OUTLINE

• What do you have to do for this project?
• Ray Class
• Isect Class
• Requirements
• Tricks
• Artifact Requirement
• Bells and Whistles
WELCOME TO THE RAYTRACER PROJECT

- You have to implement:
  - Shading (has multiple parts)
  - Reflection and Refraction
  - Sphere Intersection
  - The ability to intersect triangles
    - Complex objects consist of a 3D mesh made up of many triangles
RAY CLASS

• A 3D ray is a fundamental component of a raytracer.

• ray $r$ (start position, direction, RayType)
  • enum RayType{VISIBILITY, REFLECTION, REFRACTION, SHADOW};
  • example: ray $r$(foo, bar, ray::SHADOW);

• $r$.at($t$), returns the position end point of the ray $r$
  • $t$: the distance from the start position
VEC.H, MAT.H: MATH FUNCTIONS

- vec.h gives useful tools for 2D, 3D, and 4D vectors:
  - Easy Vector Construction
    - eg. Vec3d x = Vec3d(0,0,0);
  - Basic operators are overridden
    - +,-, arithmetic, Vec3d v3 = v1 + v2
    - *, multiply by constant, Vec3d v3 = 2*v1;
    - *, dotproduct, eg. double dot = v1 * v2;
    - ^, crossproduct, eg. Vec3d cross = v1 ^ v2;
  - Other useful functionality, read vec.h for complete details
    - normalize(), length(), iszero()
**Isect Class**

- An isect represents the location where a ray intersects an object.
- Important member variables:

  ```
  const SceneObject *obj; // the object that was intersected.
  double t; // the distance along the ray where it occurred.
  Vec3d N; // the normal to the surface where it occurred
  Vec2d uvCoordinates; // texture coordinates on the surface. [1.0,1.0]
  Material *material; // non-NULL if exists a unique material for this intersect.
  const Material &getMaterial() const; // return the material to use
  ```
REQUIREMENT: SPHERE INTERSECTION

- Fill in Sphere::intersectLocal in SceneObjects\Sphere.cpp:
  - Return *true* if ray *r* intersects the canonical sphere (sphere centered at the origin with radius 1.0) in positive time.
  - Set the values of *isect* *i*:
    - *i.obj = this*
    - *i.setT(time of intersection)*
    - *i.setN(normal at intersection).*
REQUIREMENT: TRIANGLE INTERSECTION

• Fill in TrimeshFace::intersectLocal in SceneObjects\trimesh.cpp:

• Intersect r with the triangle abc:
  
  Vec3d &a = parent->vertices[ ids [0] ];
  Vec3d &b = parent->vertices[ ids [1] ];
  Vec3d &c = parent->vertices[ ids [2] ];

• return true if ray r intersects the triangle.

• More Help? See page linked to on project website
  
REQUIREMENT:
BLINN-PHONG SPECULAR-REFLECTION MODEL

• Fill in Material::shade in material.cpp:

• Refer to the RayTracing lecture:

• To sum over the light sources, use an iterator as described in the comments of the code.

• Need to implement Phong normal interpolation
REQUIREMENT: MULTIPLE LIGHT SOURCES

• Fill in PointLight::distanceAttenuation in light.cpp (DirectionalLight::distanceAttenuation is done for you).

• Use the alternative described in the ray-tracing lecture where

\[
\begin{align*}
  a &= \text{constantTerm} \\
  b &= \text{linearTerm} \\
  c &= \text{quadraticTerm}
\end{align*}
\]

• These terms are defined in light.h.
REQUIREMENT: SHADOW ATTENUATION

• Fill in DirectionalLight::shadowAttenuation and PointLight::shadowAttenuation in light.cpp.

• The ray-tracing lecture shows you where to insert this factor into the Blinn-Phong equation (A shadow for each light).

• Rather than simply setting the attenuation to 0 if an object blocks the light, accumulate the product of k_t’s for objects which block the light (use the prod function from the vec.h).

• Extra Credit: Better shadow handling (caustics, global illumination, etc.)
REQUIREMENT: REFLECTION

- Modify RayTracer::traceRay in RayTracer.cpp to implement recursive ray tracing which takes into account reflected rays.

- See lecture notes.
REQUIREMENT: REFRACTION

• Modify RayTracer::traceRay in RayTracer.cpp
  • create refracted rays.
• Remember Snell’s law, be careful about total internal refraction and the normal direction when the ray is exiting a material into air
• You can test with simple/cube_transparent.ray
• Unlike reflection, this routine has several cases to consider:
  • an incoming ray
  • an outgoing ray
  • totally internally refracted ray.

\[ n_{\text{glass}} = 1.5 \]

\[ n_{\text{air}} = 1.0003 \]
TIPS

- Use the sign of the dot product `r.getDirection()` with `i.N` to determine whether you’re entering or exiting an object.
- Use `RAY_EPSILON` (which is defined as 0.00001) to account for computer precision error when checking for intersections.
THE DEBUGGER TOOL

• shipped with the skeleton code
• http://www.cs.washington.edu/education/courses/csep557/13wi/projects/trace/extra/debug.html
ARTIFACT REQUIREMENT

• Draw a pretty picture!
• One JPEG/PNG image traced with your Ray Tracer submitted for voting.
• Has to be a (somewhat) original scene
• For each image submitted for voting, a short .txt description of the scene or special features.
• Examples of each bell/whistle implemented with an accompanying readme.txt specifying which image demonstrates which feature (and where/how).
RAY TRACING YOUR SURFACE OF REVOLUTION

- Render your surface of revolution to earn one easy extra point
- Using this code snippet to write triangle mesh into a file
- Using this .ray file as a template
  - It contains default lighting of modeler
  - Replace polymesh{} part with your own surface of revolution
- Render your new .ray file in tracer
SAMPLE RESULTS

With texture mapping
BELLS AND WHISTLES

- **TONS of Awesome Extra Credit!!!**
- Antialiasing – A must for nice scenes (to render scenes without “jaggies”)
- Interpolate trimesh material properties – will make them look nicer
- Environment/Texture/Bump Mapping – Relatively easy ways to create complex, compelling scenes
- Single Image Random Dot Stereograms
- Depth of field, Soft shadows, Motion blur, Glossy reflection – most images we’re used to have at least one of these effects

**NOTE**: Please add control boxes for substantial ray tracing modifications so the required extensions are easily gradable
- see sample solution style
- Especially things like anti-aliasing, glossy reflection, soft shadows, etc.
3D AND 4D FRACTALS
CONSTRUCTIVE SOLID GEOMETRY

- Allows for complex objects while still just intersecting simple primitives
USING PLY MODELS

- ply is one of the standard formats for 3D models
  

- There are a lot of ply models available online

- We provide a simple tool that converts ply models into .ray files.

- You still need to add lighting and material property.
THE DREADED MEMORY LEAK!!!

- A Memory Leak can (and probably will) ruin your night of rendering hours before the artifact is due.
- depth 10, Anti-Aliasing, HUGE Image ➔ ALL MEMORY CONSUMED BY ray.exe
  - at 1.8 GB on Hardware lab machines
- Cause: not calling free after allocating memory
  - Object constructors, vector (array) creation
- It is HIGHLY RECOMMENDED you have no memory leaks
- Solution: call the “delete [object]” on ANYTHING you create that temporarily
  - i.e. 3 byte temporary vectors in rayTrace function