Project 1 – Simple Ray Tracer

CSE 168: Rendering Algorithms, Spring 2014

Description

Write a simple ray tracer capable of rendering boxes and instances and outputting the image to a BMP file. It should support diffuse lighting (without shadows) from both point & directional lights. It should also have a simple camera model with adjustable camera matrix, field of view, and output resolution.

It should be able to run the code listed below in the *Project 1 Function* section. If you use C++, it should be able to run the code with little or no modification. If you use a different programming language, or a different vector library, or choose to name things differently, you can make minor changes.

Make sure to compile with optimization on. In VisualStudio, this means setting it to 'Release' build. The final program should only take a second or so to render the image.

Starter Code

You can use the starter code provided to get the framework in place if you choose. From there, the key components required are the ray-triangle intersection routine Triangle::Intersect(), and the camera ray generation routine Camera::Render(). The lighting should be kept to a minimum.

The starter code can be found <here>.

It follows the examples in the slides from the first few lectures.

Project 1 Function

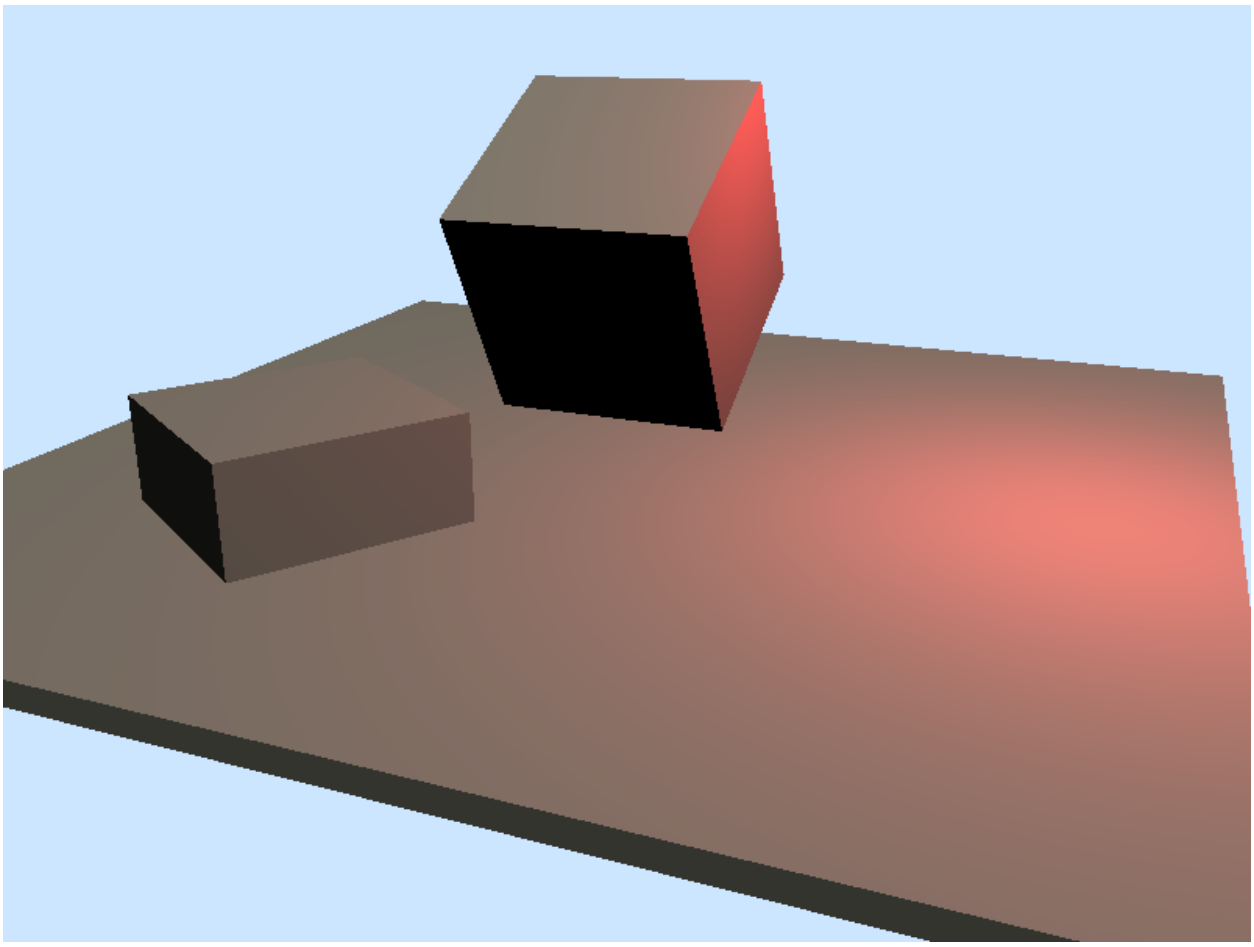
Whether or not you start with the sample code and even if you don't use C++, the code you write should be structured to be able to be run with the following sample code (or something very similar):

```
void project1() {
	// Create scene
	Scene scn;
	scn.SetSkyColor(Color(0.8f, 0.9f, 1.0f));

	// Create boxes
	MeshObject box1;
	box1.MakeBox(5.0f,0.1f,5.0f);
	scn.AddObject(box1);

	MeshObject box2;
	box2.MakeBox(1.0f,1.0f,1.0f);

	InstanceObject inst1(box2);
	Matrix34 mtx;
```

```
        mtx.MakeRotateX(0.5f);
        mtx.d.y=1.0f;
        inst1.SetMatrix(mtx);
        scn.AddObject(inst1);

        InstanceObject inst2(box2);
        mtx.MakeRotateY(1.0f);
        mtx.d.Set(-1.0f,0.0f,1.0f);
        inst2.SetMatrix(mtx);
        scn.AddObject(inst2);

        // Create lights
        DirectLight sunlgt;
        sunlgt.SetBaseColor(Color(1.0f, 1.0f, 0.9f));
        sunlgt.SetIntensity(0.5f);
        sunlgt.SetDirection(Vector3(-0.5f, -1.0f, -0.5f));
        scn.AddLight(sunlgt);

        PointLight redlgt;
        redlgt.SetBaseColor(Color(1.0f, 0.2f, 0.2f));
        redlgt.SetIntensity(2.0f);
        redlgt.SetPosition(Vector3(2.0f, 2.0f, 0.0f));
        scn.AddLight(redlgt);

        // Create camera
        Camera cam;
        cam.LookAt(Vector3(2.0f,2.0f,5.0f),Vector3(0.0f,0.0f,0.0f));
        cam.SetResolution(600,450);
        cam.SetFOV(40.0f);
        cam.SetAspect(1.33f);

        // Render image
        cam.Render(scn);
        cam.SaveBitmap("project1.bmp");
}
```

