UCLA

CS174A : Introduction to Computer Graphics

1240 Kinsey
MW 4-6pm

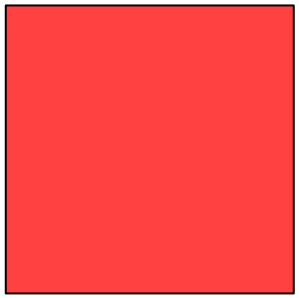
Scott Friedman, Ph.D
UCLA Institute for Digital Research and Education

Assignment #2

- Due Friday February 10, 2017 at Midnight.
  - GitHub repositories created starting with – a2
- Term project proposal
  - Due Friday 2/10 (same day)
  - Should have your teams formed by now!
    - Use Piazza!
- Mid-Term
  - Monday February 13, 2017
  - Through Texture Mapping

Quick Review

- Transparency order matters.
  - Red box (1,0,0), alpha = 0.75
  - Drawn over white background, alpha = 1.0

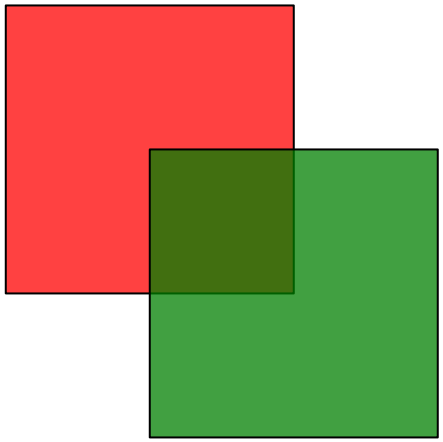


$$(R_{d'}, G_{d'}, B_{d'}, \alpha_{d'}) = (\alpha_s R_s + (1-\alpha_s)R_d, \alpha_s G_s + (1-\alpha_s)G_d, \alpha_s B_s + (1-\alpha_s)B_d, \alpha_s \alpha_d + (1-\alpha_s)\alpha_d)$$

$$(1.0, 0.25, 0.25, 1.0) = (0.75(1) + 0.25(1), 0.75(0) + 0.25(1), 0.75(0) + 0.25(1), 0.75(1) + 0.25(1)).$$

Quick Review

- Transparency order matters.
  - Green box (0,1,0), alpha = 0.75
  - Drawn *over* (i.e. after) the red box.

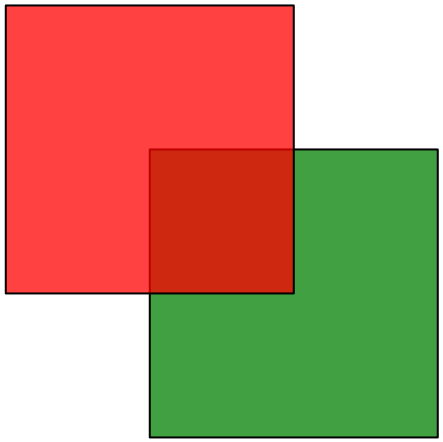


$$(R_{d'}, G_{d'}, B_{d'}, \alpha_{d'}) = (\alpha_s R_s + (1-\alpha_s) R_d, \alpha_s G_s + (1-\alpha_s) G_d, \alpha_s B_s + (1-\alpha_s) B_d, \alpha_s \alpha_d + (1-\alpha_s)\alpha_d)$$

$$(0.25, 0.8125, 0.0625, 1.0) = (0.75(0) + 0.25(1), 0.75(1) + 0.25(0.25), 0.75(0) + 0.25(.25), 0.75(1) + 0.25(1)).$$

Quick Review

- Transparency order matters.
  - If we reverse the rendering order, however.
  - We get a different blending result.

