RAY TRACER

Autumn 2010 Help Session

Christopher Raastad

OUTLINE

- Project Web Resources
- What do you have to do for this project?
- Ray Class
- Isect Class
- Requirements
- Tricks
- Artifact Requirement
- Bells and Whistles

PROJECT WEB PAGE

http://www.cs.washington.edu/education/courses/cse457/CurrentQtr/projects/trace/

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

WELCOME TO THE RAYTRACER PROJECT

- Basically, you are given a RayTracer
- 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 the 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), a method that determines the position of the ray r as a function of *t*, the distance from the start position (t*direction vector)
  - r.at(t) => where the end of the ray points at (a distance t away from the start point)

VEC.H, MAT.H: PROVIDED MATH LIBRARIES

- vec.h gives us useful tools for 2D, 3D, and 4D vectors:
  - Easy Vector Construction
    - eg. Vec3d x = Vec3d(0,0,0); or Vec4d y = Vec4d(255,255,255,1);
  - +,-,*,DotProduct and CrossProduct are operator overloaded operations
    - +,-,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()
- mat.h is very similar but for matrix operations not heavily used in this project

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
```

- This data structure is used to record the details of a ray's intersection with an object as implemented in an object's intersection routine.

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] ];
```

- Set up isect i as in the sphere intersection and return *true* if ray r intersects the plane containing triangle abc and the intersection is within the triangle.
- More Help? See page linked to on project website
  - http://www.cs.washington.edu/education/courses/457/CurrentQtr/projects/trace/extra/triangle_intersection.pdf

REQUIREMENT: BLINN-PHONG SPECULAR-REFLECTION MODEL

- Fill in Material::shade in material.cpp:
- Refer to the RayTracing lecture:
  - http://www.cs.washington.edu/education/courses/457/CurrentQtr/lectures/ray-tracing.pdf
- To sum over the light sources, use an iterator as described in the comments of the code.
- CAUTION! If you are inside an object, the object's normal will point outside. You will need to flip that normal for any shading, reflection, or refraction.
  - (Unless of course you like funky images and less points…)

REQUIREMENT: CONTRIBUTION FROM MULTIPLE LIGHT SOURCES

- Fill in PointLight::distanceAttenuation in light.cpp (DirectionalLight::distanceAttenuation is already done for you).
- Use the alternative described in the ray-tracing lecture where

  a = constantTerm

  b = linearTerm

  c = quadraticTerm
- 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).
- Count each intersection with an object by the shadow ray (which may include entering and exiting).
- See Foley, et. al. Section 16.12– this particular method is not really covered in lecture slides
- Extra Credit: Better shadow handling (caustics, global illumination, etc.)
- Again, Check out the ray-tracing lecture:
  - http://www.cs.washington.edu/education/courses/457/CurrentQtr/lectures/ray-tracing.pdf

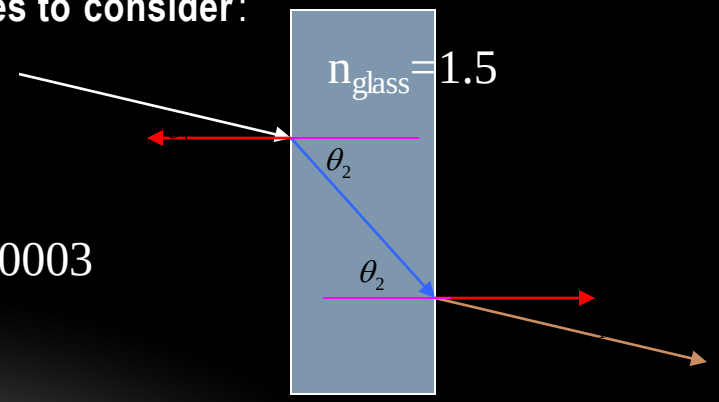
REQUIREMENT: REFLECTION

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

- See Foley, et. al. and lecture slides.

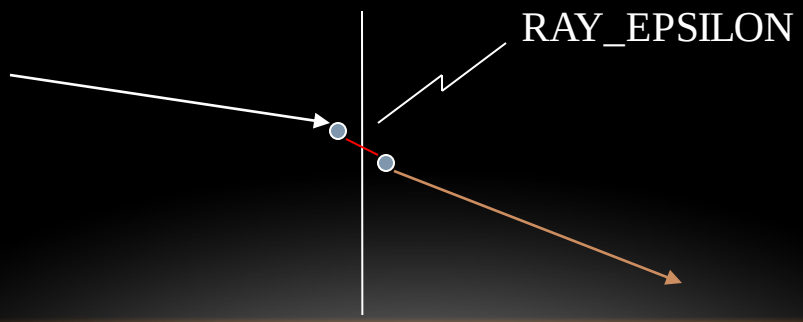
REQUIREMENT: REFRACTION

- Modify RayTracer::traceRay in RayTracer.cpp to implement recursive ray tracing which takes into account refracted rays.
- Remember Snell's law, watch out for total internal refraction, and consider the case when the ray is exiting a material into air (think about the direction of the normal)
- You can test with simple/cube_transparent.ray
- **Warning: Unlike reflection, this routine has several cases to consider:**
  - **an incoming ray**
  - **an outgoing ray**
  - **totally internally refracted ray.**



TIPS AND TRICKS

- Use the sign of the dot product r.getDirection() with i.N to determine whether you're entering or exiting an object
- DON'T WRITE TOO MUCH CODE WITHOUT TESTING!!!!
  - Lots of dependencies, you need to know what works to proceed.
- **RAY_EPSILON RAY_EPSILON RAY_EPSILON**
- Use **RAY_EPSILON** (which is defined as 0.00001) to account for computer precision error when checking for intersections



- YHBW= you have bee warned

THE DEBUGGER TOOL IS YOUR BEST FRIEND

ARTIFACT REQUIREMENT

- Draw a pretty picture!
- One JPEG/PNG image per person 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).

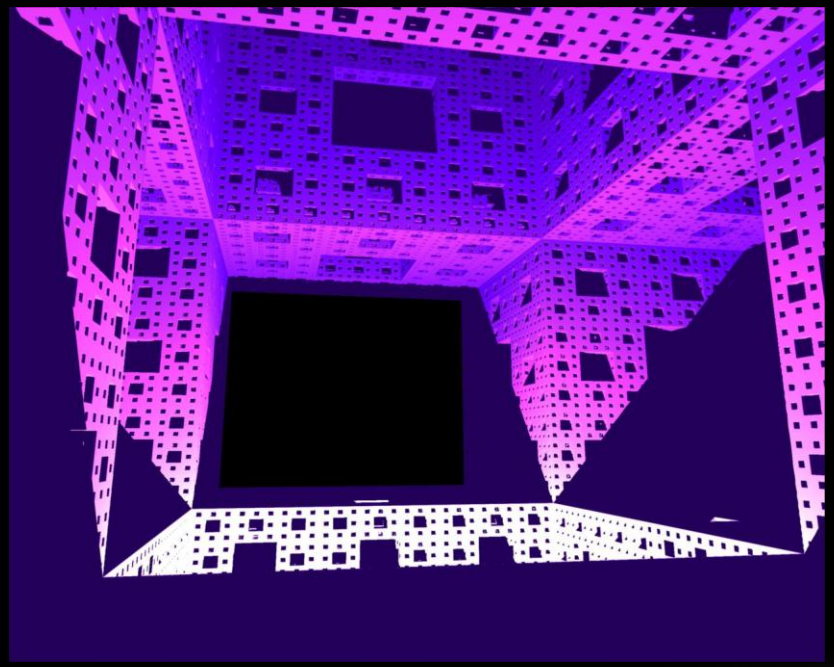
THE DREADED MEMORY LEAK!!!

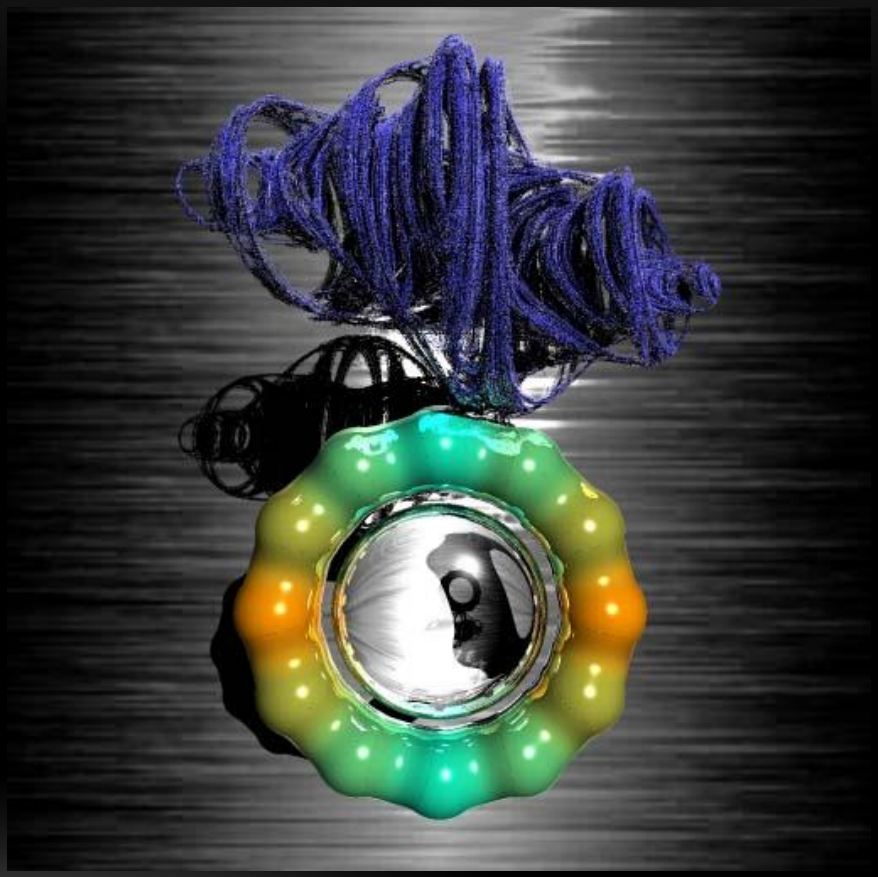
- A Memory Leak can (and probably will) ruin your night of rendering hours before the artifact is due.
- depth 10, Anti-Aliasing, HUGE Image ➔ ALL MEMORY CONSUMED BY ray.exe
  - at 1.8 GB on Hardware lab machines
- Definition: unstoppable (except program termination) increase in acquired memory (most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

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 – 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
- **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