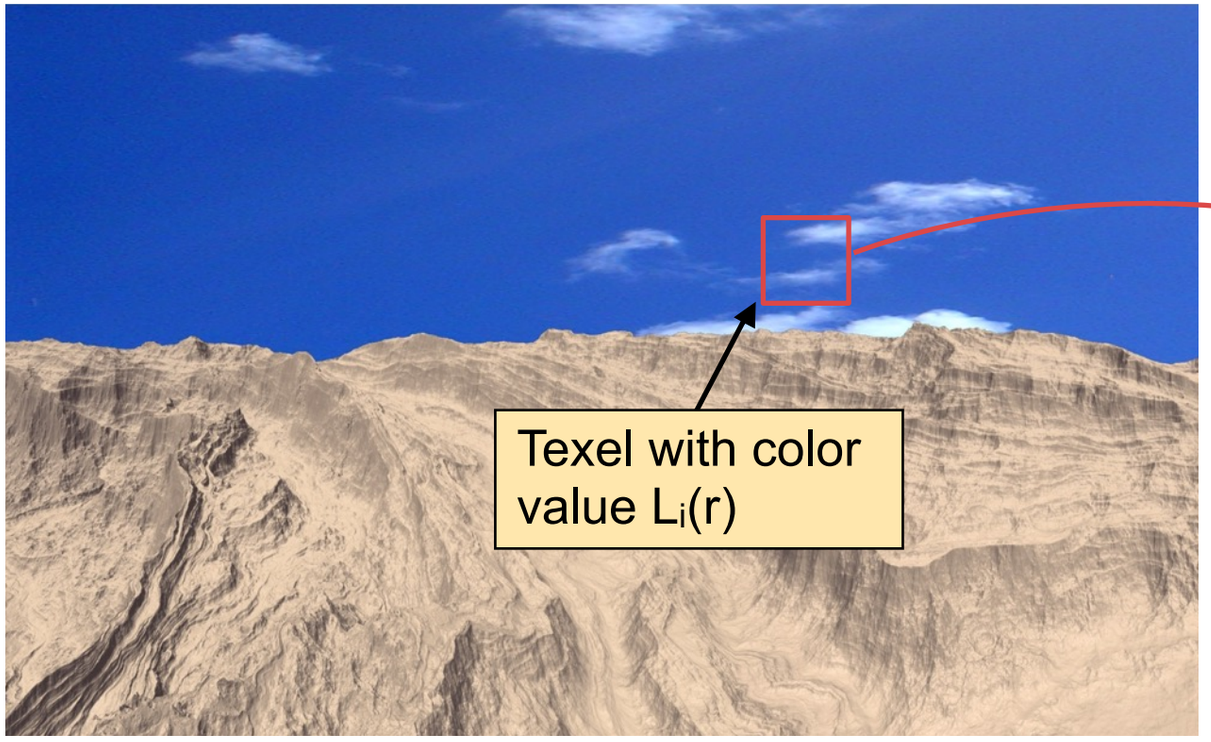
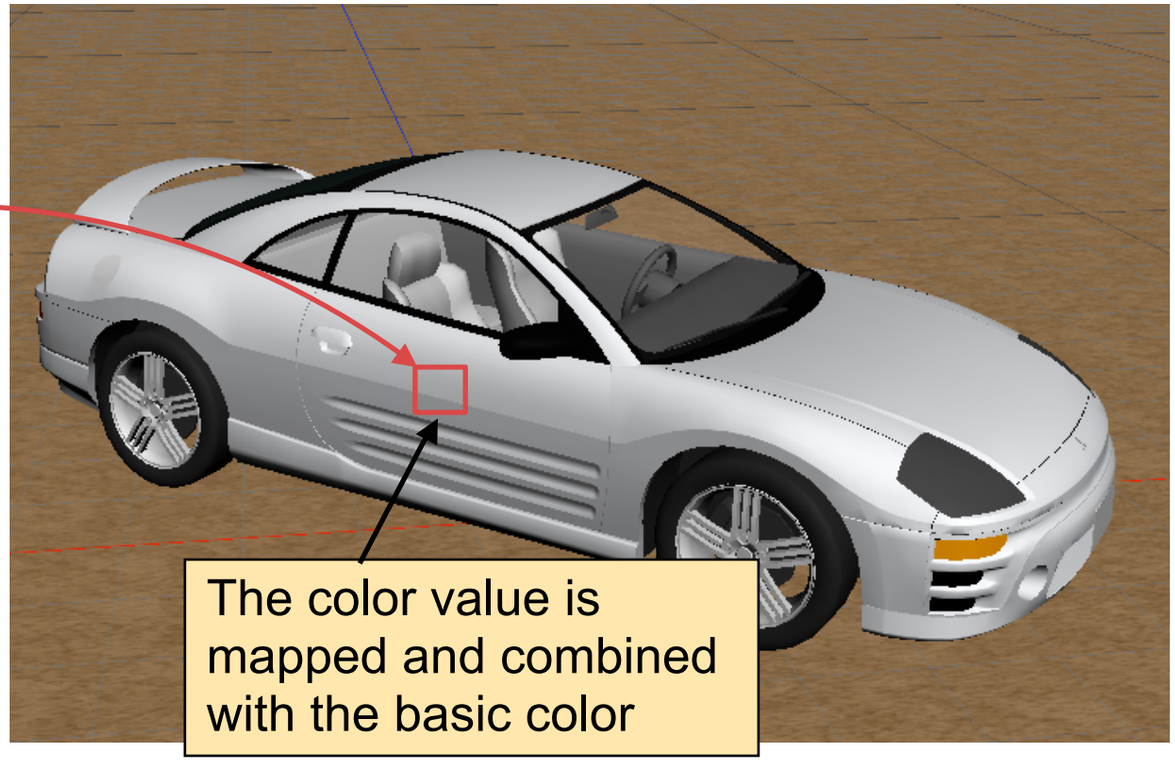


Image of an environment model



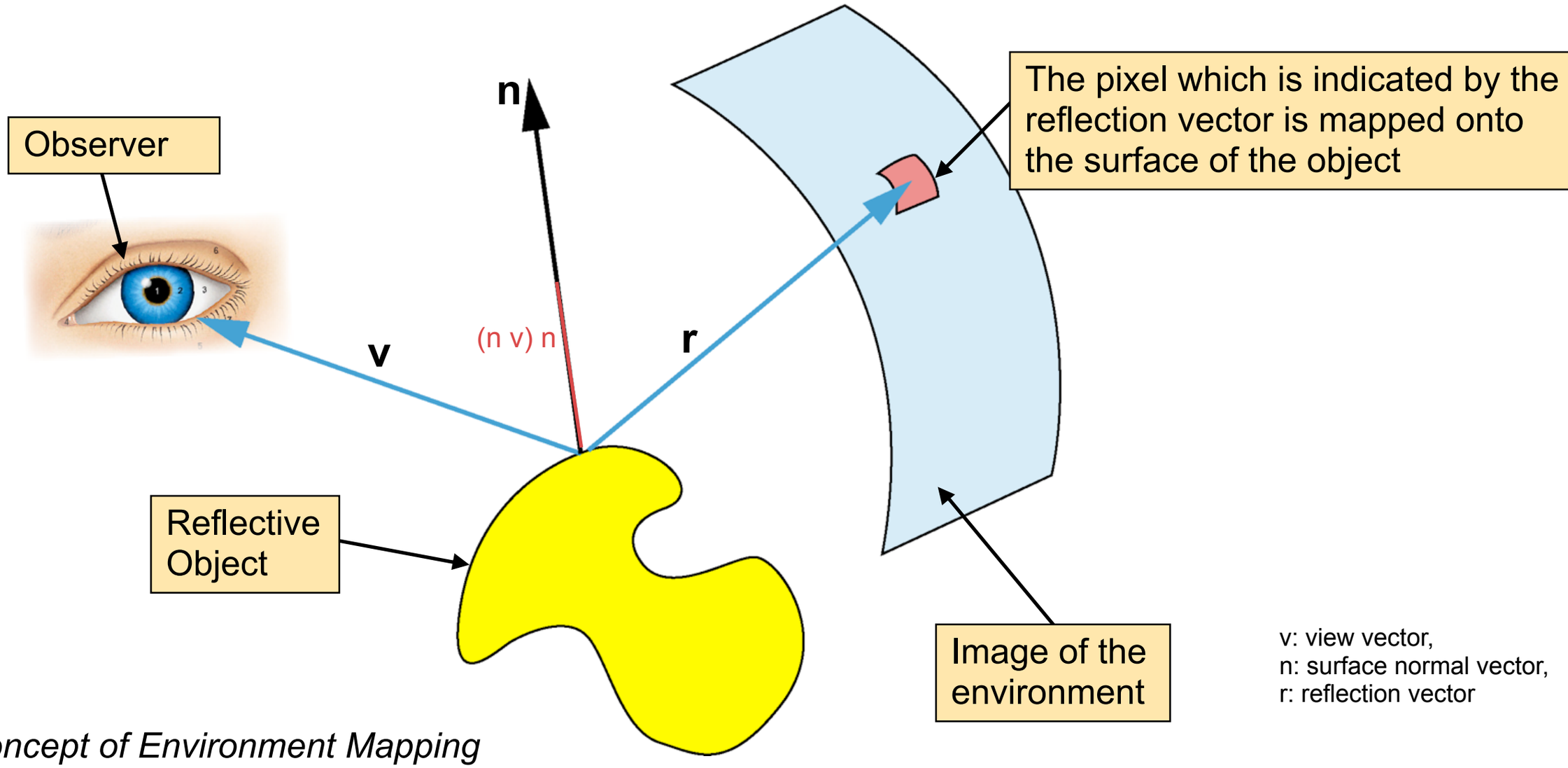
When the reflection vector ***r*** is estimated, the nearest color value L_i is fetched from the texture. In this case, the color value is considered as light information and combined with the basic color. Different operators can be use, but, for a regular reflection, basic color and light must be multiplied.