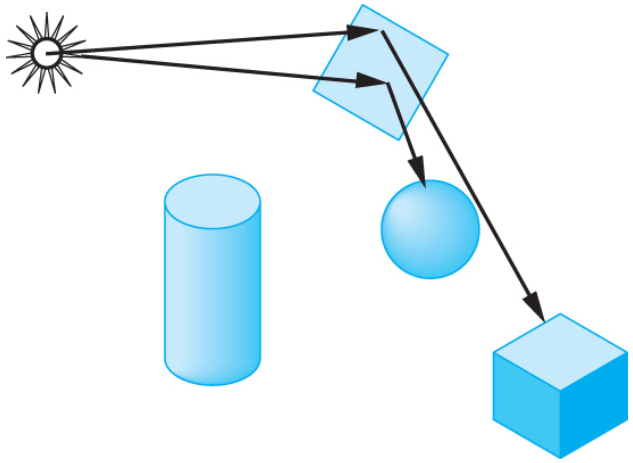


$$(R_{d'}, G_{d'}, B_{d'}, \alpha_{d'}) = (\alpha_s R_s + (1-\alpha_s) R_d, \alpha_s G_s + (1-\alpha_s) G_d, \alpha_s B_s + (1-\alpha_s) B_d, \alpha_s \alpha_d + (1-\alpha_s)\alpha_d)$$

$$(0.8125, 0.25, 0.0625, 1.0) = (0.75(1) + 0.25(0.25), 0.75(0) + 0.25(1), 0.75(0) + 0.25(0.25), 0.75(1) + 0.25(1)).$$

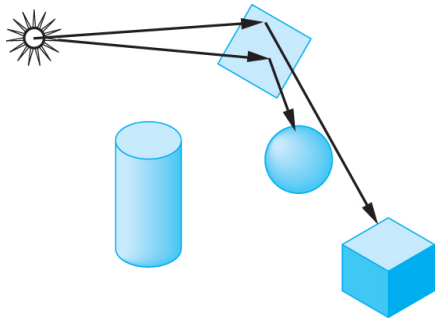
Environment Mapping

- Reflection of a **fixed** environment.
  - Sometimes called reflection mapping.
  - Reflections could be computed via ray tracing.
    - Too slow for real-time environments, generally.
  - However, a texture map can be used to approximate the effect.



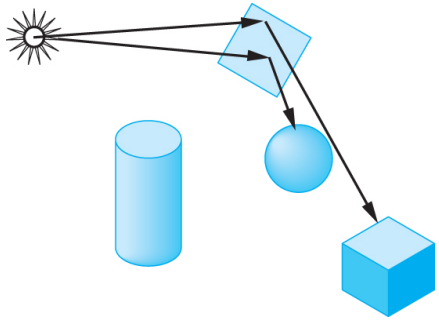
Environment Mapping

- How can we do it?
  - Take advantage of the fact that we can compute the reflection vector. (i.e. eye -> surface -> reflect -> world).
  - Intersect that ray with the scene and compute the color (shading) value.
  - Good, except this is essentially ray tracing!



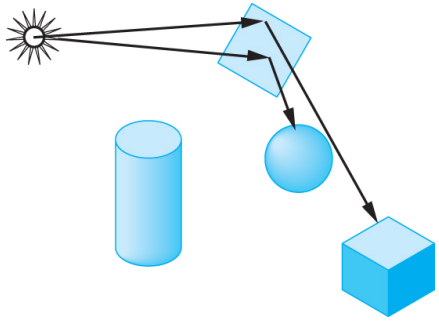
Environment Mapping

- How can we do it?
  - Approximate things with a two-pass method.
  - Place camera at the location of the mirror object.
  - Facing in direction of mirror surface normal.
  - Render scene, without mirror – into texture map
  - Render scene normally with texture map applied to mirror geometry.
    - Camera in original position and mirror back in scene



