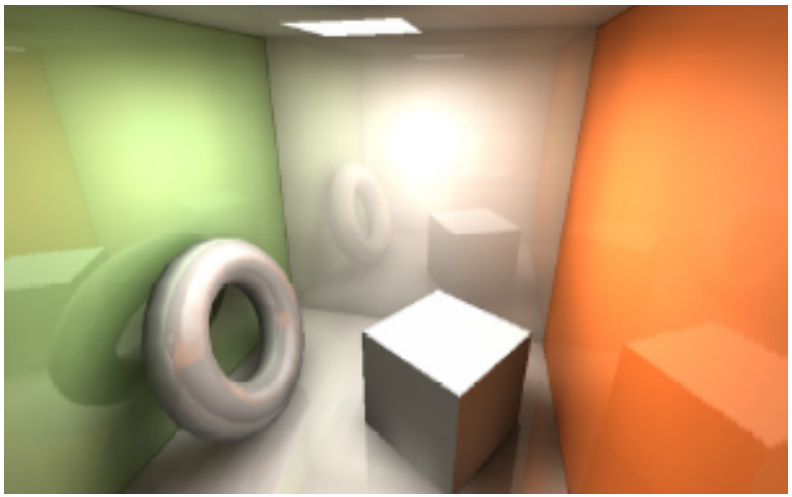
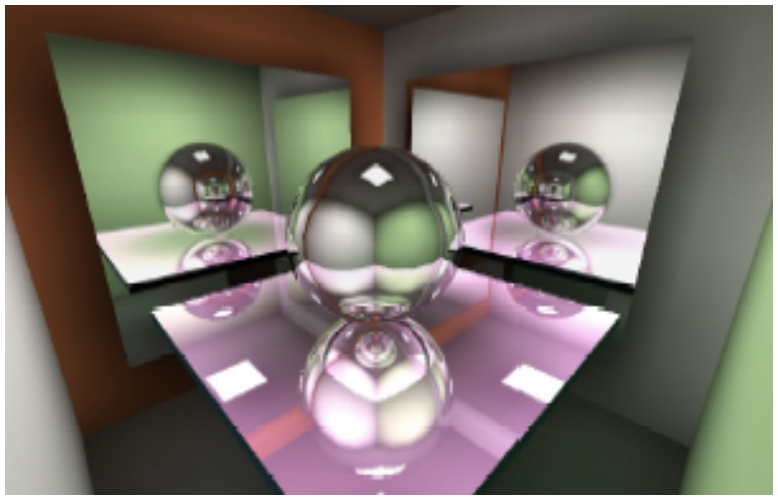
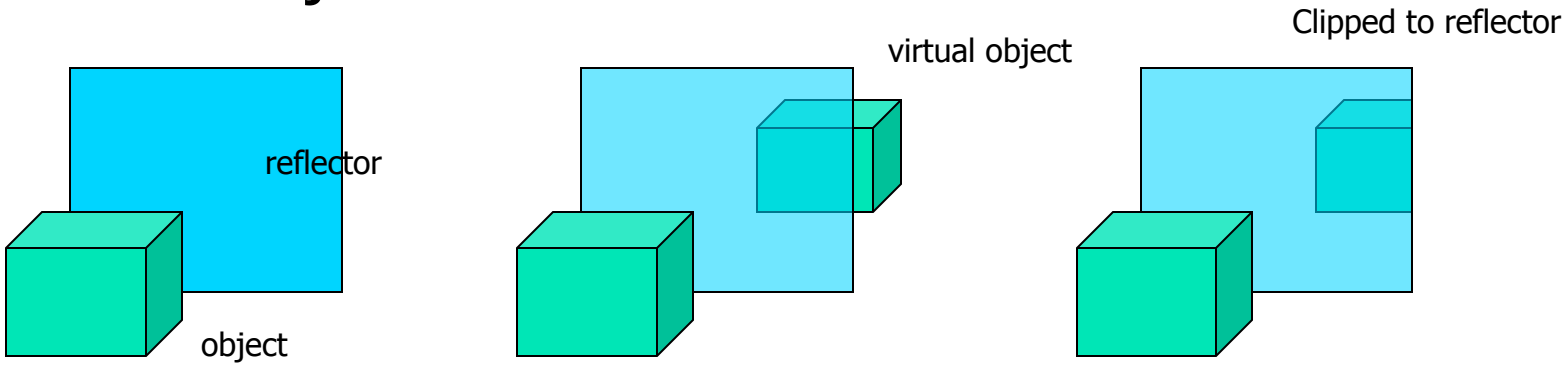
Real Time Reflections





Reflections

- One of the most noticeable effect of inter-object lighting
- Direct calculation of the physics (ray tracing) is too expensive
- Our focus is to capture the most significant reflection while minimizing the overhead via rendering the "virtual object"



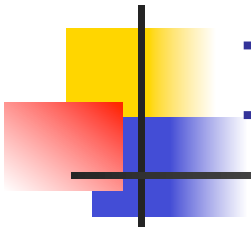
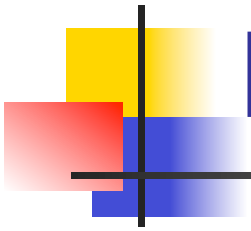


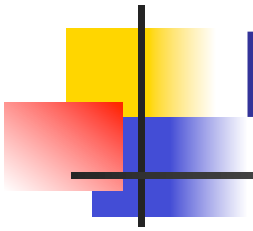
Image vs. Object Space Methods

- Image space methods: create a texture from a view of the reflected objects and apply it to the reflector
  - Advantage: does not depend on the object geometry
  - Disadvantage: sampling issue and also only an approximation (environment mapping as an example)
- Object space methods: create the actual geometry of the object to be reflected and render it to the reflector
  - Disadvantage: accuracy of the geometry
  - Advantage: more accurate reflection (for nearby objects)
- Both methods need to create the virtual objects



