Ray Tracer

Spring 2005 Help Session

Outline

- Project Web Resources
- Ray Class
- Isect Class
- Requirements
- Tricks
- Artifact Requirement
- Bells and Whistles

Project Web Page

- Description of sample scenes
- Roadmap
  - Overview of files
  - STL Information
  - List of useful equations
- File format description
- Debugging display documentation
- Triangle intersection handout

ray Class

- ray r (start position, direction, RayType)

  enum RayType{VISIBILITY, REFLECTION,
  REFRACTION, SHADOW};

- r.at(t), a method that determines the position of the ray r as a function of *t*, the distance from the start position.

isect Class

- An isect represents the location where a ray intersects a specific 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
```

Requirements

Sphere Intersection

Fill in Sphere::intersectLocal in 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).

Requirements

Triangle Intersection

Fill in TrimeshFace::intersectLocal in 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] ];
```

Set isect i and return *true* if ray r intersects the plane containing triangle abc and the intersection is within the triangle.

See handout linked from project page

Requirements

Phong specular-reflection model

Fill in Material::shade in material.cpp:

Refer to slide 20 of the shading lecture.

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

Requirements

Contribution from multiple light sources

Fill in PointLight::distanceAttenuation in light.cpp (DirectionalLight::distanceAttenuation is already done for you). Use the alternative described in slide 19 of the shading lecture where

a = constantTerm

b = linearTerm

c = quadraticTerm

in light.h.

Requirements

Shadow Attenuation

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

Take into account shadow attenuation in the f_atten term in the Phong model as suggested in the ray-tracing lecture.

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 vector package).

See Foley, et. al. Section 16.12 in course reader – this particular method is not really covered in lecture slides

Better ways to handle shadows (caustics, global illumination, etc.) get extra credit

Requirements

Reflection

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

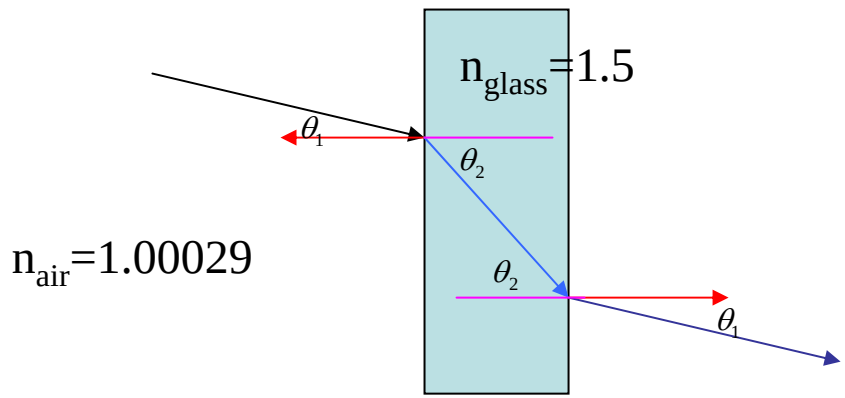
See Foley, et. al. in course reader and lecture slides.

Requirements

Refraction

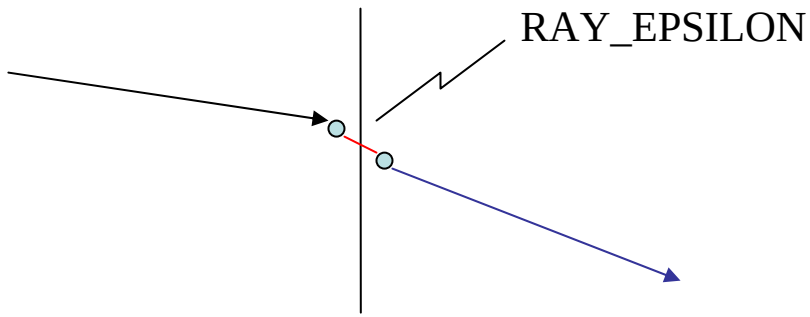
Modify RayTracer::traceRay in RayTracer.cpp to implement recursive ray tracing which takes into account refracted rays.

Remember Snell's law and watch our for total internal refraction.