$$L_O(\vec{v}) = R(\Theta) \cdot L_i(\vec{r})$$

Limitations

- the environment is far away from the object
- the object does not emit light.

Concept of Environment Mapping



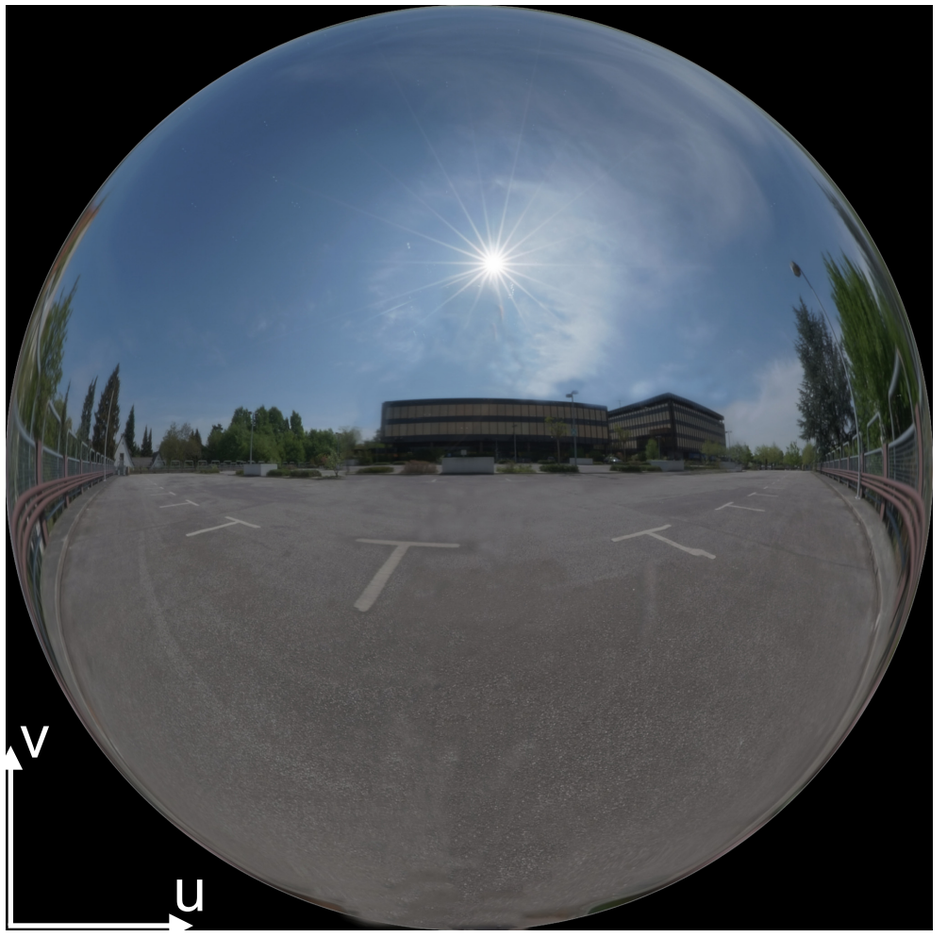
Concept of Environment Mapping

Environment Mapping uses the specular reflection equation to indicate an image point in an environment image. Therefore, the reflection vector **r** needs to be calculated with respect to the viewing vector **v**:

$$r = 2(\vec{n} \cdot \vec{v}) \cdot \vec{n} - \vec{v}$$

nach: AKENINE–MÖLLER, T.; HAINES, E.; HOFFMAN, N., Real-Time Rendering, A.K. Peters, Ltd, 2. Edition, 2002