Planar Reflections

- The most common reflection – flat mirror, floor, wall, etc
- Creating virtual objects (or reflected objects) is much easier
- A view independent operation – only consider the relative position of the object and the reflector
- The virtual object is created by transforming the object across the reflector plan



Reflection Transformation

- Can be broken into three stages:
  1. Transform the objects into the reflector's local coordinate system
     - Translate the reflector to the world origin and rotate so the reflector plane will coincident with the world's XY plane
  2. Scale by -1 in Z
  3. Transform the reflected object's local coordinates of the reflector to the world coordinates
     - Rotate the world's XY plane to the reflector's plane and then translate the origin to the reflector's local origin

- Overall - A translation of the mirror plane to the origin, a rotation embedding the mirror to x-y plane, a scale of -1 in Z, the inverse rotation previously used, and a translation back to the mirror location

Reflection Matrix

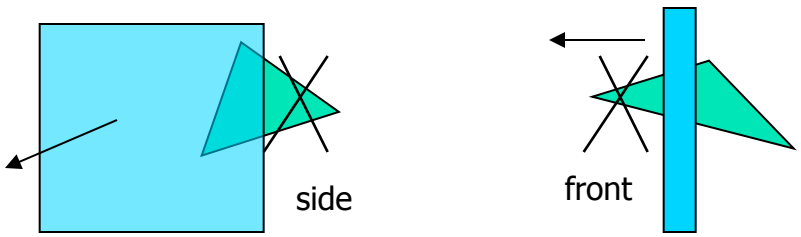
- Given a point P on the reflector plane, vector V perpendicular to the plane

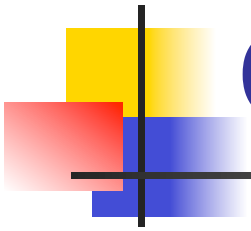
$$R = \begin{vmatrix} 1-2V_x^2 & -2V_xV_y & -2V_xV_z & 2(P.V)V_x \\ -2V_xV_y & 1-2V_y^2 & -2V_yV_z & 2(P.V)V_y \\ -2V_xV_z & -2V_yV_z & 1-2V_z^2 & 2(P.V)V_z \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

- Multiply R to the object vertices produce an reflected object on the opposite side of the reflector plane
- The entire scene is duplicated, simulating a reflector of infinite extent

Render the Reflected Geometry

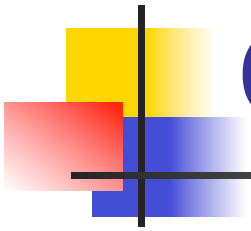
- An important task: clip the reflected geometry so it is only visible on the reflector surface
  - Beyond the reflector boundaries and in front of reflector





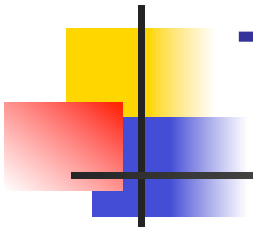
Clipping using the stencil

- The key is you only want the reflected geometry to appear on the reflector surface
- Use stencil buffer:
  - Clear the stencil buffer
  - Render the reflector and set the stencil
  - Render the reflected geometry only at where the stencil pixels have been set
- The above algorithm is to use the stencil buffer to control where to draw the reflection



Clipping using the stencil

- Another method: render the reflected object first, and then render the reflector to set the stencil buffer, then clear the color buffer everywhere except where the stencil being set
- This method is to use the stencil buffer to control where to erased the incorrect reflection
- Advantage: when it is faster to use stencil to control clearing the scene than drawing the entire scene with stenfil tests



The stencil erase algorithm

- Draw the reflected objects
- Clear the stencil and depth buffers
- Configure the stencil so that 1 will be set where polygons are rendered
- Disable draw to color buffer
- Draw the reflector
- Reconfigure the stencil
- Clear the color and depth buffer to the background color (except where the stencil being set)
- Disable stencil test
- Draw the rest of scene (except reflected objects and reflector)

