Project 3 – Ashikhmin BRDF

CSE 168: Rendering Algorithms, Spring 2014

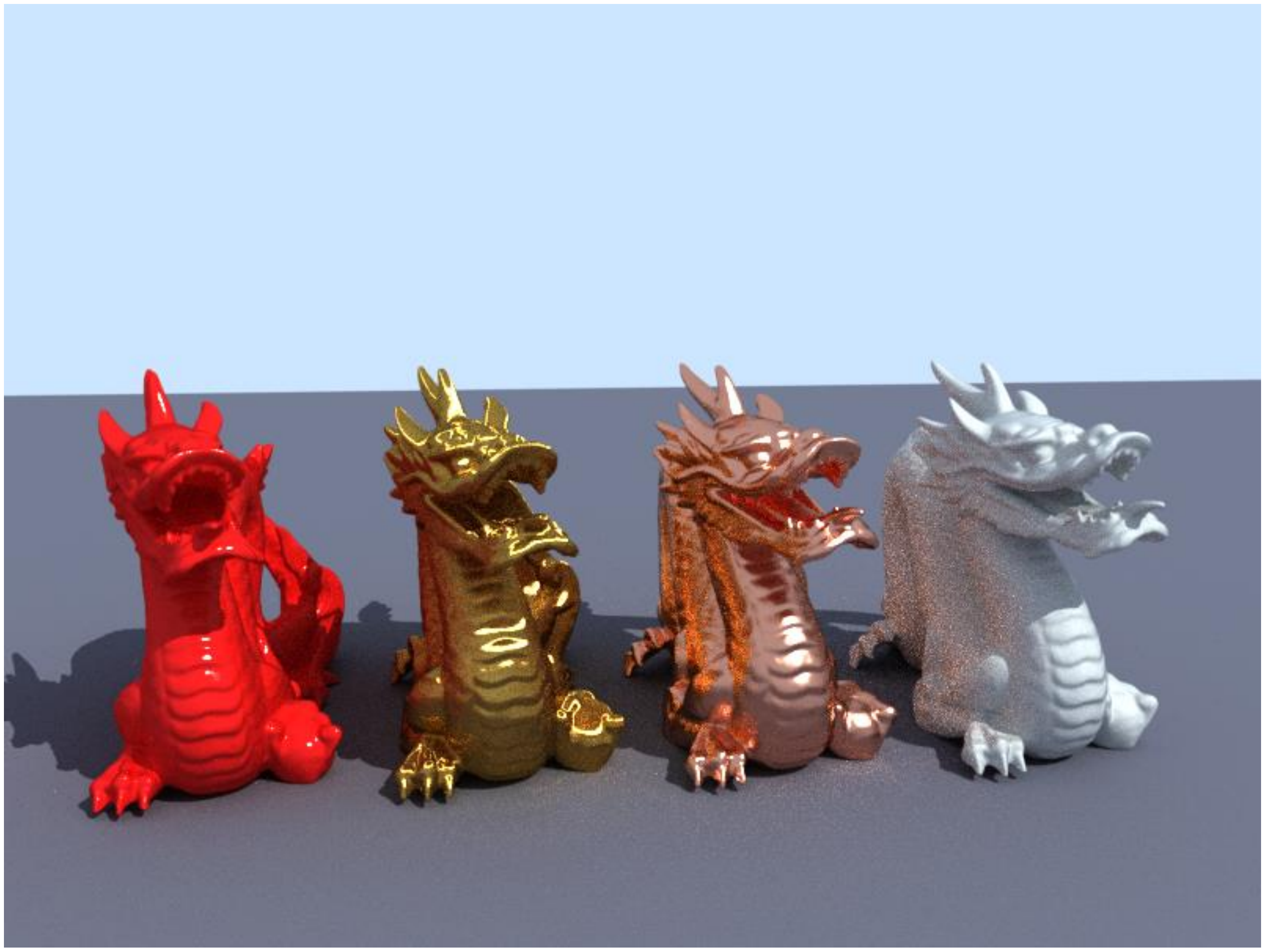
Description

Add support for multiple derived materials and implement the Lambert and Ashikhmin models. Both materials should implement the appropriate sampling routines to generate random output rays from a given input direction.

Also add support for recursive ray tracing of reflections. Use the BRDF sampling routines to generate reflection rays. You will need to shoot many camera rays per pixel and average the results.

