• project 5 concepts
• Pat Hanrahan: future of CG?
support code

• not jot
• written by Prof. Guskov
• command line raytracer: srt scene.sce rendering.tga
• “srt” = simple ray tracer
• scene: lights, camera, objects, ...
• output: targa image (see spec)
tasks

1. ray generation code
2. shading computations
3. interpolated normals for smooth shading
4. specular reflections
5. cylinder primitive
6. anti-aliasing
7. area lights
8. optimization: bounding sphere test
ray generation

• same camera model we've seen before

• parameters:
  - **e**: eye location
  - **u**: unit vector pointing right
  - **v**: unit vector pointing up
  - **w**: unit vector pointing behind us
  - rendering window width, height in pixels
  - field of view angle (in vertical direction)
  - distance to near plane
ray generation

\[(l, b, n)\]

\[(r, t, n)\]

eye coords

near plane
ray generation

- image maps to rectangle in near plane
- assume center of rectangle is (0,0)
- Q: what are (l,b) in terms of (r,t)?
ray generation

- say (0,0) is center of lower left pixel
- Q: what are coordinates at corners?
ray generation

• convert pixel coordinates \((i,j)\) to eye coords \((u,v,n)\) describing location on near clipping plane \((n\) is coordinate of near plane\)

• e.g. \((-\frac{1}{2}, -\frac{1}{2})\) in pixels maps to \((l,b,n)\) in eye coordinates

• world-space location \(s\) is then:
\[
s = e + uu + vv + wn
\]

• Q: what is the ray?
ray generation

• Q: what is the ray?
• A: \( r(t) = e + t(s - e) \)
ray generation

• Q: how to get l, r, t, b, n, f?
• e.g. simple.sce:

```plaintext
# camera
eyepos 0 -2 1.5    // e
eyedir 0 1 -0.4     // -w
eyeup 0.0 0.0 1.0   // used to find v
wdist 1.0          // distance to near plane
fovy_deg 50        // field of view vertically
```
ray generation

• Q: how to get l, r, t, b, n, f?
• A: given \( n = \text{wdist} \), and fovy \( \theta = \text{fovy}/2 \), find t:
ray generation

- $\tan(\theta) = \frac{t}{n}$
- solve for $t$
ray generation

- aspect ratio \( a = \text{image width/height} \)
- \( r = a \cdot t \)
shading

/// Returns the color from the shading computation using
/// the information in the hitinfo_t structure
/// level is the recursion level
XVecf RayTracerT::Shade(const hitinfo_t & hit, int level) {
    XVecf color(0.0f);

    // Ambient light contribution
    color = hit.m_mat.m_ca*hit.m_mat.m_cr;

    // YOUR CODE HERE
    // shading code here
    // iterate over the lights and collect their contribution
    // make a recursive call to Trace() function to get the reflections

    return color;
}
/// Returns the color from the shading computation using
/// the information in the hitinfo_t structure
/// level is the recursion level
XVecf RayTracerT::Shade(const hitinfo_t& hit, int level) {
    XVecf color(0.0f);

    // Ambient light contribution
    color = hit.m_mat.m_ca*hit.m_mat.m_cr;

    // YOUR CODE HERE
    // shading code here
    // iterate over the lights and collect their contribution
    // make a recursive call to Trace() function to get the reflections

    SceneT::LightCt::const_iterator li;
    for(li=m_scene.BeginLights(); li!=m_scene.EndLights(); ++li) {
        ...
    }
    return color;
}
shading

SceneT::LightCt::const_iterator li;
for(li=m_scene.BeginLights(); li!=m_scene.EndLights(); ++li) {
    // send ray to light
    // if hit any object before light, skip the light

    // get surface normal from hit
    // find n dot l
    // do diffuse computation using light color and
    // material diffuse color

    // add specular contribution from light

    // if material specular color is not black,
    // compute color along reflected ray via RayTracerT::Trace()
phong shading

- do per pixel normals
  - use barycentric coordinates (provided)
  - return interpolated normal within mesh triangle in `MeshT::Intersect()`
- if: `m_shade == PHONG_SHADE`
Cylinder primitive

• To implement any object, just need to define \texttt{IGel::Intersect()}

• First step: map ray from world space to object space

• E.g. Sphere (provided in support code):
  
  – Given sphere center \( c \) and radius \( r \), point \( p \) is on the sphere if \( |p - c|^2 = r^2 \).
  
  – Point \( p \) on the ray: \( p = e + td \)
  
  – Substitute in 1\textsuperscript{st} equation, solve for \( t \) via quadratic formula
Cylinder primitive

• In object space the cylinder is “canonical”, e.g. radius = 1, centered along z-axis, top at z = 1, bottom at z = 0

• to intersect: first intersect with infinite cylinder (no top or bottom)

• point \( p \) is on the cylinder if \( |p_{xy}|^2 = 1 \).

• Substitute \( p = e + td \), solve for \( t \)
Cylinder primitive

- If ray missed, skip (done).
- If \(0 \leq z \leq 1\), the ray hit the side (done).
- Else, check if ray hits top or bottom:
  - Find intersection with plane
  - See if result is inside unit circle
Antialiasing

• Modify RayTracerT::TraceAll():
• Outside of main loop over pixels:
  – if (m_opts.m_aasample>0) {
  – Create random samples within a generic pixel
  – Use jittered sampling (see text)
  – Create samples in [0,1]x[0,1] square representing locations within a pixel
  – For each area light:
    • Create random samples (similar method)
Antialiasing

- For each pixel:
  - if (m_opts.m_aasample>0) {
  - For each sample within a pixel
    • Create view ray
    • Compute color seen along the ray
    • Add up, divide by total number of rays
Area lights

• Define area light class in light.h
• Load area lights in loadscene.cpp
• Handle area lights in raytracer.cpp
  – For each pixel:
  – Before generating rays, shuffle the samples within each light (see text)
  – During loop over pixel samples, store current sample number in:
    RayTracerT::m_current_sample
Area lights

• In RayTracerT::Shade(), when iterating over lights, pass the current sample number to each light as a “hint”:
  • (*li)->HintSample(m_current_sample);
• It uses the corresponding jittered sample
• Because of shuffling, there is no correlation of sampling pattern within a pixel to the sampling pattern within the light
Bounding sphere test

- In MeshT::ComputeBV(), compute a bounding volume
- Find average location, max distance to average location
- Use these as sphere center, radius
- When sphere is created, call ComputeBV
- Use BallT (add member variable to MeshT class)
Bounding sphere test

• In MeshT::Intersect(), compute the ray in object space, then before iterating over mesh triangles, check:

```java
if(!m_bball.Intersect(ray, hitdummy))
    return false; // skip it!
// else check every triangle...
```
My problem

● Method for shuffling samples and passing around hints seems extra complicated
● At each pixel, we shuffle the samples in each light...
● Q: What is the point of all this?
My problem

• Q: What is the point of all this?
• A: So we can precompute the samples, and not have to call the random number generator too much.
My problem

• Q: But don't you have to call the random number generator a whole lot of times for the shuffling?

• A: Well... yes. D00d, I just work here!
My problem

• Q: And how expensive is it exactly to call the random number generator anyway?
• A: Um... I don't know, I never checked...
• Possible case of premature optimization?
Reasonable alternative

- Don't precompute any jittered samples
- Just compute random samples on the fly as needed
Pat Hanrahan keynote

• Realistic or Abstract Imagery: The Future of Computer Graphics?

http://www.graphics.stanford.edu/~hanrahan/talks/realistic-abstract