The stencil erase algorithm

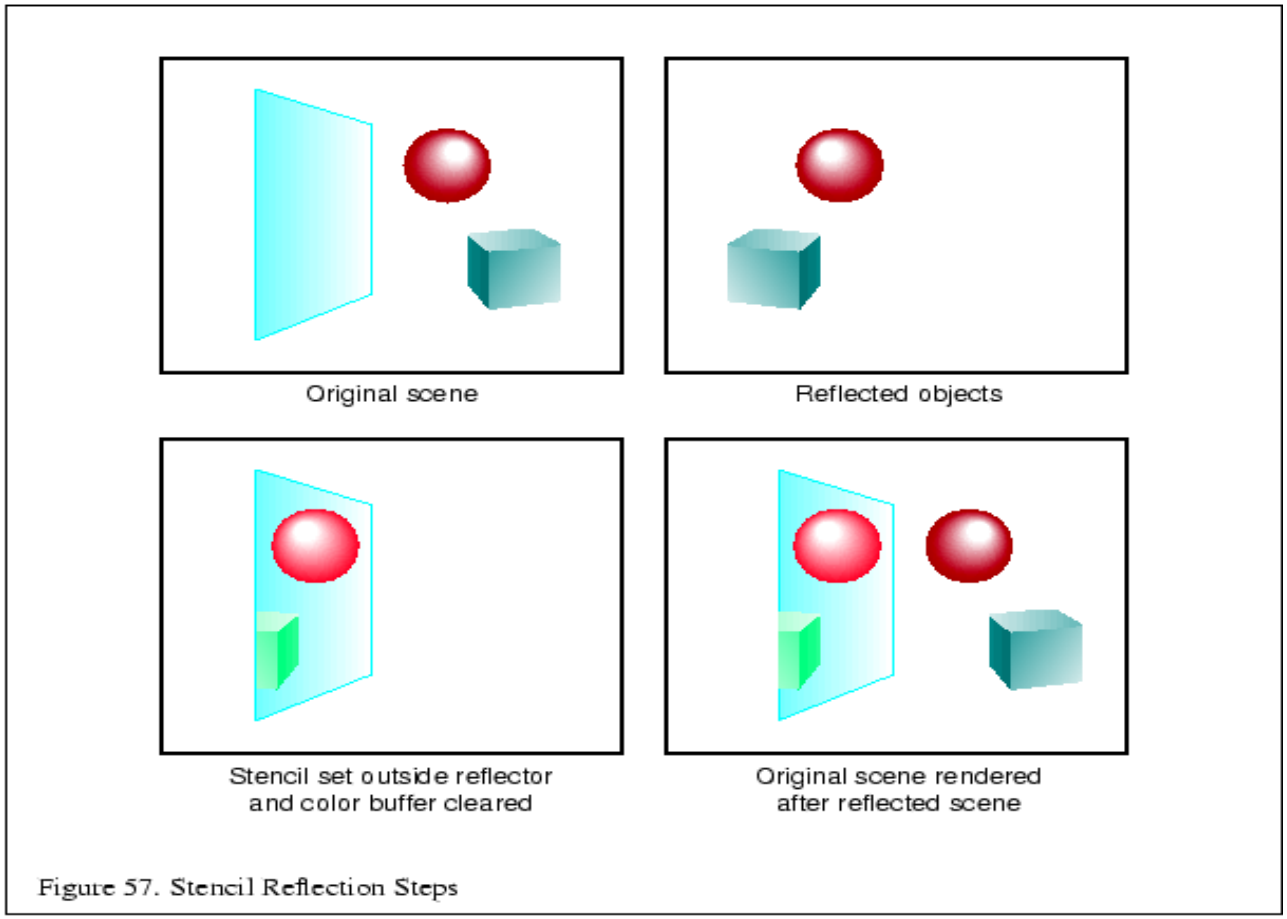
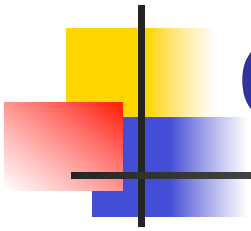
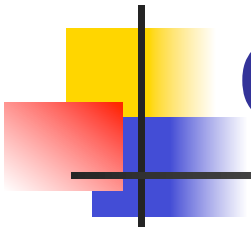


Figure 57. Stencil Reflection Steps



Clipping using texture mapping

- Render the reflected geometry
- Store the image into a texture (using glCopyTexImage2D)
- Clear the color and depth buffer
- Redraw the entire scene, with the reflector textured with the previous texture (the process of texturing reflector will automatically clip the reflected geometry outside the boundary)
- Note that reflected geometry in front of the reflector still needs to be clipped before the texture map is created (using the reflector as the clipping plane – supported by OpenGL's application clipping plane)



Clipping using texture mapping

- The key is to assign correct texture coordinates to the reflector
- Basic idea – s and t coordinates correlate to x and y window coordinates of the reflector
- Using glTexGen and texture transformation matrix
  - Configure the texgen to GL_OBJECT_LINEAR
  - Set the s,t,r to match one to one with x,y,z in eye space
  - The texture matrix is a concatenation of modelview and projection matrices used to render the scene
- Bias and scale the texture coordinates from [-1,1] to [0,1]

Clipping using texture mapping

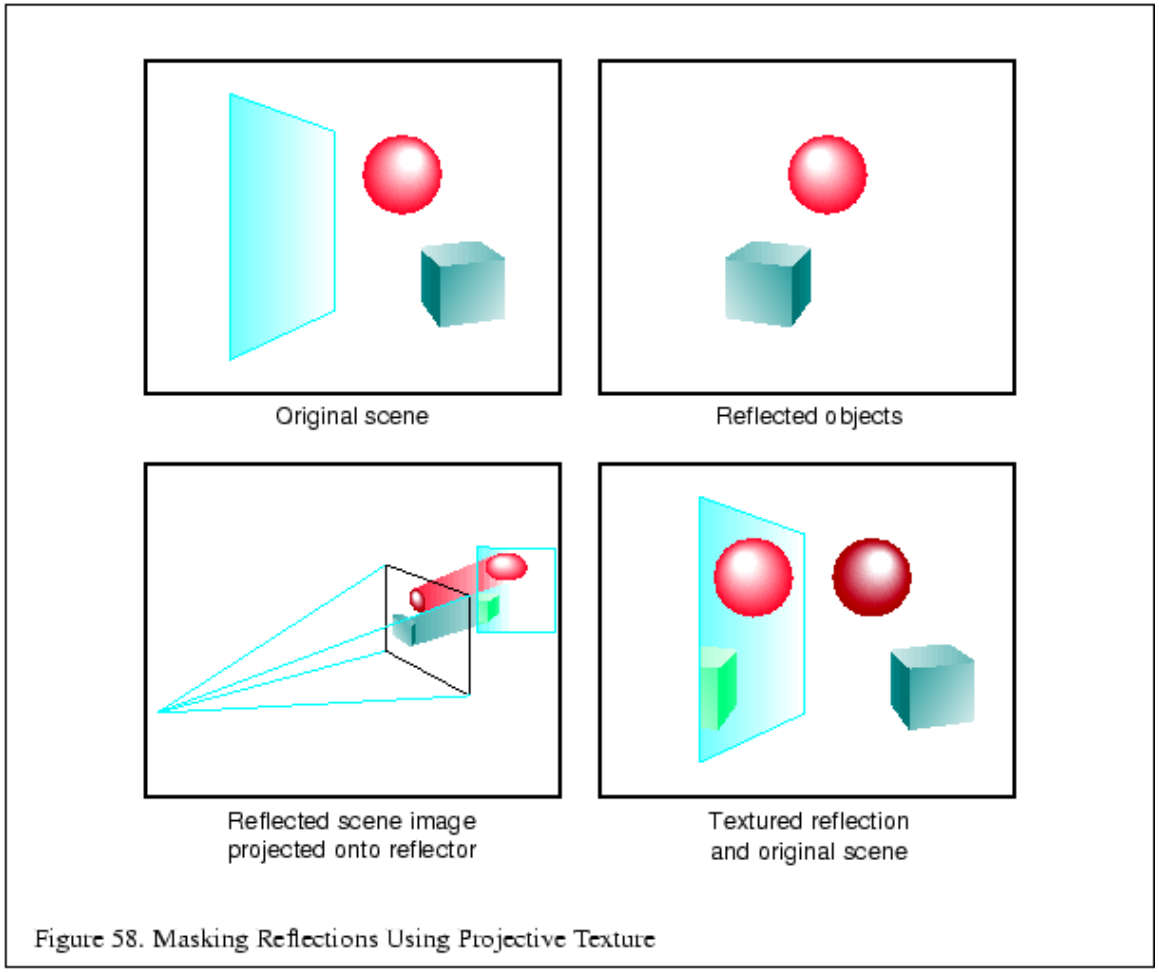
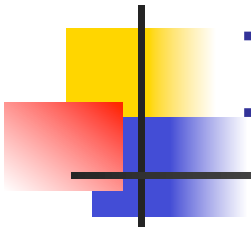
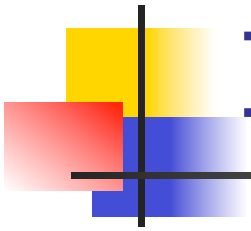