Sample Image

Project 3 should generate the following image (this used 100 rays per pixel, with up to 10 bounces):



Project 3 Function

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

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

	// Materials
	const int nummtls=4;
	AshikhminMaterial mtl[nummtls];

	// Diffuse
	mtl[0].SetSpecularLevel(0.0f);
	mtl[0].SetDiffuseLevel(1.0f);
	mtl[0].SetDiffuseColor(Color(0.7f,0.7f,0.7f));

	// Roughened copper
	mtl[1].SetDiffuseLevel(0.0f);
	mtl[1].SetSpecularLevel(1.0f);
	mtl[1].SetSpecularColor(Color(0.9f,0.6f,0.5f));
	mtl[1].SetRoughness(100.0f,100.0f);

	// Anisotropic gold
	mtl[2].SetDiffuseLevel(0.0f);
	mtl[2].SetSpecularLevel(1.0f);
	mtl[2].SetSpecularColor(Color(0.95f,0.7f,0.3f));
	mtl[2].SetRoughness(1.0f,1000.0f);

	// Red plastic
	mtl[3].SetDiffuseColor(Color(1.0f,0.1f,0.1f));
	mtl[3].SetDiffuseLevel(0.8f);
	mtl[3].SetSpecularLevel(0.2f);
	mtl[2].SetSpecularColor(Color(1.0f,1.0f,1.0f));
	mtl[3].SetRoughness(1000.0f,1000.0f);

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

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

	// Create dragon instances
	Matrix34 mtx;
	for(int i=0;i<nummtls;i++) {
		InstanceObject *inst=new InstanceObject(tree);
		mtx.d.Set(0.0f,0.0f,-0.1f*float(i));
		inst->SetMatrix(mtx);
		inst->SetMaterial(&mtl[i]);
		scn.AddObject(*inst);
	}

