Review of Tuesday

• **Normal Mapping**
  - Textures can be used to store normal data in tangent space
  - Increases lighting details, but preserves geometry/silhouette

• **Environment Mapping**
  - Sphere and Cube-Mapping are established techniques
  - Environment is pre-rendered into texture(s)
  - Texture coordinates are created dynamically based on reflection vector
  - Can also be used for refraction, skyboxes, etc.

• **Shadow Mapping**
  - Shadow mapping and shadow volumes
Shadows

- **Shadow volumes** create shadow by clipping geometry against volumes cast by light
Shadows

- In OpenGL shadow volumes can be implemented efficiently using the **stencil buffer**
  1. Draw objects that can receive shadows
  2. Disable write to depth and color buffer, initialize stencil with 0
  3. Draw front-faces of all shadow volumes (increase stencil value of all fragments that pass depth test)
  4. Draw back-faces of all shadow volumes (decrease stencil value of all fragments that pass depth test)
  5. Enable write to color buffer (and depth, if required)
  6. Draw shadow color everywhere, where stencil buffer > 0
Shadows

• Stencil shadows
Shadows

• **Shadow volumes vs. Shadow maps**
  • **Shadow volumes**
    • Require the extraction of silhouettes/volumes
    • Compute exact per-pixel shadows
    • Generate shadows that generally have sharp edges
  • **Shadow maps**
    • Inherently a two/multi-pass method
    • Resolution of (camera) depth map governs resolution of shadows
    • Shadow maps can be blurred
Ambient Occlusion

- **Ambient occlusion** is a basic approximation of global illumination that is able to mimic shadowing based on partial occlusion.
Ambient Occlusion

• For any point on an object surface with normal $n$, ambient occlusion is given as

$$A(p) = \int_{\Omega_n} V(p, \omega) \ d\omega$$

$V$ is a visibility function that indicated whether the ‘sky’ is visible from $p$ in the direction of $\omega$. 
Ambient Occlusion

- This integral can be approximated by sampling a hemisphere with rays (e.g., Monte Carlo Sampling)

explicit ray-casting is expensive
Ambient Occlusion

- Popular: **Screen-space ambient occlusion (SSAO)**
  - Runs in real time
  - Approximates ambient occlusion integral
  - Samples the depth-buffer of a scene

depth values in the z-buffer can be used to approximate occlusion

Simplest approach: count number of neighbors that are ‘in front’
Ambient Occlusion

- Screen-space ambient occlusion

scene in depth buffer  screen-space AO created by sampling z-buffer  final image
Ray-Tracing
Ray-Tracing

• The standard graphics pipeline relies on transformation, projection, and rasterization of triangulated geometry.

• Physically speaking, the colors that we see are reflected rays of light. **Ray-tracing** is an image-generation process that mimics this behavior.
Ray-Tracing

shadows

reflections
Ray-Tracing

- Ray-tracing especially useful for **very reflective scenes**
- Repeated ray splitting may make rendering inefficient

reflection + transmission/refraction
Ray-Tracing

- Ray-tracing is not part of the standard graphics pipeline
- Can be implemented on parallel computers (such as the GPU) due to independent ray computations

- Ray directions define camera/lens model

![Ray directions diagram]

rays pass through virtual pixel centers on image plane
Ray-Tracing

• Rays are traced recursively

```c
color c = trace(point p, vector d, int step)
{
    if( step > maxstep ) return background_color;
    point q = intersect(p, d, status);
    if( status == light_source ) return light_color;
    if( status == no_intersection ) return background_color;

    normal n = normal(q);
    vector r = reflect(q, n);
    vector t = transmit(q, n);
    color local = phong(q, n, r);
    color reflected = trace(q, r, step+1);
    color transmitted = trace(q, t, step+1);

    return local + reflected + transmitted;
}
```

see E. Angel “Interactive Computer Graphics” textbook
Ray-Tracing

- Intersection tests are generally the most expensive part of ray-tracing

\[ n = \frac{\nabla f}{\|\nabla f\|} \]

ray-tracing implicit functions is ‘easy’

normal of level set surface is identical to normalized gradient
Ray-Tracing Implicit Functions

• We can often solve for intersections explicitly

\[ f(x, y, z) = f(p) = 0 \quad \text{and} \quad p(t) = p_0 + t(p_1 - p_0) \]

Solve \( f(p(t)) = 0 \) for \( t \)

• Otherwise, we step along the ray until sign changes
Ray-Tracing
Ray-Tracing

- Intersection tests with meshes or curves are more complicated

- We again make use of hierarchical concepts to improve efficiency
  - Bounding boxes
  - Spatial subdivision/partitioning (BSP trees, Octrees, etc.)
Ray-Tracing

• Rays cast during ray-tracing represent a sampling of the scene by infinitely thin rays. However, the pixel associated with a ray has a non-zero area.

• We know from texture mapping that such a sampling of high-frequency images results in artifacts (e.g., Moiré Patterns).

• Similar anti-aliasing strategies can be employed:
  • Super sampling (cast more rays per pixel)
  • Adaptive sampling (cast more rays, where ‘needed’)
  • Stochastic sampling
The Rendering Equation and Radiosity

- Modeling realistic global illumination by performing ray-tracing for diffuse surfaces is inefficient.
- Diffuse surfaces reflect light into all directions equally.

- Other approaches are more suitable if the complete rendering equation needs to be approximated.
The Rendering Equation

- The rendering equation

\[
L(x, x') = g(x, x') \cdot \left( L_e(x, x') + \int_S b(x, x', x'') L(x', x'') \, dx'' \right)
\]

**Spectral radiance**

‘Amount’ of light that reaches x from x'

**Light emission**

from x' to x

**Incoming spectral radiance function**

**Geometric term**

(attenuation)

(occlusion)

**Reflectance function**

How much light coming from x'' is reflected by x' to x
Radiosity

• The classic Phong lighting model does not account for diffuse-diffuse interactions
• Radiosity methods can handle perfectly diffuse surfaces

• Scene is divided into $n$ patches
• Radiosity equation has $n$ linear equations for $n$ unknown radiosities $b_i$

$$b_i = e_i + \rho_i \sum_{j=1}^{n} f_{ij} b_j$$
reflectivity of patch $i$
Form factors $f_{ij}$ between patches $i$ and $j$ are computed as

$$o_{ij} = \begin{cases} 
1, & \text{if } p_j \text{ visible from } p_i \\
0, & \text{otherwise}
\end{cases}$$

$$f_{ij} = \frac{1}{a_i} \int_{a_i} \int_{a_j} o_{ij} \frac{\cos \phi_i \cos \phi_j}{\pi r^2} \ da_i da_j$$
Radiosity