Sample Images

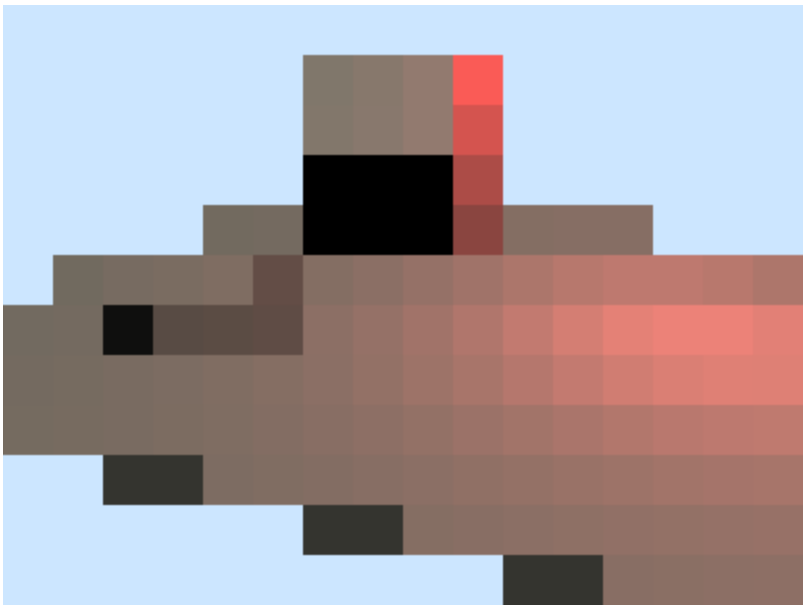
The sample code above should generate the following image exactly, pixel for pixel:



The box on the left is actually interpenetrating with the ground box, but the ray tracing should resolve this correctly because it should only accept the closest hit for each ray. If the box doesn't render correctly, then make sure your ray-triangle intersect routine is only accepting a hit if it is closer than any previous hits.