Figure 58. Masking Reflections Using Projective Texture



Interreflection

- Goal: let the reflections to “bounce” between reflectors
- The number of bounces needs to be limited to preserve interactivity
- The reflection transformations need to be concatenated
- Render the deepest interreflection first

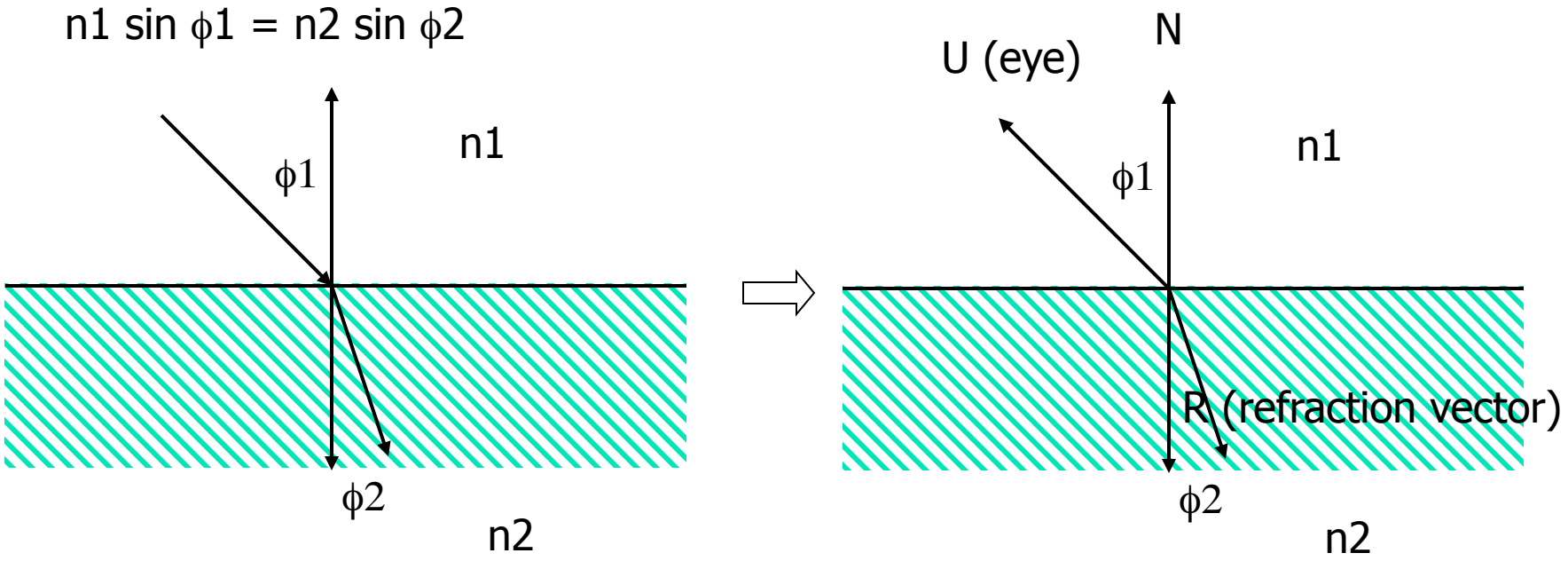


Interreflection

- Clear the stencil buffer
- Set the stencil operation to increment the stencil values where pixels are rendered
- Render each reflector involved in interreflection into stencil buffer
- Set the stencil test to pass where the stencil value equal the number of interreflection
- Apply planar transformation
- Draw the reflected scene
- Draw the reflector, blending if desired

Refraction

- Snell's law – refraction angel is based on the medium's refraction index

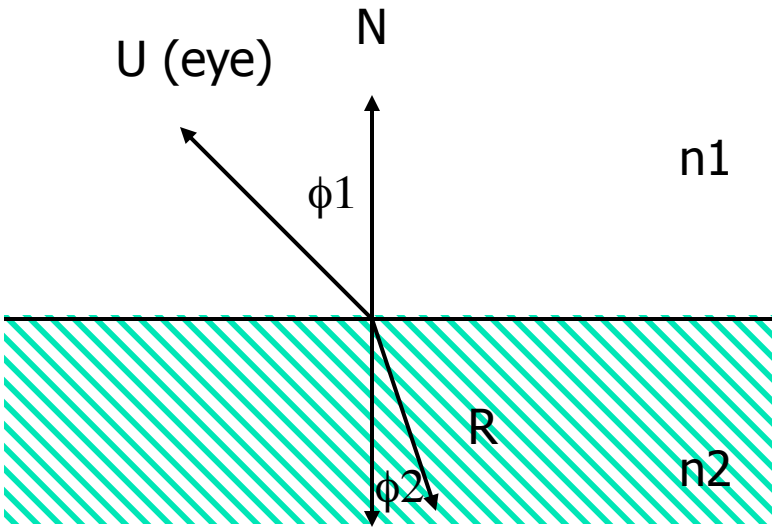


Refraction Vector

- n = n1/n2

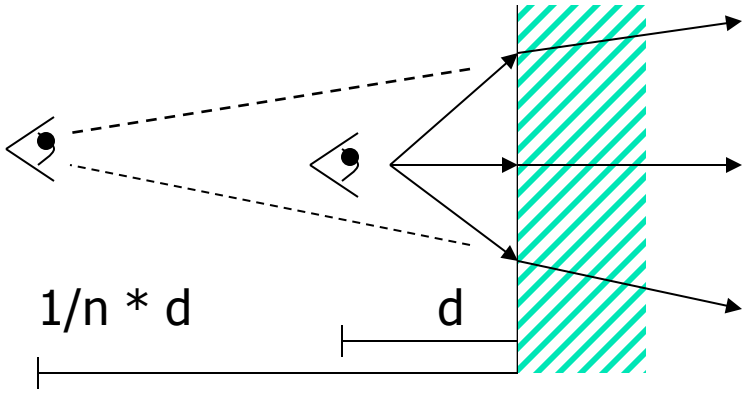
$R = nU - N(n(N.U) + \sqrt{1-n^2(1-(N.U)^2)})$ or

$R = U - (1-n)N(N.U)$ (a simplified form)

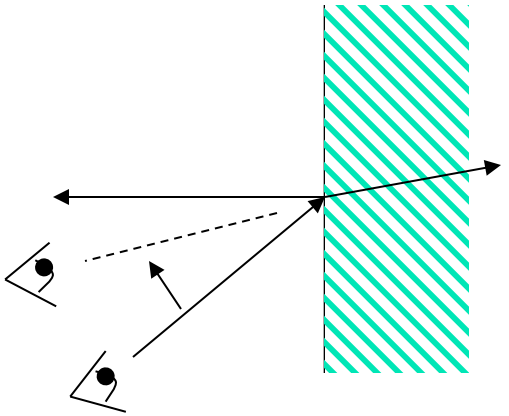


Planar Refraction

- Move the eye point to match the refraction vector

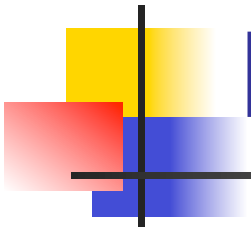


away from the interface
(when eye ray is perpendicualr)



rotate the eye (when not Perpendicular)

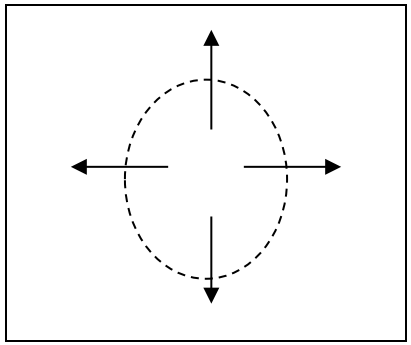
When the refraction ray bent toward the normal direction (a common case)



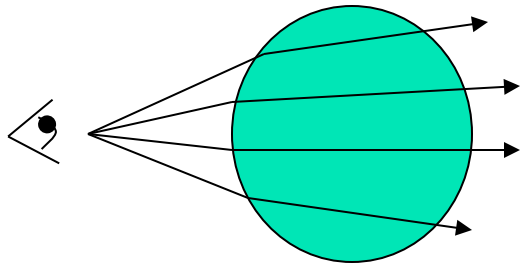
Environment Map Refraction

- Take 6 shots from the refractor to its environment
- Based on the eye rays, calculate the refracted rays using snell's law at each refractor's vertex
- Using the refracted rays as texture coordinate (s,t,r) to look up the environment cube map

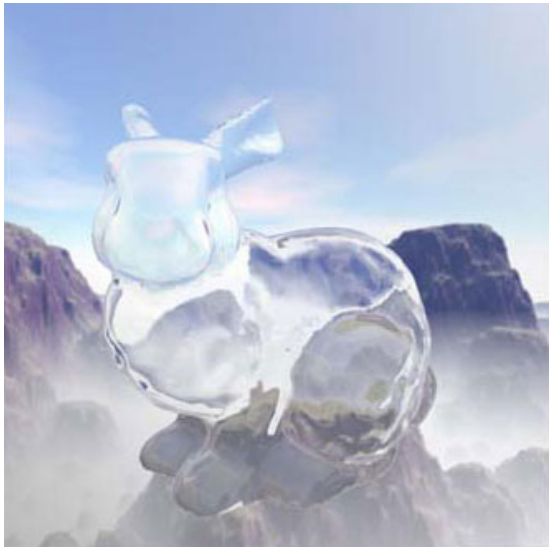
Environment Map Refraction



Create Environment Cube Map



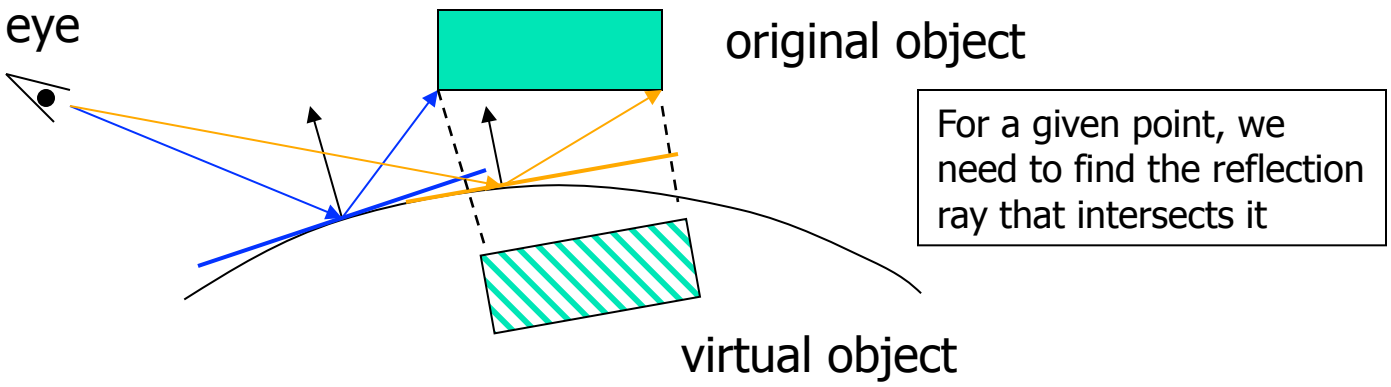
Calculate the refractor ray and
Use it as the texture coordinates
to the environment cube map

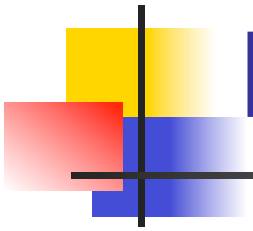


result

Curved Reflectors

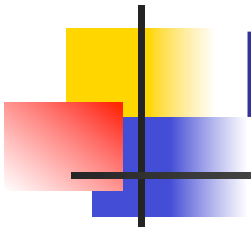
- The idea is similar to planar reflectors – create virtual objects by transforming the object vertices
- The difference is that there is no longer a single plane as the reflector so the transformation becomes view-dependent and varies from vertex to vertex





Polygonal Reflectors

- Trivial part: Each polygon/triangle can be thought of as a planar reflector with a single facet normal and reflection plane
- Accurate only when the reflector is faceted
- Non-trivial issue: for a given object vertex, which triangles in the reflector should be used to create virtual vertices?
- Brute-force: reflect every vertices against every triangle and rely on clipping
- Better way: using spatial subdivision or explosion maps

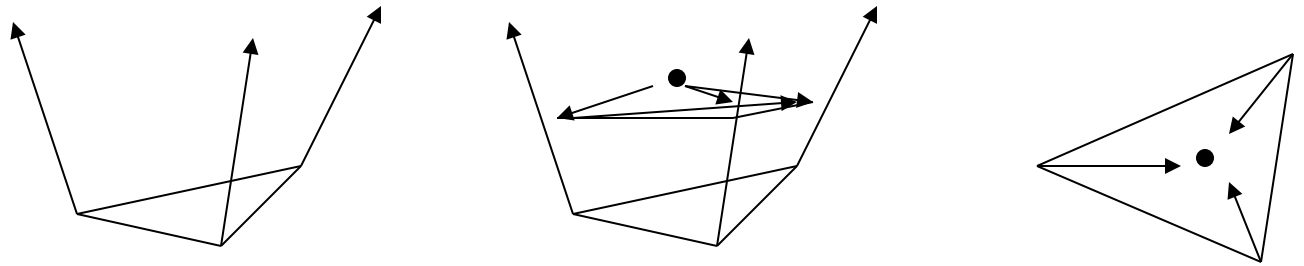


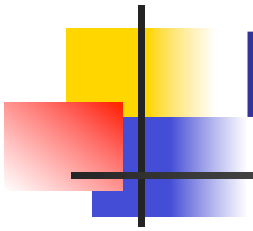
Polygonal Reflectors

- When the model is considered as smooth, treating each triangle as a faceted reflection is not good enough
- Instead of one normal per face, we should find the reflection point and form the reflection plane by interpolating the normals at vertices using barycentric coordinates

Interpolating Parameters

- Extend each vertex's normal into a ray
- Project the point onto each of the rays
- Perform barycentric interpolation of normals, and reflection position to form the reflection plane





Explosion Maps

- To answer the question: given an object vertex, how to find th proper triangle that contains the reflection point?
- Explosion map allows us to encode the volumes of space owned by the triangles in the reflector's mesh

- For every reflector mesh triangle vertex, we compute the reflection ray
- We use the reflection ray to intersect a sphere and get a vector (x,y,z) from the sphere center to the sphere intersection point
- Map the unit vector (x,y,z) into 2D coordinates (s,t) where
  - s = r/2 (1 + x / sqrt(2(z+1)))
  - t = r/2 (1 + y / sqrt(2(z+1)))

  Where r is the radius of a circle inscribed in the explosion map
- We then render the identification color for each triangle into the explosion map

(http://www.opengl.org/resources/tutorials/sig99/advanced99/notes/node166.html)