	// Create ground
	LambertMaterial lambert;
	lambert.SetDiffuseColor(Color(0.3f,0.3f,0.35f));
```

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

        // Create lights
        DirectLight sunlgt;
        sunlgt.SetBaseColor(Color(1.0f, 1.0f, 0.9f));
        sunlgt.SetIntensity(1.0f);
        sunlgt.SetDirection(Vector3(2.0f, -3.0f, -2.0f));
        scn.AddLight(sunlgt);

        // Create camera
        Camera cam;
        cam.LookAt(Vector3(-0.5f,0.25f,-0.2f),Vector3(0.0f,0.15f,-0.15f));
        cam.SetFOV(40.0f);
        cam.SetAspect(1.33f);
        cam.SetResolution(800,600);
        cam.SetSuperSample(10);

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

Grading

This project is worth 12 points:

-	Ashikhman BRDF Evaluation	4
-	Ashikhman BRDF Sampling	2
-	Lambert BRDF with sampling	2
-	Reflections rays	4
-	Total	12

Notes

Ashikhmin BRDF

Details on the Ashikhmin BRDF can be found at:

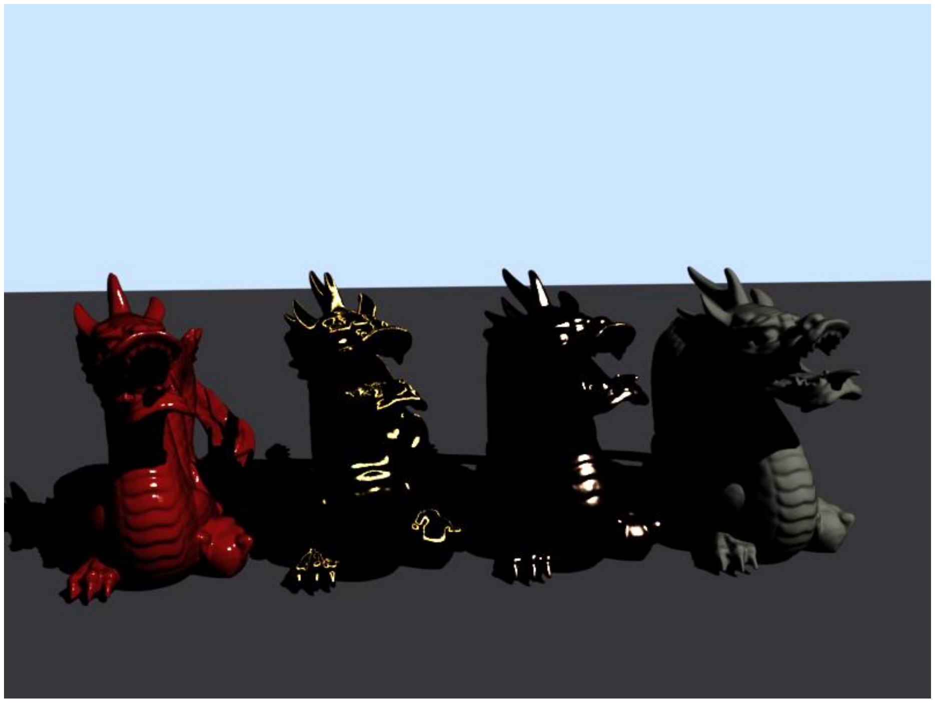
http://www.cs.utah.edu/~michael/brdfs/jgtbrdf.pdf

Supersampling

The cam.SetSuperSample(10) is meant to set the camera to do 10x10 supersampling. If you haven't implemented antialiasing, then you can still just use have it generate 100 rays through the center of the pixel and average the colors (before converting to 24-bit integer).

Forward BRDF Evaluation

To get things started, I suggest first implementing the forward evaluation of the BRDF and ignoring the sampling function and recursive ray reflections. If you render the image without any reflections and just computing the direct lighting on the BRDF, it will look like this:



This shows how the BRDF looks when lit from a single directional light only. The rest of the color in the final image comes from reflection rays hitting other surfaces in the environment.

Lambert BRDF

The Lambert material provided in the sample code should be a good start. All that is needed is a GenerateSample() function that uses the cosine weighted hemisphere mapping. As it is much simpler than the Ashikhmin BRDF, it might be a good idea to try to get the project running with Lambert first.

Tangent Vectors

You will need to support tangent vectors in the Intersection class. I suggest adding a Vector3 TangentU,TangentV; to the Intersection class. Computing correct tangents requires proper texture coordinates, and the sample models do not have any. Therefore, I suggest adding something like the following in Triangle::Intersect() after computing the normal:

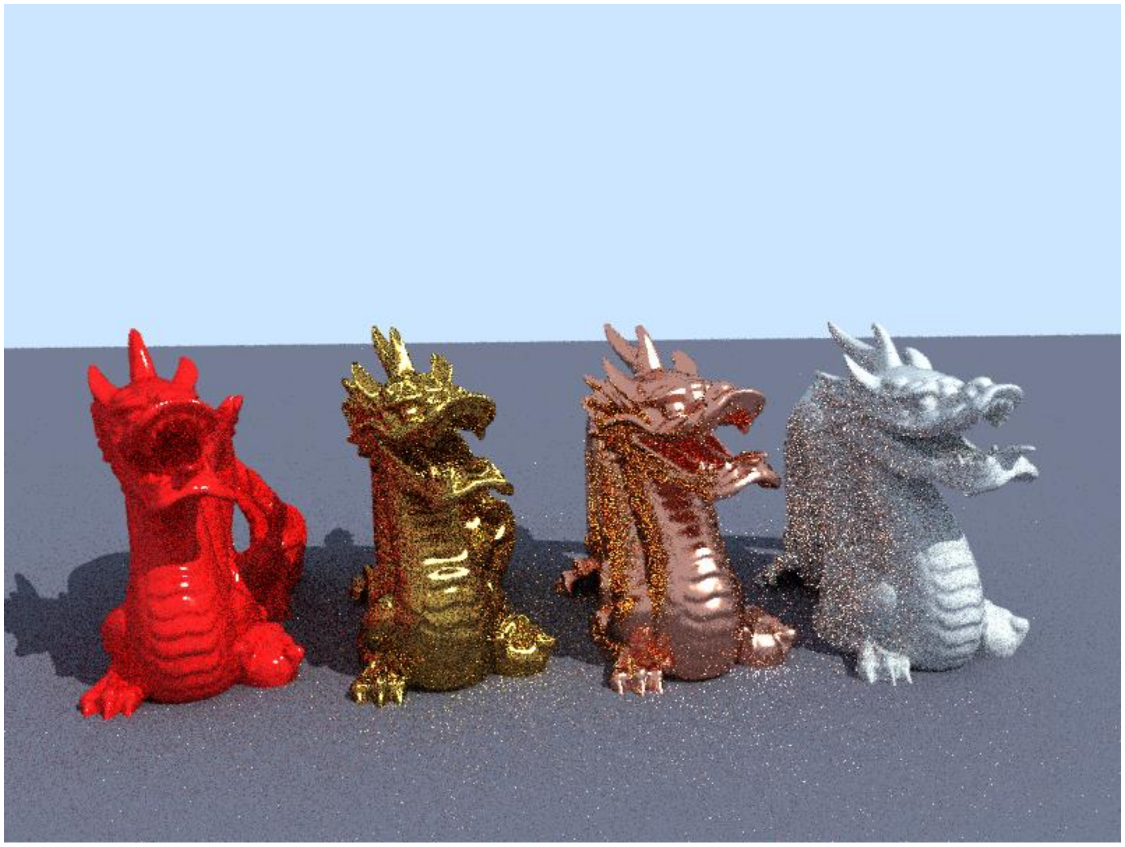
```
hit.TangentU.Cross(rtVector3::YAXIS,hit.Normal);
hit.TangentU.Normalize();
hit.TangentV.Cross(hit.Normal,hit.TangentU);
```

Instanced Materials

Notice the addition of the InstanceObject::SetMaterial() function. This is intended to allow instances to override the material of the object's 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

The rendering will have a lot of noise if you don't take many samples. While testing, you will probably want to use far fewer samples per pixel to keep the render speed fast. 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)

And then render an image something like these: