Photon mapping: starter code

Christopher Twigg
23 March 2005
Starter code basics

- Prerequisite for photon mapping: ray tracer
- We are providing you with some starter code
- Various parts of this code are by
  - University of Washington graphics group
  - Henrik Wann Jensen
  - Me (Chris Twigg)
What’s in there?

• Full Whitted-style ray tracer, supports
  – Arbitrary polymesh geometry
  – AABB acceleration
  – Texture mapping
  – Antialiasing using distributed rays
• User interface for visualizing scenes
• Completely portable code, we support both Windows and Linux
Directory Structure

• `src/`
  – `fileio/` : utility stuff for the parser
  – `parser/` : parsing of .ray files
  – `scene/` : scene graph, shading, lighting
  – `SceneObjects/` : geometry & intersection
  – `ui/` : user interface (graphical and command-line)
  – `vecmath/` : wrappers for the VL library

• `scenes/` : sample scenes
Visual Code Overview

Recall how ray tracing works…
Now, let’s see how we do it in the starter code:
Now, let’s see how we do it in the starter code:

- `Light::getDirection`
- `Material::shade`
- `SceneObject::intersect`
- `Scene::intersect`
- `Camera::rayThrough(i, j)`
- `RayTracer::traceRay`
- `Scene::intersect`
The scene graph

Scene

SceneObject
  TransformNode
    Material
SceneObject
  TransformNode
    Material
SceneObject
  TransformNode
    Material
Tracing rays...

**RayTracer::traceRay**

- Check if ray intersects scene

  >>> Scene::intersect

  >>> position
  >>> t value
  >>> object

- Compute local Phong model

  >>> Material::shade

  >>> isect

  >>> color

  if( reflective )
  recurse on reflected ray

  if( refracted )
  recurse on refracted ray

  return $c_{\text{reflect}} + c_{\text{refract}} + c_{\text{Phong}}$

**RayTracer::traceRay**
User Interface

Clicking here traces a single ray through the scene, so you can set breakpoints, etc.
User Interface

- Think of creative ways to use the interface
- Visual debugging is a useful skill in graphics...
Implementing: soft shadows

• For soft shadows, we need to distribute our shadow rays over the light source area

• A square with emissivity is our only area light source:

```plaintext
// light
translate( 2.78, 5.48, 2.295,
scale( 1.3, 1.0, 1.05,
    rotate( 1.0, 0.0, 0.0, 1.5708,
        square {
            material = {
                emissive = (250.0, 250.0, 250.0);
                diffuse = (0.750, 0.750, 0.750);
            }
        }
    )
)
```

Implementing: soft shadows

- Right now, `Square::getDirection` acts like a point light:

```cpp
LightProperties Square::getDirection(
    const Vec3f& P,
    unsigned int index ) const
{
    // Treat it like a point light
    vl::Vec3f position = transform
        ->localToGlobalCoords(vl::Vec3f(0,0,0));
    Vec3f d = (position-P);
    float dL = len(d);
    d /= dL;
    float t = dot(position - P, d);
    float falloff = 1.0/(4*pi*dL*dL);
    return LightProperties(
        ray(P, d, index, ray::SHADOW),
        t, falloff );
}
```
Implementing: soft shadows

Points on the square are defined in *local* coordinates.

We use `transform->localToGlobalCoords` coordinates to get worldspace coords.
Implementing: photon maps

- Scene::recomputePhotonMaps is called before any rays are traced
- Can access photon map from within Material::shade (use the Scene* that is passed in)
Implementing: photon maps

• Henrik’s photon map implementation (from the book) is included (class `Photon_map`)

• Key functions:
  – `store`: store a photon in the map
  – `scale_photon_power`: scale all photons (since last time function called) by same value
  – `balance`: balance the kd-tree, must be called after all photons are stored but before obtaining any irradiance estimates
  – `irradiance_estimate`
Implementing: photon maps

Pseudocode:

```c
foreach Light* L
{
    for( i = 1 to nPhotons )
    {
        {power, position, direction} =
            trace_photon( L, L->randomDir(), L->color() );
        photonMap_->_store( power, position, direction );
    }

    photonMap_->_scale_photon_power( 1.0/nPhotons );
}
photonMap_->_balance();
```
Useful functions

• Sources of randomness:
  – RayTracer::uniform01()
  – RayTracer::uniformOnSphere()
  – RayTracer::uniformInt(N)

• Note: Can access these from anywhere with the following syntax
  – traceUI->rayTracer().uniform01();
Sample scenes

Soft shadows
Global illum.
cornellBox.ray

+refractive caustic
(sphere only)
cornellBoxSpheres.ray

+refractive caustic
(arbitrary geom.)
cornellBoxWater.ray

+reflective caustic
(arbitrary geom.)
cornellBoxReflective.ray

probably optional
(pending Doug’s decision)

easiest ➤ hardest
Make your own!

Jernej Barbic

Kris Poppendorf

Adam Kushner
.ray file format

Arbitrary, nested transforms:

```plaintext
// light
translate( 2.78, 5.48, 2.295,
    scale( 1.3, 1.0, 1.05,
        rotate( 1.0, 0.0, 0.0, 1.5708,
            square {
                material = {
                    emissive = (250.0, 250.0, 250.0);
                    diffuse = (0.750, 0.750, 0.750);
                }
            }
        )
    )
) )
```
Scoping is allowed:

```plaintext
scale( 0.01, 0.01, 0.01,
   {
      sphere {}  
      box {}
   }
)
```
.ray file format

Arbitrary .obj geometry:

```plaintext
cell {
  material = {
    diffuse = (0.750, 0.750, 0.750);
  }
  objfile = "box.obj";
  objgroup = "Cube";
}
```

note that you must still specify material params
.ray file format

Texture map any material attribute:

```plaintext
polymesh {
    material = {
        // map the diffuse color
        diffuse = map( "checkerboardDark.bmp" );
    }
    objfile = "torus.obj";
    objgroup = "pTorus1";
}

... material = {
    // map the specular color
    specular = map( "checkerboardDark.bmp" );
    shininess = 100;
}
...
Many more possibilities…

Be creative!

source: FACHE Sebastien

source: Henrik Wann Jensen

source: Henrik Wann Jensen