Tricks

- 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`** to account for computer precision error when checking for intersections



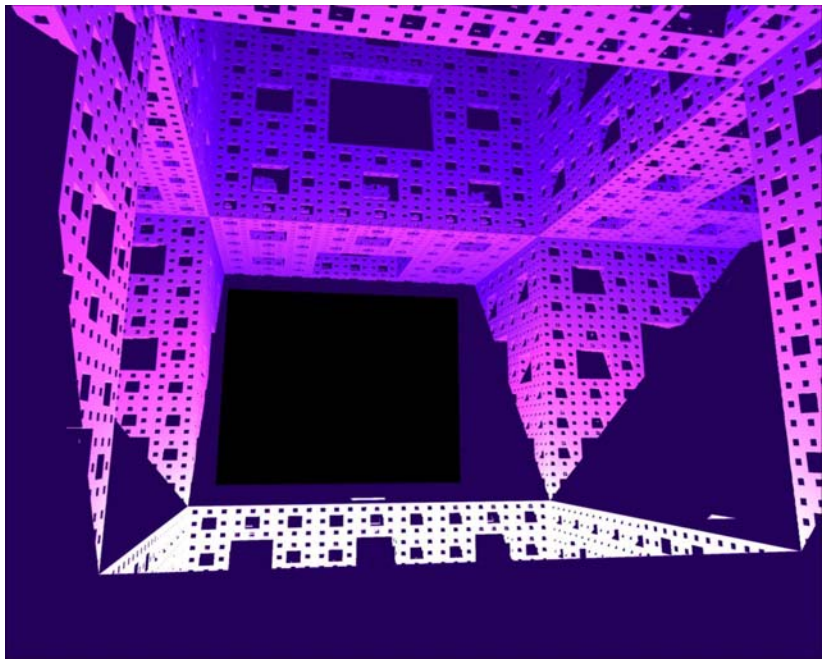
Artifact Requirements

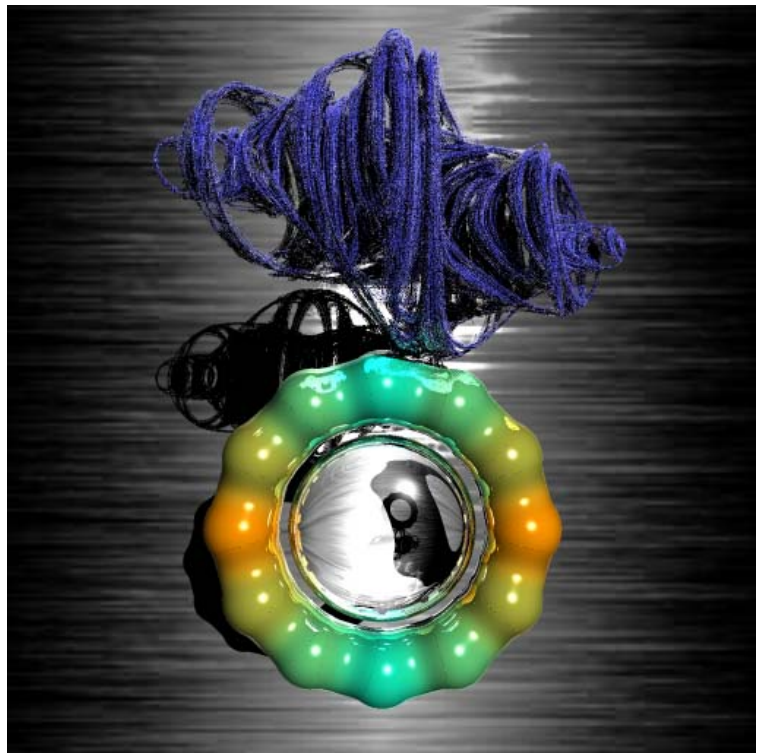
- One (or two) JPEG images traced with your Ray Tracer per group submitted for voting.
- Has to be a (somewhat) original scene
- For each image submitted for voting, a short .txt description of the scene or special features.
- Examples of each bell/whistle implemented with an accompanying readme.txt specifying which image demonstrates which feature (and where/how).

Bells and Whistles

- 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 – I have no idea, but they look cool!
- Depth of field, Soft shadows, Motion blur, Glossy reflection – most images we’re used to have at least one of these effects

3D and 4D Fractals





Constructive Solid Geometry

- Allows for complex objects while still just intersecting simple primitives