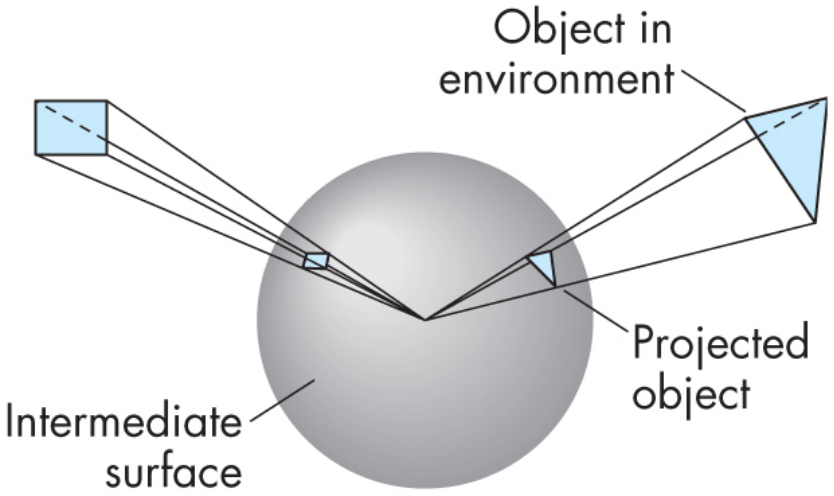
Environment Mapping

- How can we do it?
  - There are several problems with this approach.
    - Where exactly to put the camera?
    - Mirror is missing from first render pass
      - » Which can mess up lighting / mirror occludes something.
    - Projection from mirror can be tricky to setup – off-axis
    - Have to re-render 'reflection' every time camera or mirror moves.
      - » Maybe slow



Environment Mapping

- How can we do it?
  - Projection of scene onto a sphere centered at COP.
  - Viewer cannot tell difference between object and projection.
  - Common example is the experience inside of a planetarium.



Environment Mapping

- How can we do it?
  - The result looks like this. (360x360 degree view)
  - Given a reflection vector we can look up the shaded color in the scene directly.
  - Google Street View, the 360 attachments to phones and GoPro's etc, QuicktimeVR(?) are variations of this

Environment Mapping

Environment Mapping

- How can we do it?
  - WebGL supports this method directly
    - Given a reflection vector - determining texture coordinates *s* and *t* is straightforward.

$$\mathbf{r} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = 2(\mathbf{n} \bullet \mathbf{v})\mathbf{n} - \mathbf{v}$$

$$f = 2\sqrt{r_x^2 + r_y^2 + (r_z + 1)^2}$$

$$s = \frac{r_x}{f} + \frac{1}{2}$$

$$t = \frac{r_y}{f} + \frac{1}{2}$$

Environment Mapping

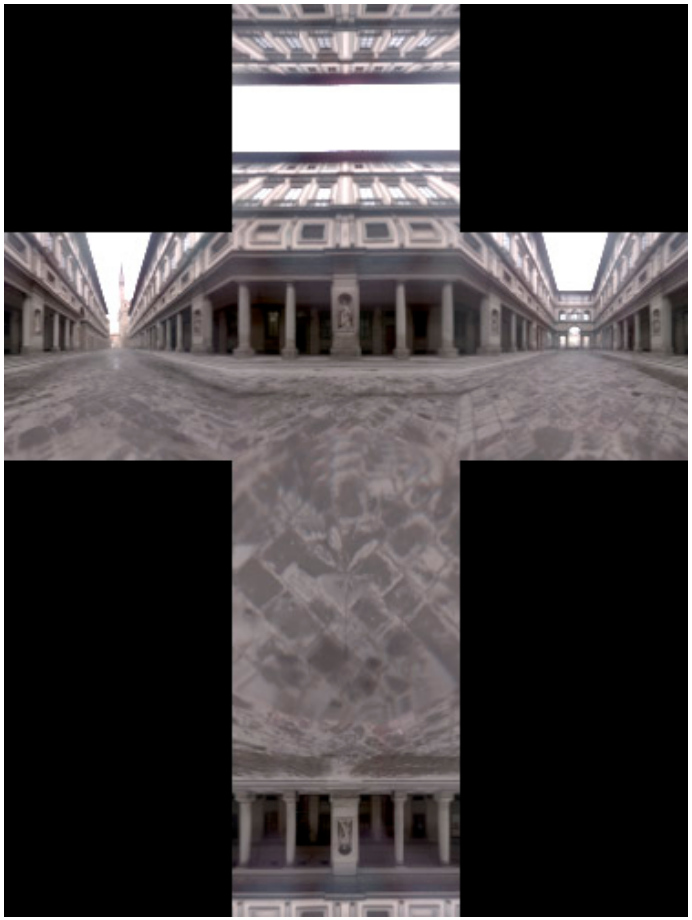
- How can we do it?
  - A not really correct method (but good enough)
    - Created at a specific location (origin?)
    - Needs to be recomputed if scene changes.

Environment Mapping

- How can we do it?
  - Difficult, but doable to create map.
    - Real cameras have spherical lenses (Google Street View)
    - Sometimes cylindrical maps are created.
  - Create map with WebGL has a simpler way.
    - We can use the projections we have already developed.

Environment Mapping

- Make things even simpler
- Cube mapping
  - Render six images
  - Each centered on an axis
  - Results in the inside of a cube
    - Unfolded here →
  - FOV must be 90 deg.
  - Edges have to match



Environment Mapping

- Cube mapping
  - Given we are passing correctly transformed surface normals into the vertex shader.
    - We can compute the reflection vector and pass it on to the fragment shader.

```
attribute vec4 vPosition;
attribute vec4 normal;
varying vec3 reflection;

uniform mat4 ModelView;
uniform mat4 Projection;

void main( void )
{
  gl_Position = Projection * ModelView * vPosition;
  vec3 eyePos = vPosition.xyz;
  reflection = reflect( eyePos, normalize( (ModelView * normal).xyz );
}
```

Environment Mapping

- Cube mapping
  - The reflection vector is interpolated for us when it arrives in fragment shader.
  - So the rest is simple
  - Notice
    - samplerCube type (similar in function to sampler2D)
    - textureCube function

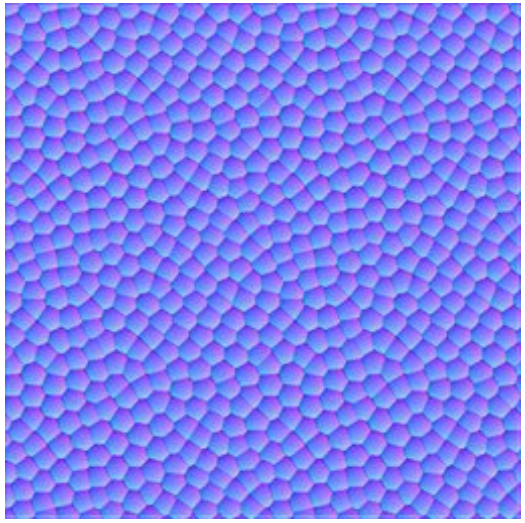
```
varying vec3 reflection;
uniform samplerCube cubeMap;  // passed this in from our application

void main( void )
{
  gl_FragColor = textureCube( cubeMap, reflection );
}
```

Bump Mapping

- Bump mapping
  - Bump mapping is really Displacement Mapping.
  - A texture map stores displacements to the surface.
  - Requires computing a normal at the new, displaced, surface point.
  - Partial derivatives must be found to solve this.
    - Finite differences can be used.
    - Slow-ish when solving for every fragment.
  - Advantage is we can use the result to perturb the normal in *object* space.

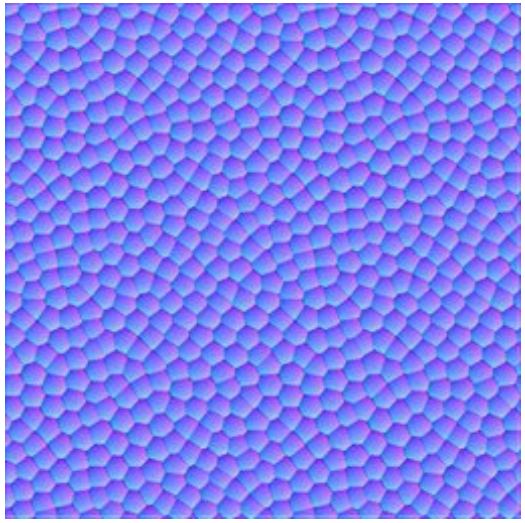
Bump Mapping

- Normal mapping
  - What we typically think of as Bump Mapping.
  - Here we store *normals* in a texture map.
    - *x*, *y* and *z* components of a normal vector are stored in the R, G and B channels of the texture map.



Bump Mapping

- Normal mapping
  - We have to map the [-1,1] range of our normal components to the [0,1] range of color values.
  - Easily done by [R,G,B]=([x, y, z] + 1) / 2



Bump Mapping

- Normal mapping
  - This is not enough.
  - Normal maps require a new *coordinate frame*.
  - *Tangent Space* or TBN space
    - **TBN** = **T**angent, **B**i-tangent, **N**ormal space
    - The **T**angent is the tangent to the surface at point **p** **(R)**
    - The **B**i-tangent is the cross product of the normal and tangent. (G)
    - The **N**ormal is the surface normal at point **p** **(B)**
      - » *Why the normal maps tend to look blueish*
  - We operate in TBN space (a new basis) so that if the orientation of the surface changes the map still works as the map is defined with respect to the texture not object space.

Bump Mapping

- Normal mapping
  - The TBN vectors define a new frame at the *point* we are lighting on the surface.
  - We align the T and B vectors with the *s* and *t* dimensions of our normal (texture) map.
    - So the normal maps align when applied across multiple triangles.
  - We can get from object space to tangent space by a transformation.

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix}$$

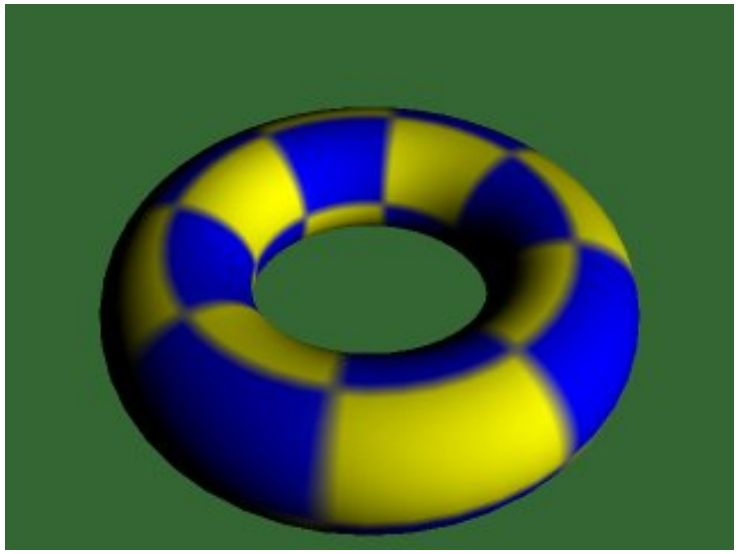
Bump Mapping

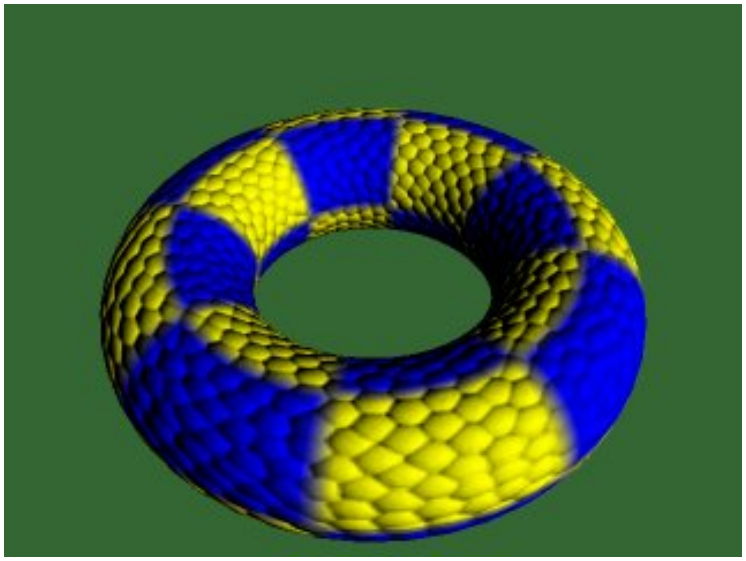
- Normal mapping
  - Compute T and B
  - Can now transform lighting equation vectors into TBN space
  - Normal map normal can now be used in lighting equations.

```
vec3 dPos1 = v1 – v0
vec3 dPos2 = v2 – v0
vec2 dST1 = st1 – st0
vec2 dST2 = st2 – st0
float r = 1.0 / ( dST1.x * dST2.y – dST1.y * dST2.x )
vec3 T = ( dPos1 * dST2.y – dPos2 * dST1.y ) * r
vec3 B = ( dPos2 * dST1.x – dPos1 * dST2.x ) * r
```

Bump Mapping

- Normal mapping
  - We transform the light vector by the TBN matrix.
  - The light and normal map vectors are then in the same coordinate system so lighting equations are correct.
  - Voila! A snake donught!





Go finish Assignment #1