Explosion Maps

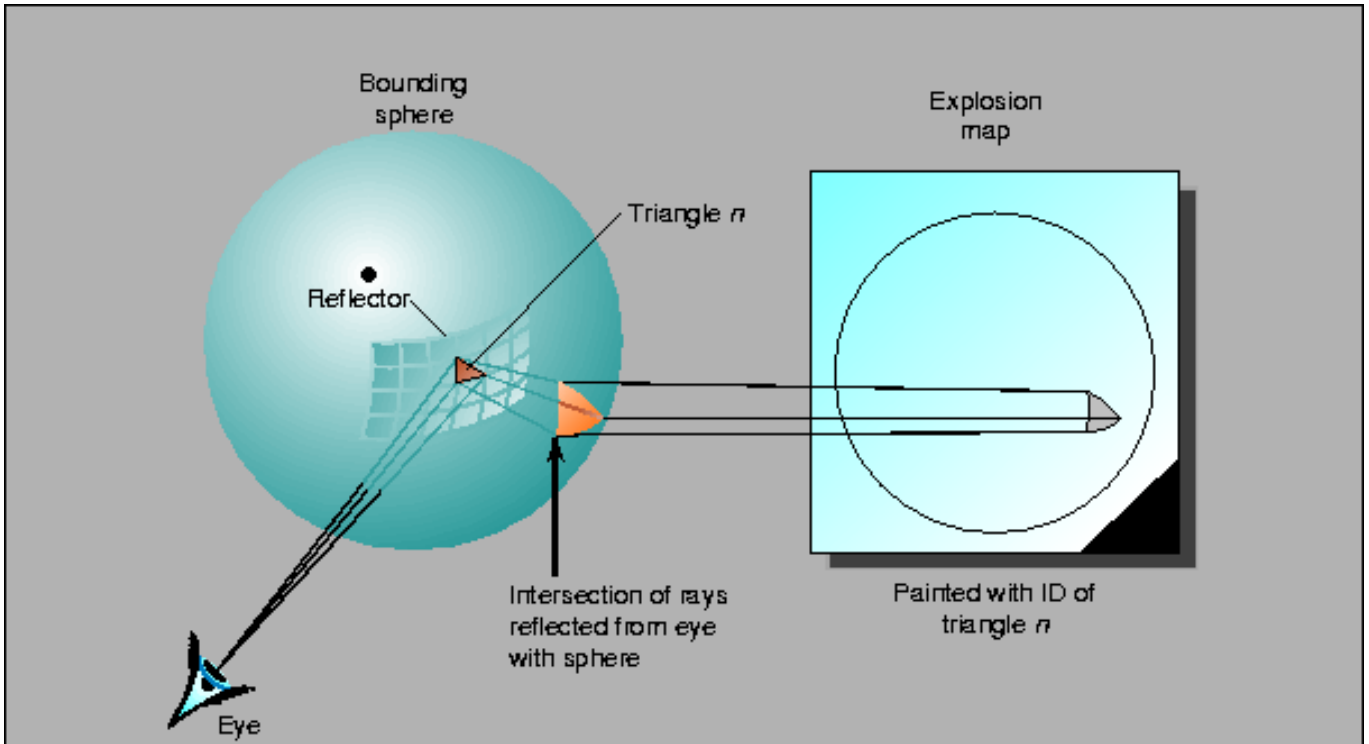


Figure 61. Mapping Reflection Vectors into Explosion Map Coordinates

Explosion Maps

- With the explosion map, for any point on the sphere, we know exactly which triangle will reflect that point.

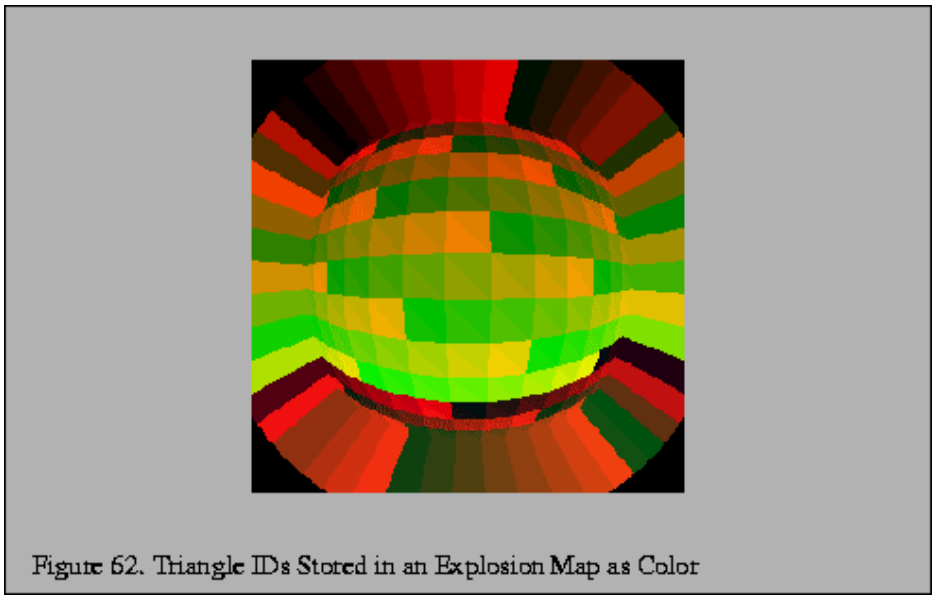


Figure 62. Triangle IDs Stored in an Explosion Map as Color

Explosion Maps

- Find the reflector triangle:

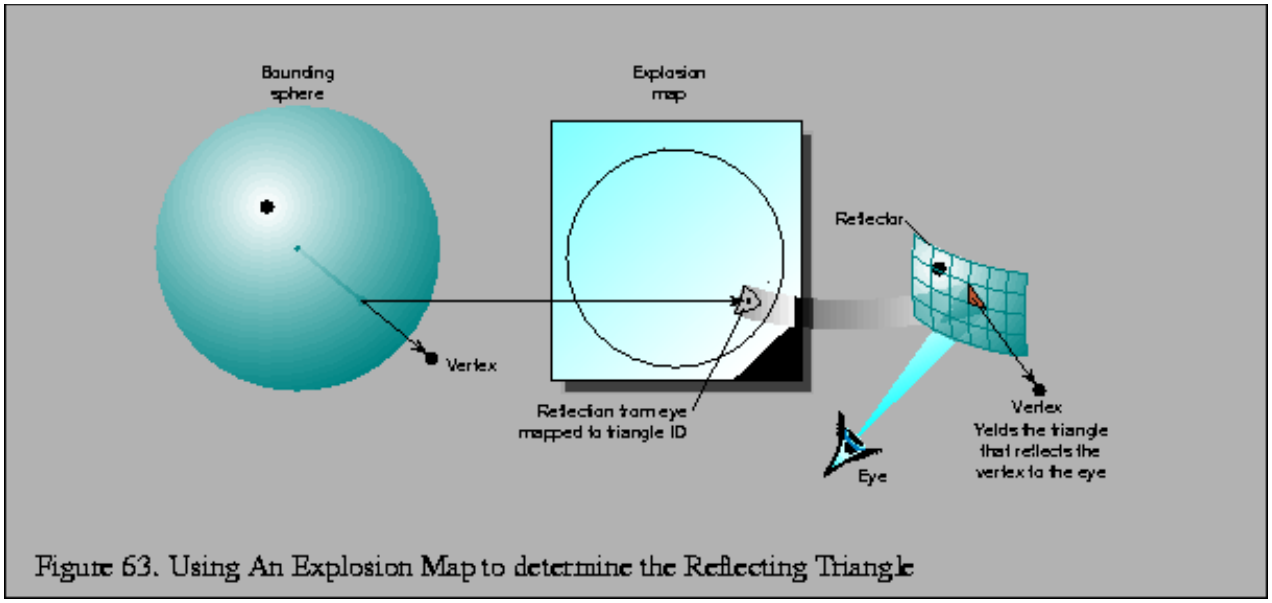
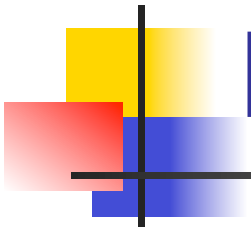


Figure 63. Using An Explosion Map to determine the Reflecting Triangle



Explosion Maps

- We need to get a sphere that will contain all scene vertices (difficult)
- Use two spheres – one tightly bound the reflector and the other bound the entire scene
- Two explosion maps are created as a result and we will interpolate to find the reflected vertex

Explosion Maps

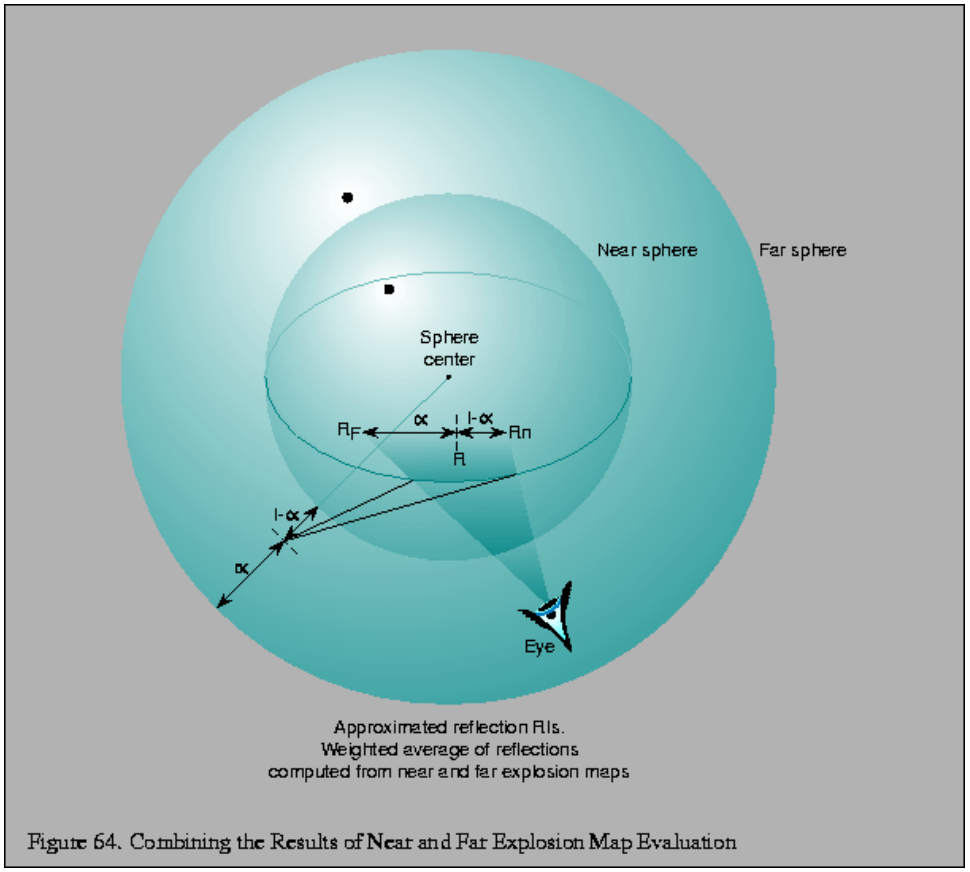


Figure 64. Combining the Results of Near and Far Explosion Map Evaluation