Environment Image



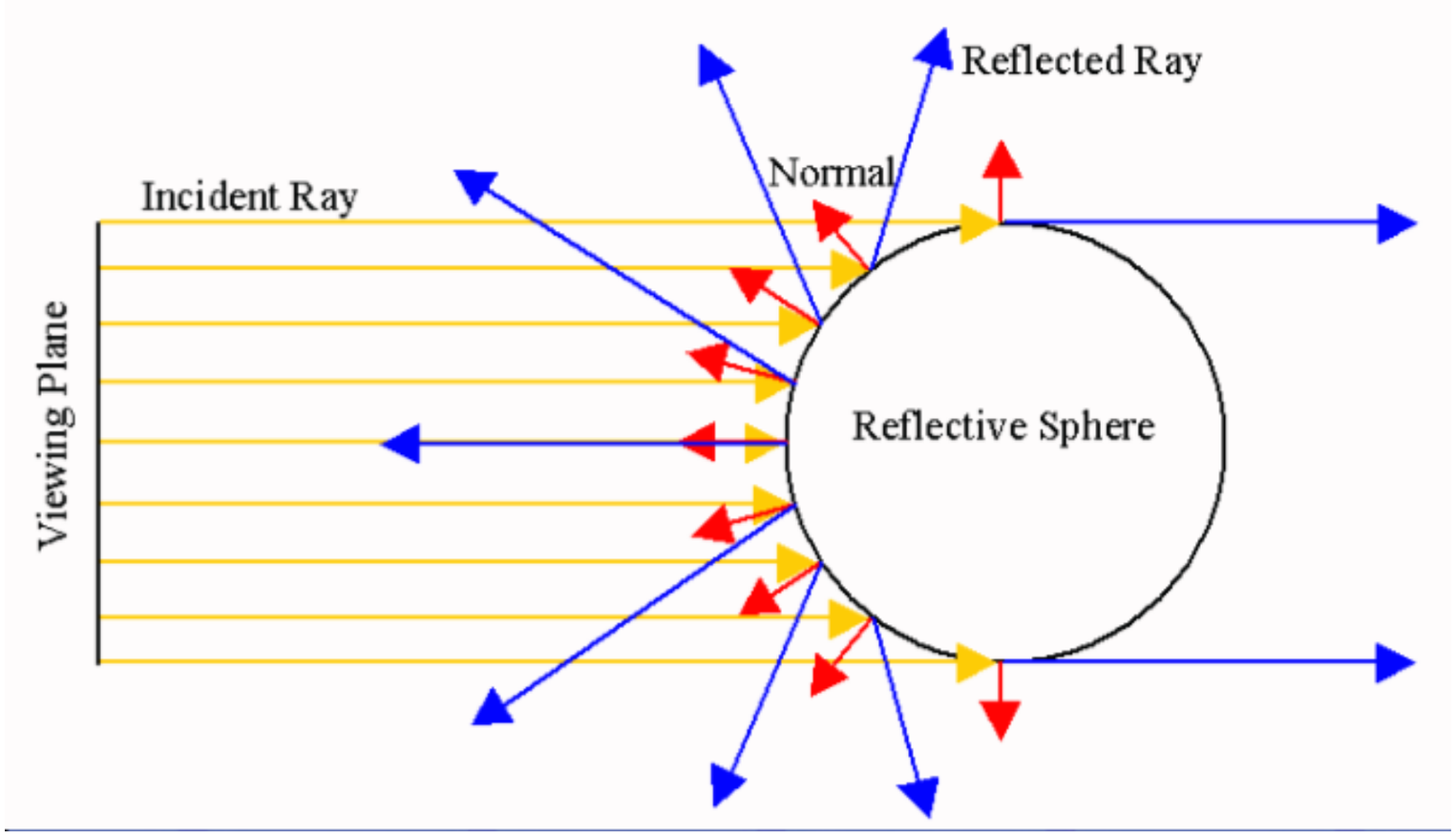
A sphere map

A sphere map can be created by taking a picture of a reflective sphere

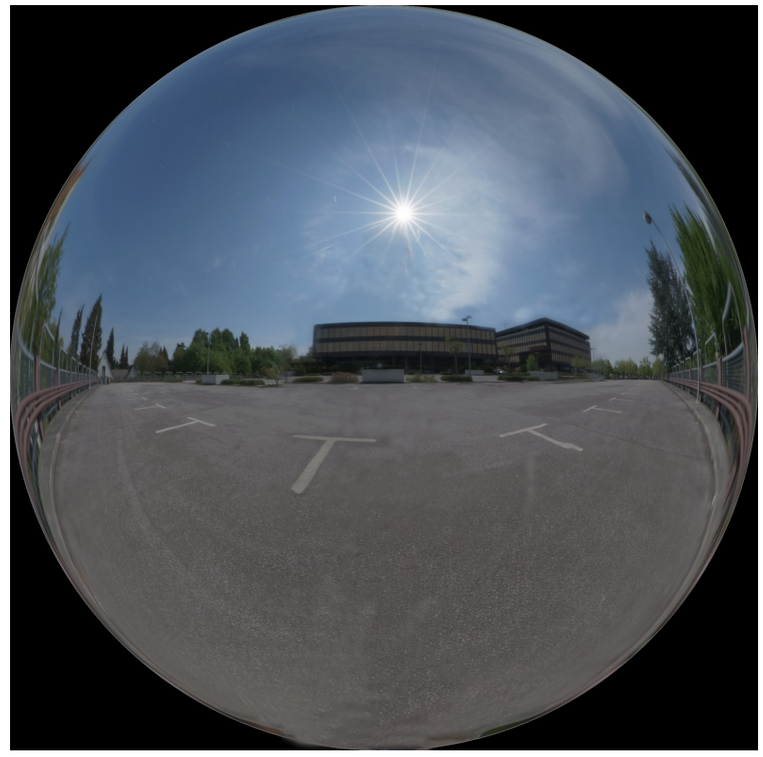
An image for an environment map can be generated by taking a picture of an reflective sphere. A reflective sphere shows an omnidirectional view of the environment.

