Project 3 – Path Tracing

CSE 168: Rendering Algorithms, Spring 2017

Description
Add antialiasing and path tracing of diffuse surfaces and Fresnel metals to your renderer.

Project 3 is due by 5:00 pm, Wednesday May 10, 2017.

Antialiasing
For the antialiasing, the Camera should have three additional functions:

SetSuperSample(int xsamples, int ysamples);  // Supersample each pixel in a x*y grid
SetJitter(bool enable);                 // Enable/disable jittering of each supersample
SetShirley(bool enable);               // Enable/disable Shirley mapping

All three of these functions should work in any combination, i.e., for each pixel, you will do a loop like this:

for(each supersample) {
    Compute subpixel position (0...1 range in x & y)
    If(EnableJitter)
        Apply jitter to subpixel position
    If(EnableShirley)
        Apply Shirley mapping to subpixel position

    Turn subpixel position into pixel position & trace path
}
FinalColor = average of all path colors

Path Tracing
To implement path tracing, I’d suggest by starting with implementing a Material base class if you haven’t already. You will need derived materials for LambertMaterial and FresnelMetalMaterial.

I suggest adding a virtual function to the Material class:

virtual void GenerateSample(const Intersection &isect, const glm::vec3 &inDir, glm::vec3 &outDir, Color &outColor);

The generate sample function takes an incoming Intersection class (to provide the surface position & normal) and an incoming beam direction (inDir) and produce a randomly distributed output direction (outDir) as well as an output color (outColor). For the Fresnel metal, for example, the generated sample
direction would simply be a reflection across the surface normal, and the color would compute the Fresnel equations for the given angles.

For the diffuse Lambert material, the GenerateSample() function would generate a cosine-weighted hemispherical direction, distributed around the surface normal. The output color would just be the (constant) color of the material.

To compute the shaded color from a path, we shoot the first ray of the path into the scene from the camera. When we hit a surface, we compute the color reflected back along the path as the sum of the direct illumination and the indirect illumination on that surface. If no surface is hit, then we just use the sky color.

To estimate the direct illumination, you can loop through all lights and sum up the direct illumination as in project 2 by tracing one shadow ray to each light. Optionally, you can generate a single shadow ray to one randomly selected light, selected according to a distribution weighted by the potential intensity of the light. The resulting illumination is then scaled by 1/prob, where prob is the probability of selecting the light that was chosen.

To compute the indirect light, we generate a single reflected sample by calling the GenerateSample() function on the material the ray hit. We then trace a new ray in that direction to extend the path, which may recursively repeat the process up to some desired path length. The shaded color returned from that ray is scaled by the outColor from the material and added to the direct illumination, and finally returned as the total color.

To terminate the paths, you can either choose a fixed path length (maybe around 5), or you can use the Russian Roulette algorithm as presented in the slides.

**Grading**
This project is worth 15 points:

- Path tracing diffuse surfaces 5
- Path tracing Fresnel metals 5
- Antialiasing 5
- Total 15
**Sample Image**

Project 3 should generate the following image (this used 10x10 paths per pixel, with up to 10 bounces):

![Sample Image](image)

**Project 3 Function**

Project 3 should be able to be run with the following sample code (or something very similar):

```c
void project3() {
    // Create scene
    Scene scn;
    scn.SetSkyColor(Color(0.8f,0.9f,1.0f));

    // Create ground
    LambertMaterial groundMtl;
    groundMtl.SetColor(Color(0.25f,0.25f,0.25f));

    MeshObject ground;
    ground.MakeBox(2.0f,0.11f,2.0f,&groundMtl);
    scn.AddObject(ground);

    // Load dragon mesh
```
MeshObject dragon;
dragon.LoadPLY("dragon.ply");

// Create box tree
BoxTreeObject tree;
tree.Construct(dragon);

// Materials
LambertMaterial white;
white.SetColor(Color(0.7f,0.7f,0.7f));

LambertMaterial red;
red.SetColor(Color(0.7f,0.1f,0.1f));

MetalMaterial metal;
m metal.SetColor(Color(0.95f,0.64f,0.54f));

const int numDragons=4;
Material *mtl[numDragons]={<&white,&metal,&red,&white>};

// Create dragon instances
glm::mat4 mtx;
for(int i=0;i<numDragons;i++) {
    InstanceObject *inst=new InstanceObject(tree);
    mtx[3]=glm::vec4(0.0f,0.0f,0.3f*(float(i)/float(numDragons-1)-0.5f),1.0f);
    inst->SetMatrix(mtx);
    inst->SetMaterial(mtl[i]);
    scn.AddObject(*inst);
}

// Create lights
DirectLight sunlgt;
sunlgt.SetBaseColor(Color(1.0f,1.0f,0.9f));
sunlgt.SetIntensity(1.0f);
sunlgt.SetDirection(glm::vec3(2.0f,-3.0f,-2.0f));
scn.AddLight(sunlgt);

// Create camera
Camera cam;
cam.SetResolution(640,480);
cam.SetAspect(1.33f);
cam.LookAt(glm::vec3(-0.5f,0.25f,-0.2f),glm::vec3(0.0f,0.15f,0.0f));
cam.SetFOV(40.0f);
cam.SetSuperSample(10,10);
cam.SetJitter(true);
cam.SetShirley(true);

// Render image
cam.Render(scn);
cam.SaveBitmap("project3.bmp");
}
Notes

Instanced Materials

Notice the addition of the InstanceObject::SetMaterial() function. This is intended to allow instances to override the material of the objects they instance. To implement this, add a Material* to InstanceObject that defaults to null. In the Intersect() function, after an intersection is detected, it should set the hit.Mtl pointer to the instance material if it is non-null. If it is null, it should leave the hit.Mtl alone as it will be set to the individual triangle material.

Noise

To speed up testing, you can render with a lower sampling rate, such as 2x2. In this case, the rendering will have a lot of noise but should only take a few seconds rather than a couple minutes at 10x10. This is how the image would look with only 2x2 samples:
Extra Credit
For 1 extra credit point total, you must do all of the following:

- Finite camera aperture (depth of field)
- Motion blur (initial and final matrix settable in InstanceObject)

[sample picture coming soon]