If the render resolution is changed to cam.SetResolution(16,12), the image should look like this:

Or scaled up a bit:



This can be used to verify the correctness of the camera ray generation and camera field of view. If it doesn't match, double check that you are shooting rays through the *center* of each pixel, and also make sure that you are converting the FOV from degrees to radians.

Grading

This project is worth 12 points:

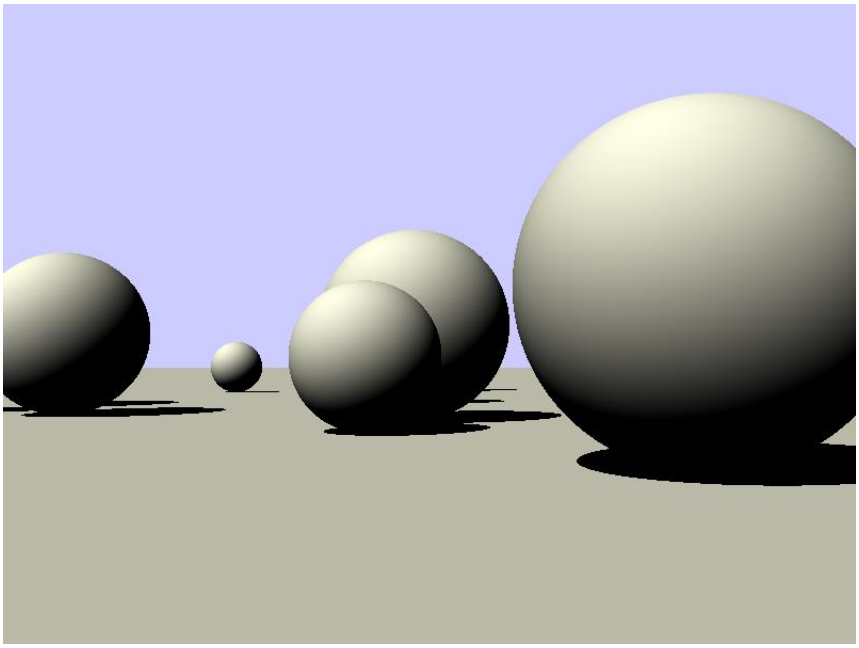
- Render anything resembling the shape of a box	2
- Render instances correctly positioned	2
- Render correct scene geometry from correct camera position	3
- Camera FOV working exactly to the pixel	3
- Render with correct lighting	2
- Total	12

Extra Credit

For 1 extra credit point total, you must do *all* of the following:

- Add support for multiple derived Object types
- Implement a SphereObject
- Implement a PlaneObject
- Add cast shadows (you'll have to do this anyway for project 2)

And then render an image something like this:



Code for the above image (NOTE- due to the random number generation, you may get different results):

```
float RangeRand(float min,float max) {
	return min+(max-min)*float(rand()) / float(RAND_MAX);
}

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

	// Create ground plane
	PlaneObject ground;
	scn.AddObject(ground);

	// Create spheres
	for(int i=0;i<20;i++) {
		SphereObject *sphere=new SphereObject;
		float rad=RangeRand(0.25f,0.5f);
		Vector3 pos(RangeRand(-5.0f,5.0f),rad,RangeRand(-5.0f,5.0f));
		sphere->SetRadius(rad);
		sphere->SetCenter(pos);
		scn.AddObject(*sphere);
	}

	// 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(-0.75f,0.25f,5.0f),Vector3(0.0f,0.5f,0.0f));
        cam.SetFOV(40.0f);
        cam.SetAspect(1.33f);
        cam.SetResolution(800,600);

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