Unit Coordinates of a Sphere

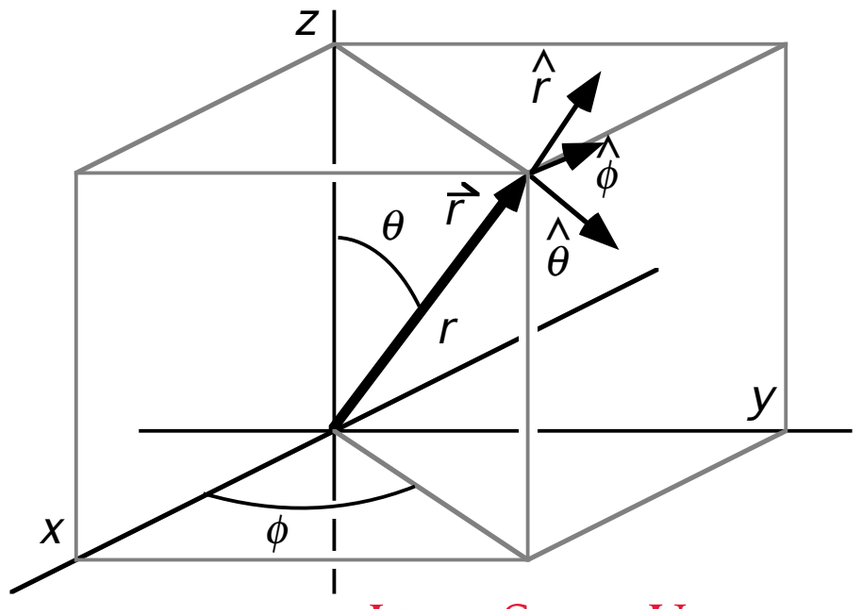
ARLAB



Paul Debevec, Environment Maps and Tone Mapping

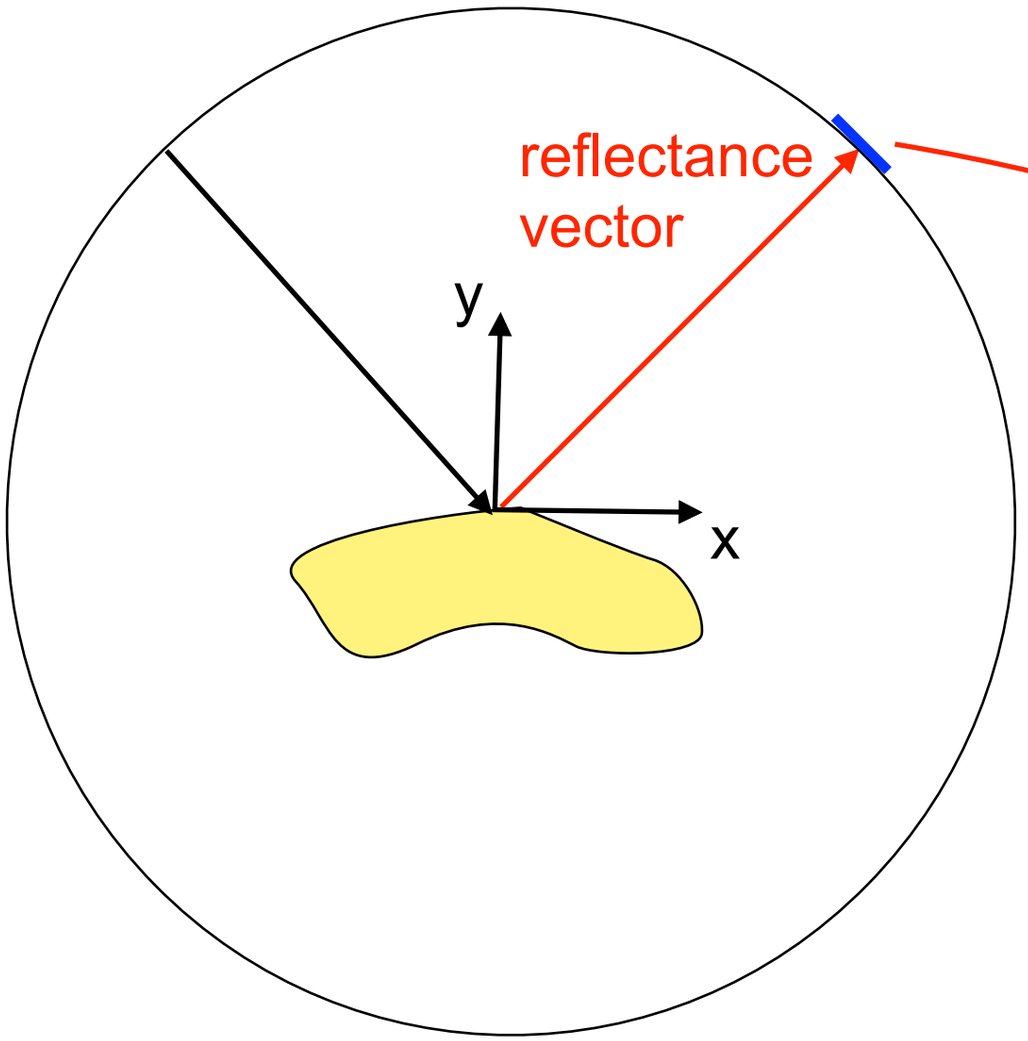


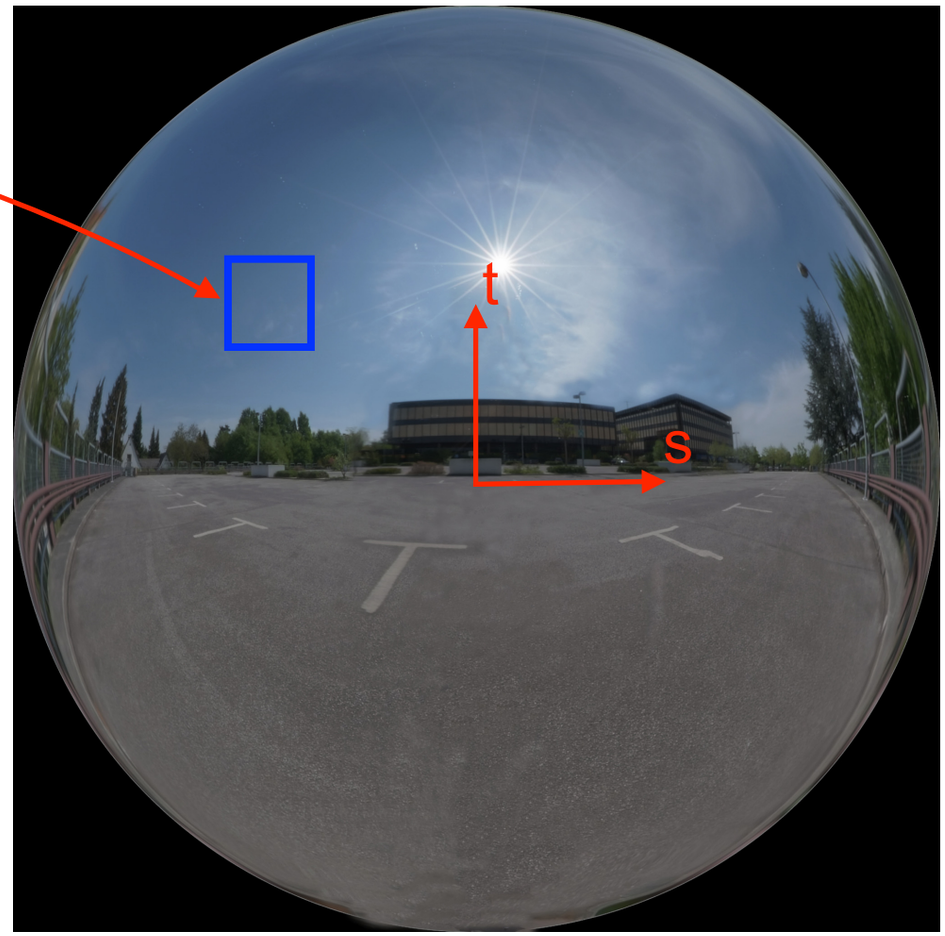
$$\hat{r} = \frac{\vec{r}}{r} = \frac{x\hat{x} + y\hat{y} + z\hat{z}}{r} = \hat{x}\sin\theta\cos\phi + \hat{y}\sin\theta\sin\phi + \hat{z}\cos\theta$$



Find the texture coordinate

ARLAB





$$(s,t) = \left(\frac{x}{2\sqrt{x^2+y^2+(z+1)^2}} + 0.5, \frac{y}{2\sqrt{x^2+y^2+(z+1)^2}} + 0.5 \right)$$

http://www.pauldebevec.com/ReflectionMapping/
http://www.pauldebevec.com/ReflectionMapping/miller.html

Limitations

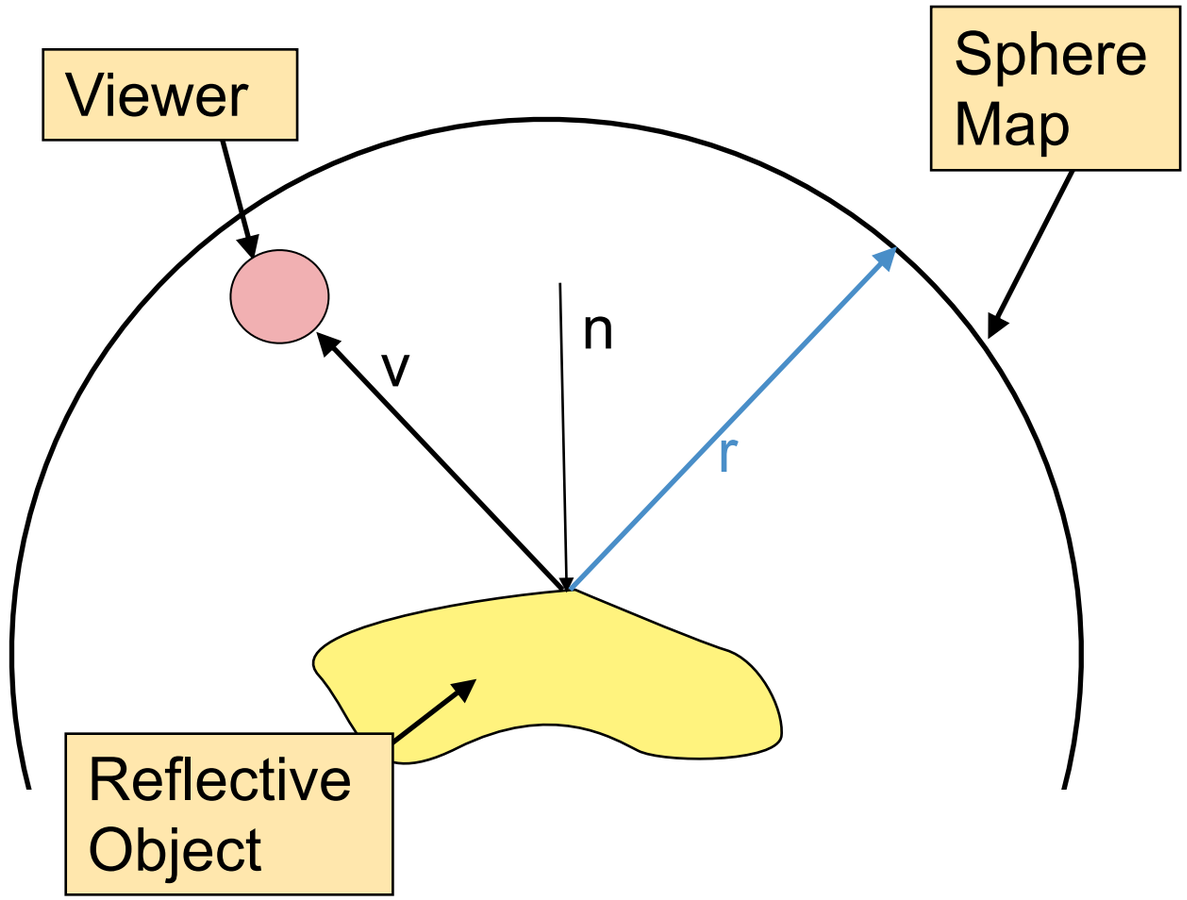
- The environment is far away from the object. Every point is in the center of the sphere which is only true if all points in the environment are far away.
- The object does not emit light.

ARLAB

Environment Mapping with OpenGL

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Sphere Map Process



Transformation from world to sphere map coordinates.

During initialization:

- prepare spherical texture
- *Texture coordinates are not required; we calculate them.*

During runtime:

- Calculate the reflectance vector
- Transfer the vector in texture coordinates (u,v).
- Map the color on the surface fragment.

Prepare a Spherical Texture

ARLAB

Follow the regular texture generation process. There is nothing special here.

```
// Generate a texture
glGenTextures(1, &_texture );

// Set a texture as active texture.
glBindTexture( GL_TEXTURE_2D, _texture );

// Change the parameters of your texture
glTexParameterf( GL_TEXTURE_2D, ... );
glTexParameterf( GL_TEXTURE_2D, ... );
glTexParameterf( GL_TEXTURE_2D, ... );
glTexParameterf( GL_TEXTURE_2D, ... );

// Create a texture
if(channels == 3)
    glTexImage2D(GL_TEXTURE_2D, 0, .....
else if(channels == 4)
    glTexImage2D(GL_TEXTURE_2D, 0, .....
```

That is all what need to be done in your host program.

GLSL Fragment Shader

ARLAB

```
void main(void)
{
    // Caculate the normal vector and surface position in world coordinates
    vec3 normal = normalize(in_Normal);
    vec4 transformedNormal =  normalize(transpose(inverse(modelMatrixBox)) * vec4( normal, 1.0 ));
    vec4 surfacePosition = modelMatrixBox * vec4(in_Position, 1.0);

    // Calculate the color
    vec4 linearColor = vec4(0.0,0.0,0.0,0.0);
    for (int i=0; i<numLights; i++) {
        vec4 new_light = useLight(allLights[i], surfacePosition, transformedNormal, normal, allMaterials[0] );
        linearColor = linearColor + new_light;
    }

    //-------------------------------------------------------------------------
    // Environment mapping
    vec3 cameraPosition = vec3( inverseViewMatrix[3][0], inverseViewMatrix[3][1], inverseViewMatrix[3][2]);
    vN = spherical_mapping(cameraPosition, surfacePosition.xyz, normal.xyz);


    // Gamma correction
    vec4 gamma = vec4(1.0/2.2);
    vec4 finalColor = pow(linearColor, gamma);

    // Pass the color
    pass_Color =  vec4(finalColor);

    // Passes the projected position to the fragment shader / rasterization process.
    gl_Position = projectionMatrixBox * viewMatrixBox * modelMatrixBox * vec4(in_Position, 1.0);

    // Passes the texture coordinates to the next pipeline processes.
    pass_TexCoord = in_TexCoord;

}
```

GLSL Fragment Shader

ARLAB

```
void main(void)
{
    // Caculate the normal vector and surface position in world coordinates
    vec3 normal = normalize(in_Normal);
    vec4 transformedNormal =  normalize(transpose(inverse(modelMatrixBox)) * vec4( normal, 1.0 ));
    vec4 surfacePosition = modelMatrixBox * vec4(in_Position, 1.0);
```

Light calculation

```
);
```

```
//-------------------------------------------------------------------------
// Environment mapping
vec3 cameraPosition = vec3( inverseViewMatrix[3][0], inverseViewMatrix[3][1], inverseViewMatrix[3][2]);
vN = spherical_mapping(cameraPosition, surfacePosition.xyz, normal.xyz);
```

Position and variables to pass to the fragment shader

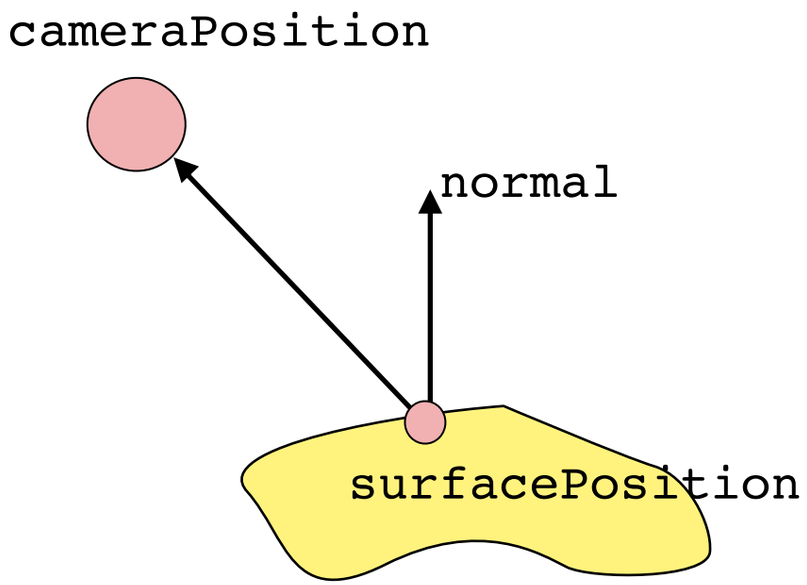
```
    // Passes the texture coordinates to the next pipeline processes.
    pass_TexCoord = in_TexCoord;

}
```

GLSL Fragment Shader

ARLAB

```
void main(void)
{
    // Caculate the normal vector and surface position in world coordinates
    vec3 normal = normalize(in_Normal);

[...]

    vec4 surfacePosition = modelMatrixBox * vec4(in_Position, 1.0);

[...]

    //-----------------------------------------------------------------------
    // Environment mapping
    vec3 cameraPosition = vec3( inverseViewMatrix[3][0],
                                inverseViewMatrix[3][1],
                                inverseViewMatrix[3][2]);

    vN = spherical_mapping(cameraPosition, surfacePosition.xyz, normal.xyz);

[...]

    // Passes the texture coordinates to the next pipeline processes.
    pass_TexCoord = in_TexCoord;

}
```

Vectors



First calculate all the points and vectors:

- camera position
- normal vector (normalized vector in world coordinates)
- surface position (vertex coordinate in world coordinates.)

```
// Caculate the normal vector and surface position in world coordinates
vec3 normal = normalize(in_Normal);


vec4 surfacePosition = modelMatrixBox * vec4(in_Position, 1.0);


vec3 cameraPosition = vec3( inverseViewMatrix[3][0],
                            inverseViewMatrix[3][1],
                            inverseViewMatrix[3][2]);
```

modelMatrixBox: this is the model matrix, inverseViewMatrix: literally, the inverse view matrix.

Sphere Mapping

ARLAB

```
vec2 spherical_mapping(vec3 cameraPosition, vec3 surfacePosition, vec3 normal)
{
    vec2 sphericalUV = vec2(0.0f, 0.0f);

    // calculate the eye-to-surface vector
    vec3 eye_to_surface = normalize(surfacePosition - cameraPosition);

    // calculate the reflectance vector
    vec3 r = normalize(reflect( eye_to_surface, normal ));

    // calculate the position of the texel
    float m = 2. * sqrt(
            pow( r.x, 2. ) +
            pow( r.y, 2. ) +
            pow( r.z + 1., 2. ));

    sphericalUV = r.xy / m + .5;

    return sphericalUV;
}
```

Sphere Mapping

ARLAB

```
vec2 spherical_mapping(vec3 cameraPosition, vec3 surfacePosition, vec3 normal)
{
    vec2 sphericalUV = vec2(0.0f, 0.0f);
      // calculate the eye-to-surface vector
      vec3 eye_to_surface = normalize(surfacePosition - cameraPosition);

      // calculate the reflectance vector
      vec3 r = normalize(reflect( eye_to_surface, normal ));

      // calculate the position of the texel
      float m = 2. * sqrt(
              pow( r.x, 2. ) +
              pow( r.y, 2. ) +
              pow( r.z + 1., 2. ));

      sphericalUV = r.xy / m + .5;
    return sphericalUV;
}
```

Goal: to find this coordinate

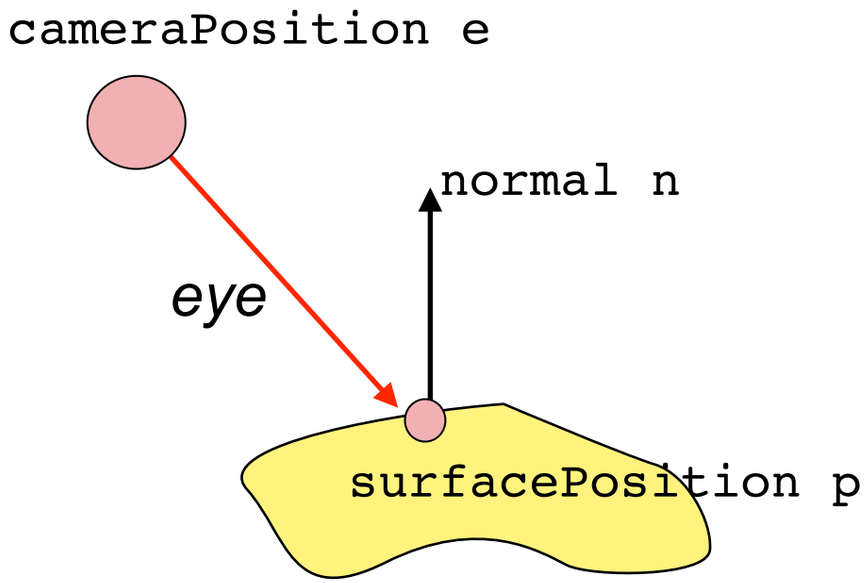
Sphere Mapping

ARLAB

```
vec2 spherical_mapping(vec3 cameraPosition, vec3 surfacePosition, vec3 normal)
{
    vec2 sphericalUV = vec2(0.0f, 0.0f);

    // calculate the eye-to-surface vector
    vec3 eye_to_surface = normalize(surfacePosition - cameraPosition);

    // calculate the reflectance vector
    vec3 r = normalize(reflect( eye_to_surface, normal ));

    // calculate the position of the texel
    float m = 2. * sqrt(
            pow( r.x, 2. ) +
            pow( r.y, 2. ) +
            pow( r.z + 1., 2. ));

    sphericalUV = r.xy / m + .5;

    return sphericalUV;
}
```

Calculate the incident vector

Sphere Mapping



- Calculate the incident vector: the incoming light vector.

$$e\vec{y}e = \vec{p} - \vec{e}$$

- Note, we inverse the camera vector because the following vector requires the indicated direction.

```
// calculate the eye-to-surface vector
vec3 eye_to_surface = normalize(surfacePosition - cameraPosition);
```

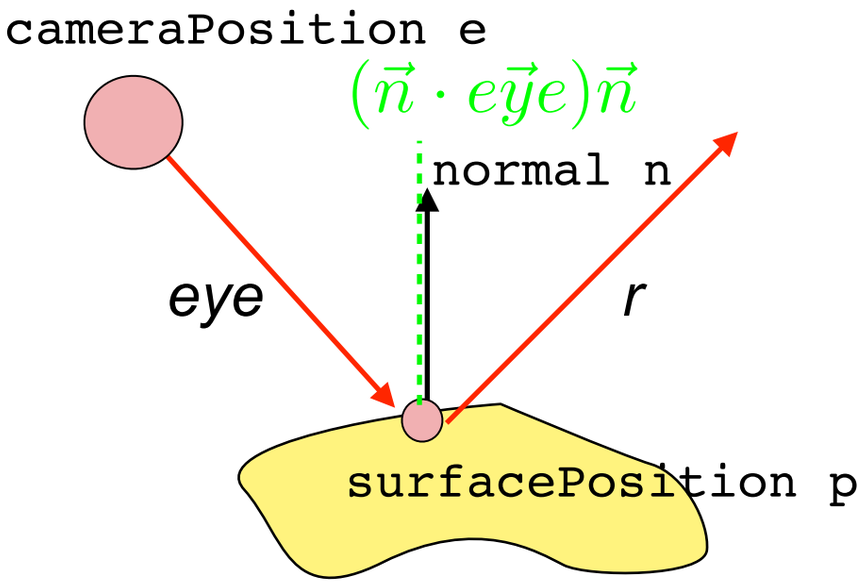
Sphere Mapping

ARLAB

```
vec2 spherical_mapping(vec3 cameraPosition, vec3 surfacePosition, vec3 normal)
{
    vec2 sphericalUV = vec2(0.0f, 0.0f);

    // calculate the eye-to-surface vector
    vec3 eye_to_surface = normalize(surfacePosition - cameraPosition);

    // calculate the reflectance vector
    vec3 r = normalize(reflect( eye_to_surface, normal ));

    // calculate the position of the texel
    float m = 2. * sqrt(
            pow( r.x, 2. ) +
            pow( r.y, 2. ) +
            pow( r.z + 1., 2. ));

    sphericalUV = r.xy / m + .5;

    return sphericalUV;
}
```

Calculate the reflectance vector

Sphere Mapping



- Calculate the reflectance vector

$$\vec{r} = 2(\vec{n} \cdot e\vec{y}e)\vec{n} - e\vec{y}e$$

```
// calculate the reflectance vector
vec3 r = normalize(reflect( eye_to_surface, normal ));
```

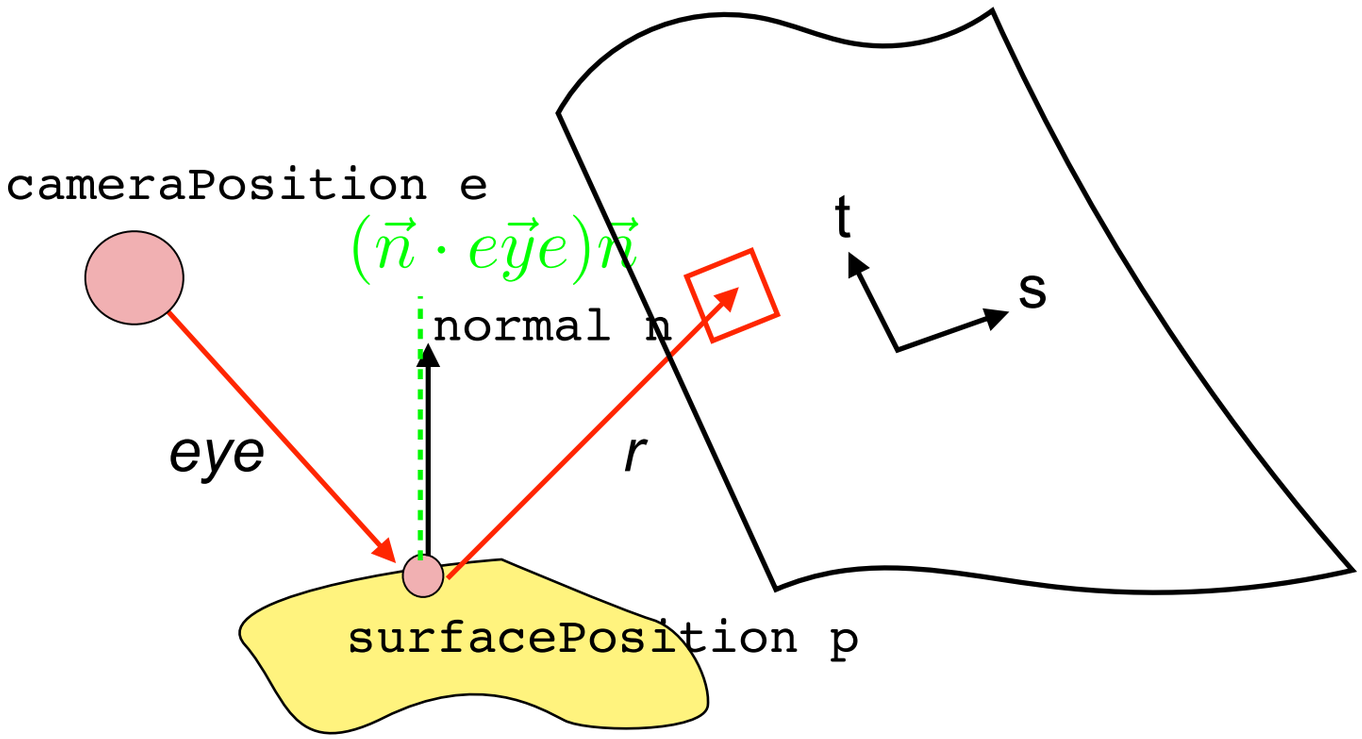
Sphere Mapping

ARLAB

```
vec2 spherical_mapping(vec3 cameraPosition, vec3 surfacePosition, vec3 normal)
{
    vec2 sphericalUV = vec2(0.0f, 0.0f);

    // calculate the eye-to-surface vector
    vec3 eye_to_surface = normalize(surfacePosition - cameraPosition);

    // calculate the reflectance vector
    vec3 r = normalize(reflect( eye_to_surface, normal ));

    // calculate the position of the texel
    float m = 2. * sqrt(
            pow( r.x, 2. ) +
            pow( r.y, 2. ) +
            pow( r.z + 1., 2. ));

    sphericalUV = r.xy / m + .5;

    return sphericalUV;
}
```

Find the texel location

Sphere Mapping



Calculate the u,v (also called s,t) coordinates

$$(s,t) = \left(\frac{x}{2\sqrt{x^2+y^2+(z+1)^2}} + 0.5, \frac{y}{2\sqrt{x^2+y^2+(z+1)^2}} + 0.5 \right)$$

```
//-------------------------------------------------------------------------
// FOR ENVIRONMENT MAPPING
// the vector from which we fetch our color component
// for environment mapping.
in vec2 vN;


void main(void)
{
    // This function finds the color component for each texture coordinate.
    vec4 tex_color =  texture(tex, vN);

     // This mixes the background color with the texture color.
     // The GLSL shader code replaces the former environment. It is now up to us
     // to figure out how we like to blend a texture with the background color.
     if(texture_blend == 0)
     {
         color = 0.5 * pass_Color + 0.5 * tex_color;
     }
     else if(texture_blend == 1)
     {
         color = pass_Color * tex_color;
     }
     else if(texture_blend == 2)
     {
         color = (1-pass_Color.w)*pass_Color + tex_color;
     }
     else
     {
         color = pass_Color + tex_color;
     }

}
```

Take the sphere map and the coordinates.

Mix the color information

Fragment Shader Program

Texture Modes

VRAC | HCI

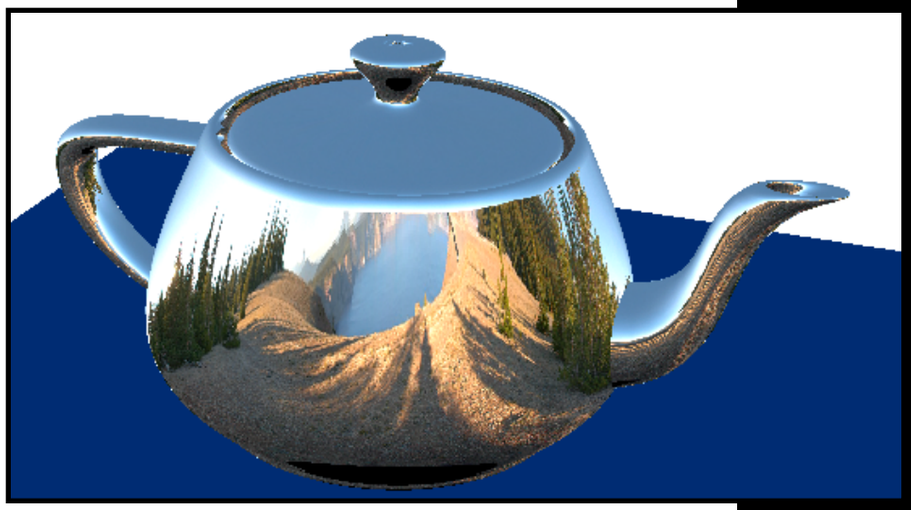
Modulate

Blend

Decal

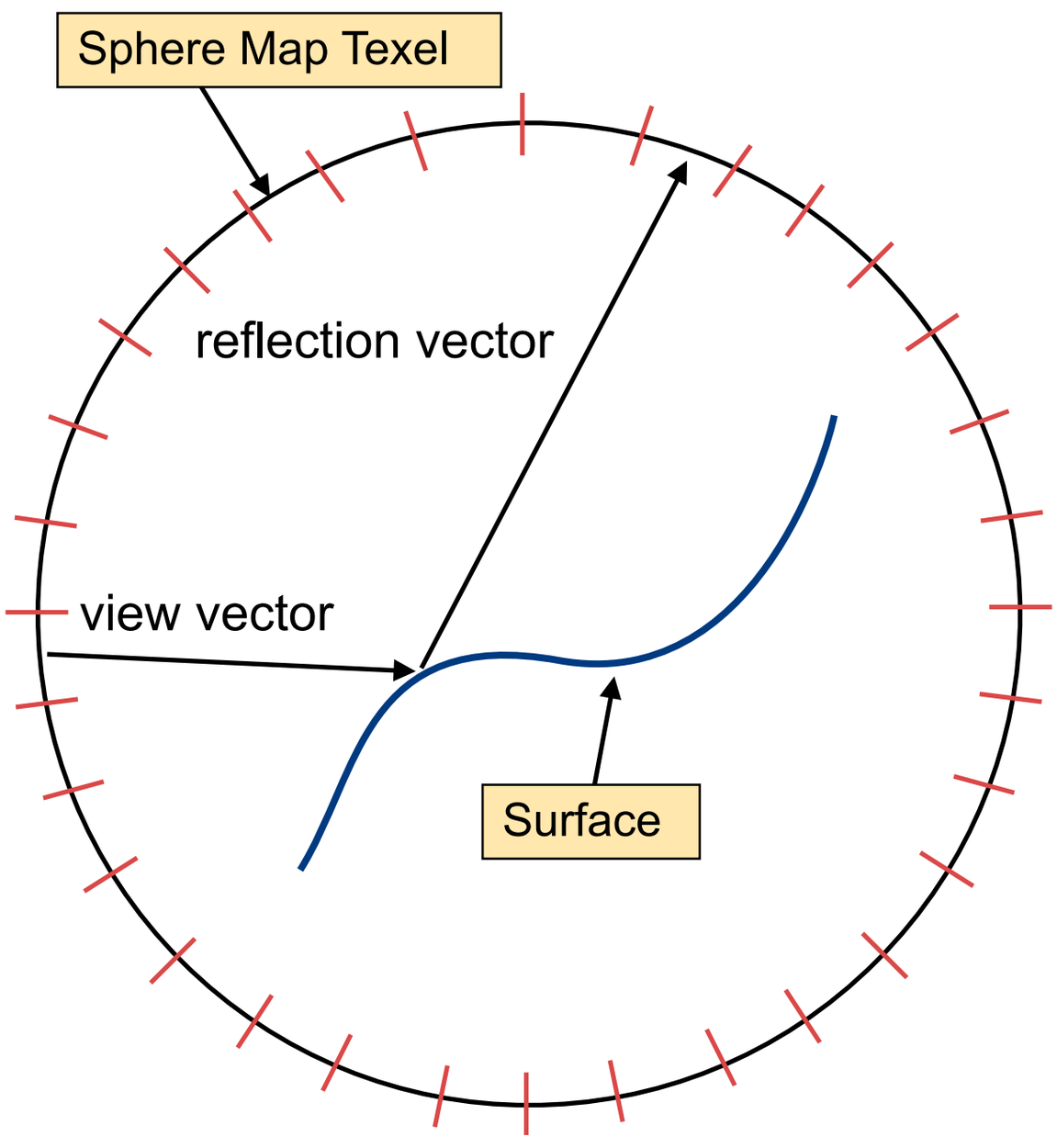
Replace

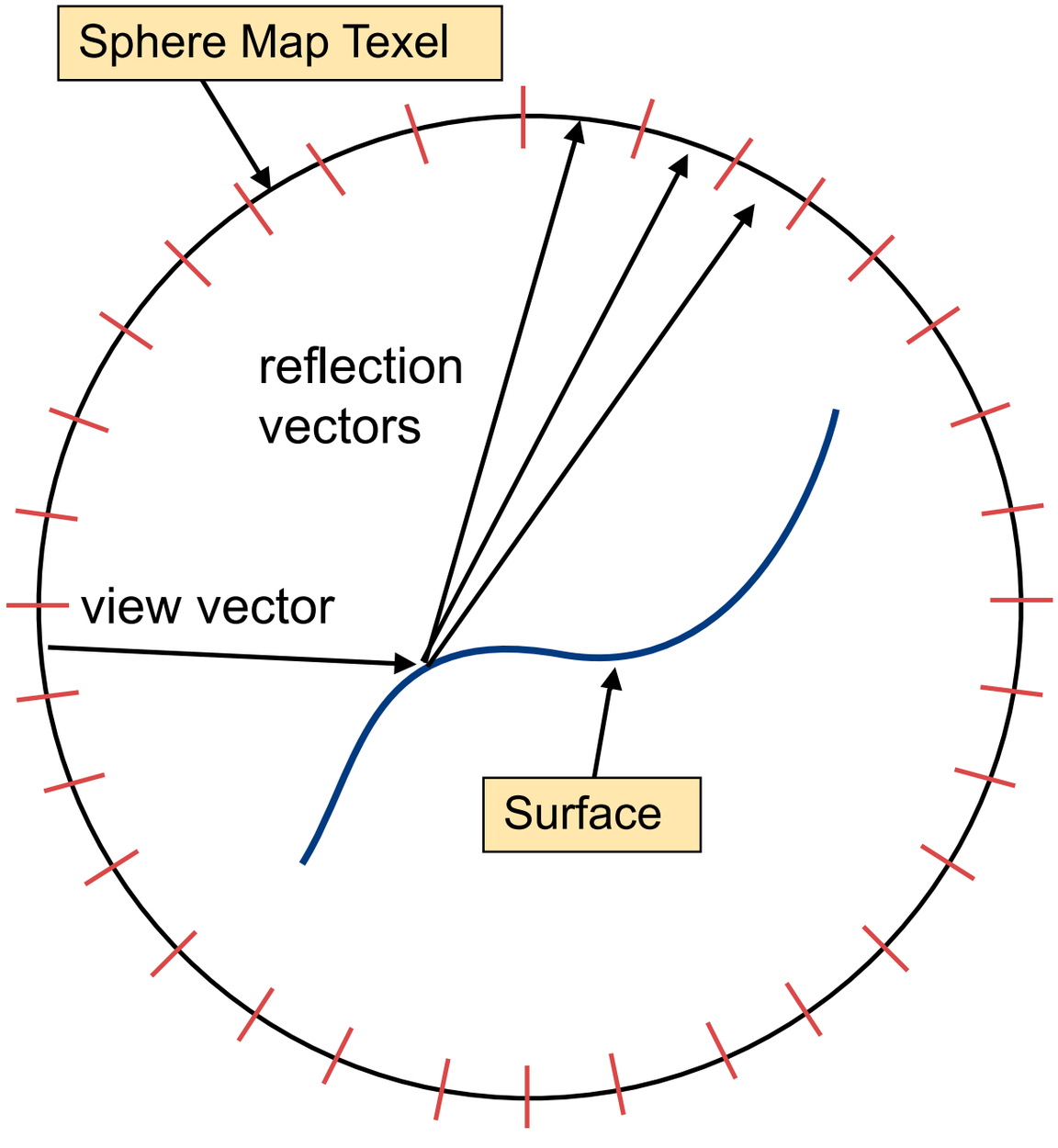
Glossy Effect



Glossy environment maps are created with blurred images.

Glossy Reflection

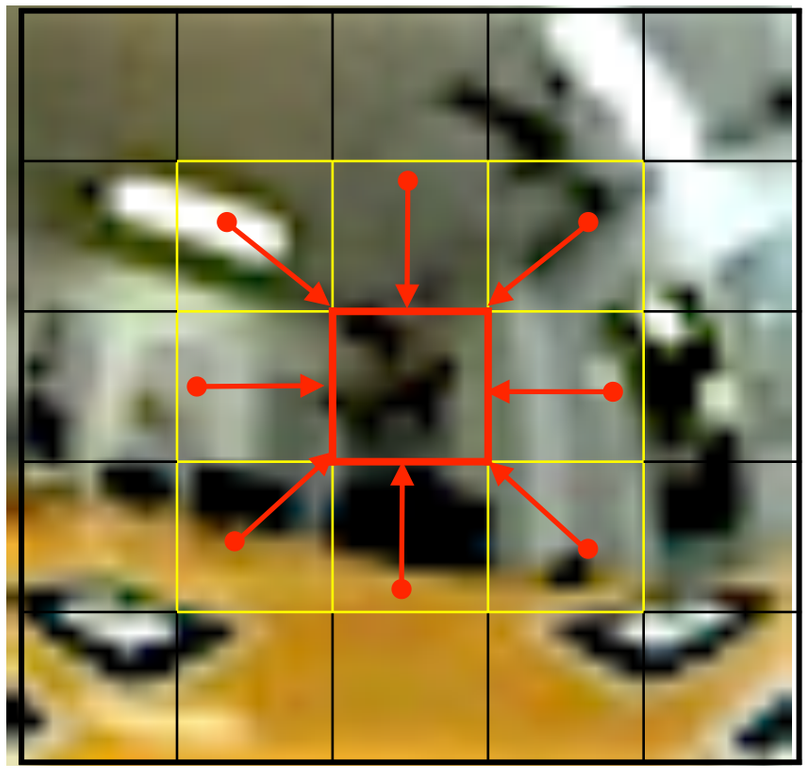




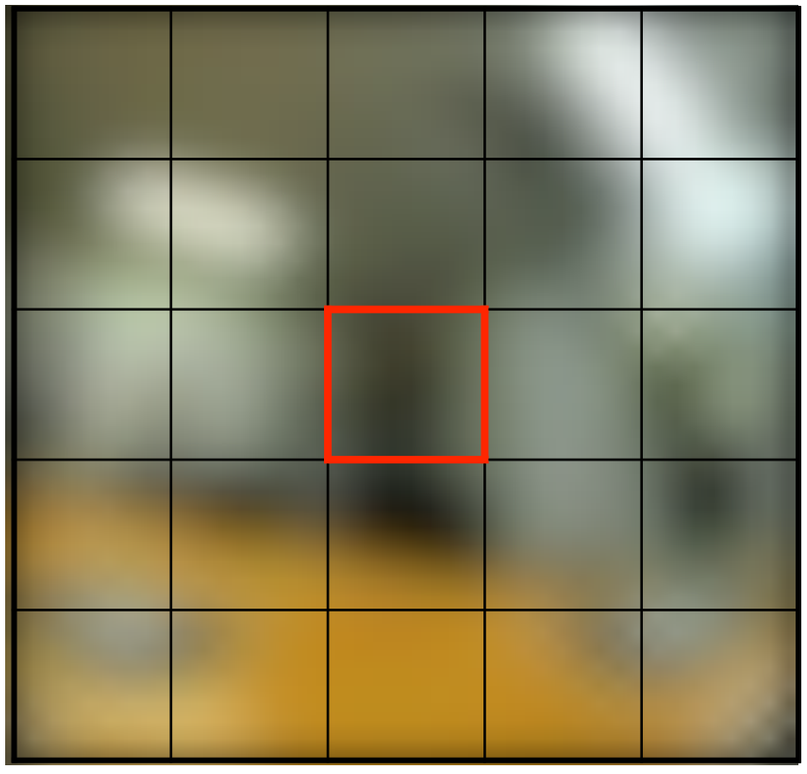
A glossy reflection is created by picking the color information of more than one texel.

The resulting reflection appears like a glowing object in dark scenes and like a glossy diffuse object in bright scene.

Blur the image



Initial image



Final image

Use a Gausian blur filter to blur the image. The image filter changes the color of a particular pixel by adding the color information of the surrounding pixels. This filter simulates multiple reflection vectors. Thus, we do not need to change the reflection equations.

Next Steps

ARLAB

Paul Debevec. A Tutorial on Image-Based Lighting. IEEE Computer Graphics and Applications, Jan/Feb 2002.

Thank you!

Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 5001, USA

+1 515.294.5580
+1 515.294.5530(fax)

rafael@iastate.edu

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY