Project 0x02: Backward Rays
Part A Due Friday, Sept. 26, 2008 at 6:00 pm
Part B Due Monday, Sept. 29, 2008 at 11:59 pm

1 Specification

Implement a physically based rendering system that is a complete solution to the rendering equation for objects that are opaque or have $\eta = 1$. Use the recursive backwards ray tracing algorithm for paths that terminate in a series of specular scattering events and a photon mapping radiance estimate for other light transport paths. Your program must provide all of the features from project 0x01, plus:

Part A:

1. Perfect mirror (pure specular reflection) BSDF
2. Transmissive BSDF
3. Ability to render the following scenes, chosen by command line argument:
   a. “specular”: Cornell box containing a mirror and translucent surfaces, approximately matching the scene shown above.
   b. One additional scene of your choice
4. Appropriate class and member documentation; no overview or results documentation is required for part A.

Transmission is required for this project. Although we’ve discussed it in class, refraction is not a required part of this project.
Part B:

1. Triangle meshes with smoothly interpolated surface normals
2. Triangle meshes with smoothly interpolated texture coordinates imported from data files
3. Spatially varying (“texture mapped”) Lambertian BSDF
4. Ability to render:
   a. “sponza”: Sponza atrium
   b. At least one additional scene of your choice

For part B, your documentation must provide overview information on a @mainpage that explains your software design and links to the relevant classes. It must also contain a “results” page or section that discusses the performance (including anything clever you did to improve it) and uses images and text to demonstrate correctness. You may need to construction additional scenes for testing or demonstrating correctness.

1.1 Restrictions

Your group submits a single solution through subversion. I encourage you to discuss the project with others outside your group and to help each other debug, derive algorithms, and understand the documentation. The code that you submit should be written solely by your group, except for trivial changes made while someone is helping you debug and code that you obtained externally in accordance with the guidelines from the syllabus. Any code not written by your group must be clearly noted in comments and documentation.

For this project, there are no restrictions on source code that you use (provided that is not written by another student working on the current assignment.) You specifically may use any code that you obtain from the internet, G3D, or books, provided the source is properly credited.

1.2 Evaluation

Your work will be evaluated for the following properties, in order of significance:
1. Satisfaction of the specification
2. Readability, including documentation
3. Mathematical correctness
4. Presentation of results
5. Design (including performance)
6. Programmatic correctness (compiling, memory management, safety)

In the event that you have known bugs, add a documentation section explaining what they are, how you attempted to debug them, and what you think the problem is. Use images as necessary. In general I will not deduct many points for bugs that you are aware of and investigated.

2 Discussion

This week you’re learning how to overcome the limits of Russian roulette sampling in the presence of a probability distribution function containing infinite peaks (a.k.a. delta functions, specularities.)

You’re learning about texture coordinates and per-vertex normals, two of the major kinds of data in 3D computer graphics, as well as how to interpolate them across a triangle. With these, you’re performing texture mapping. The process of wrapping a grid of data onto a 3D surface is one of the most fundamental in graphics. Everything that you see in a CG film or a video game today was rendered using a triangle mesh with a texture map applied to it, and now you’ll know the algorithm for that process inside and out.

On the practice side, you’re refining your BSDF design to be more general, a process called refactoring. A lot of software engineering involves refactoring based on new specifications. It is likely that changes to the BSDF will propagate backwards into the rest of your code. If you were starting from scratch again, think about how you would design your major classes to insulate them from changes to the interface of BSDF as much as possible.

This is the last week that you’ll work with your ray tracing infrastructure. The ray tracer in some sense represents the ideal rendering algorithm. It closely follows actual physics and uses statistical methods to evaluate the rendering equation efficiently. To create animations, you need to render many frames. To make animations that interact with the operator, you need to render those frames in less than 1/12 of a second. Most games go farther and render about 60 frames per second. In future weeks we’ll accelerate our evaluation until it runs that fast. We’ve reached the limit of what we can do by sampling more cleverly—now we’re going to have to start approximating the rendering equation, and in doing so will introduce some error. The reason that we spent three weeks ray tracing is so that we can introduce that error intelligently and be able to accurately measure how we are biasing the solution. So, although you’ll be able to render images much faster in a few weeks, they aren’t going to look as realistic.

Although we’ve implemented a complete solution to the rendering equation, there are still lots more features one could add to the ray tracer to make it easier to model interesting scenes, and a few tricks for tracing specific kinds of scenes much faster. These include a lens (focus), refraction, extinction, direct illumination, implicit surfaces (blobby things), participating media (e.g., fog), procedural textures, bump maps, and lots of different kinds of BSDFs and emission functions. For your final project, one good option is picking up your ray tracing code and exploring these features.
3 Advice

This project involves a lot less programming than previous weeks (hooray!) If your existing infrastructure was designed to be sufficiently general, you may only have to write two new BSDF subclasses, make a minor change to your triangle mesh class, and add two recursive calls from your primary backtracer method. If your code was not designed for this, then you may need to spend some time generalizing your existing code base before moving forward. If that is the case, recall that you are allowed to use any student’s code from last week, so you can ask others outside your group if they will give you pieces of their project.

G3D::Vector3 has reflect and refract methods. Beware that these might not follow the exact conventions that you used for directions, so you might need to negate the input or output vectors.

3.1 Interpolation

This week, each triangle or rectangle vertex has three properties: a position, a normal, and a texture coordinate. How can each vertex have its own normal? These aren’t real normals. Instead, we’re going to use a lot of triangles to approximate a smooth surface. To hide the faceting that occurs from using individually flat surfaces, we’ll lie to the ray tracer and tell it that the normal (for shading purposes) is a blend of the vertex normals for points inside the triangle. When computing the geometric intersection, however, you need to keep using the true surface normal or you’ll miss the intersection.

Texture coordinates are coordinates (typically on the unit square) that indicate how an image is supposed to be wrapped across the surface of a primitive in order to make it look like there is more geometric detail on the surface than is really present. By convention (0,0) is the upper left corner of the image and (1,1) is the lower right. Also by convention, the texture is assumed to “wrap” at the sides, so (-0.2, 0.3) is the same point as (0.8, 0.3).

ArticulatedModel provides texture coordinates and vertex normals for some IFS files and for most 3DS files (like sponza.3DS). These are in arrays that are parallel to (use the same indices as) the geometry.vertexArray that you extracted for your mesh class. The texture coordinates are specified in an absolute reference frame, so you can use them unmodified. The normals are relative to the coordinate frame of each part. To use them you need to call CoordinateFrame::normalToWorldSpace using the same CoordinateFrame that you used to transform vertices to world space (don’t worry, soon, we’ll learn what a CoordinateFrame really is and what this “world space” reference frame is! For now, just call the methods.)

You can specify texture coordinates and normals manually for a rectangle.

Your triangle and rectangle intersection code probably already generates (u, v) values between 0 and 1, which are two of the three Barycentric coordinates (see RTR 16.8.1). The Barycentric coordinates are the weights that you need to scale the vertex normals and texture coordinates by to obtain the normal and texture coordinate at a point within the triangle or rectangle. Note that you will need to rescale the normal to unit length after interpolation.

I recommend storing the texture coordinate as a Vector3 with z=0 instead of as a Vector2. The reason for this is that you might later want to add solid texturing or cube-map texturing, which need 3D vectors.
RTR 6 describes some methods for assigning texture coordinates to surfaces (like the sphere) that do not have explicit vertices, such as spherical or cylindrical projection. You are not required to implement any of these—e.g., spheres can simply return (0, 0) as their texture coordinate, always, and meshes either come in with texture coordinates when imported from a file or default to (0, 0) as well.

One way to debug texture coordinates is to turn off photon mapping and have the backward tracer use the texture coordinate \((x,y,z)\) value as the \((r,g,b)\) value for the radiance estimate. This will show you a picture of the texture coordinates. The amount of red and green on the screen at each point is a visualization of the coordinate, as shown on the right. Note that there will be some values outside the \([0, 1]\) range because of wrapping, which will be too bright or completely black.

You can show the texture coordinates for debugging in two different way. Both are valuable. Since you now have real-time preview code, set the RenderDevice shade mode to shade smooth (like you did for the gradient), and try drawing the texture coordinates as colors at the vertices in preview mode. Then, when you’re convinced that they are right, make sure that your back tracer can generate the same image as preview mode. Don’t forget to turn off this debugging visualization before submitting your work (you may want to leave it commented out or as a runtime option tied to a checkbox for later debugging).

This notion of drawing something on the screen to debug is a common practice in computer graphics, and is often the only practical way of understanding how thousands of values are changing.

You can debug your vertex normals by the same method, or by explicitly drawing them in preview mode. The following code will iterate through a BSP tree of triangles and generate two arrays, one of vertices and one of normals, and then draw them. This isn’t particularly fast code, but is is for debugging so that doesn’t matter. Here, “it” is the iterator. It acts like a pointer to one triangle. The code below assumes that the triangle has members \(v_0, v_1, v_2, n_0, n_1, n_2\). My implementation actually didn’t; I was storing \(v_0, \) edge1, and edge2. But I changed it to look like this for demonstration purposes. See the G3D manual for more details about Draw and AABSPTree.
I rendered the picture on the right in preview mode using this code. Note that the sphere is not a triangle mesh, so the triangle mesh code can’t draw its normals!

Note that in the picture on the right the vertex normals for the icosahedron are pointing in all different directions, like a little bush at each vertex. If you just load the icosahedron model directly with ArticulatedModel, it will have a single “smooth” vertex normal, attempting to draw an icosahedron that looks like a sphere. In order to turn the icosahedron (and the box that it is sitting on) back into faced polyhedra, I called ArticulatedModel::facet after loading and before turning it into a mesh entity. For sponza or a smooth shape, I do not call facet() because I want them to look smooth.

3.2 Texture Mapping

Make TexturedLambertianBSDF a different class than your existing LambertianBSDF; both will be useful to you.

In order to avoid allocating tons of memory (and running slowly as a result), memoize your BSDFs when creating them. G3D contains all of the necessary data structures: just make a Table<Material, BSDFRef> and check to see if the material that you are trying to create is already in that table before allocating a new one. Don’t forget to put any newly allocated BSDFs into the table as well! The basic algorithm for this is:
The TexturedLambertianBSDF should be constructible from a reflectivity (color) and an Image3Ref. You can get the reflectivity from Material::diffuse.constant and the Image3 from Material::diffuse.map. The latter is a TextureRef; call Texture::toImage3(true) to get the Image3. The Image3 is a 2D array of reflectivities that map over the surface. The actual reflectivity that you return from your BSDF should be the product of the constant, the map reflectivities, and 0.99 (to ensure that no surface is perfectly reflective). You can either precompute the product at each location by modifying the Image3 on construction or simply perform that product every time a method is invoked. When you’re memoizing the TexturedLambertianBSDF (in its own table, separate from other BSDFs), you have two keys that you need to memoize on. The easiest way to deal with this is to simply use the triList’s Material from which you extracted the color and image in the first place. Table already has a hash function for Material, so you can just use Table<Material, BSDFRef> and get the right result.

You can read about texture mapping in RTR 6.2 and 6.3, and it will be discussed in lecture. Note that you are not required to implement anything other than bilinear magnification (i.e. Image3::bilinear(x,y))

Image3 uses integer coordinates between (0,0) and (w-1, h-1) to address pixels (fractional indices indicate a blend of adjacent pixels). You need to multiply the texture coordinates by the width and height of the image in order to scale them up correctly before calling Image3 methods. Set the Image3 wrap mode to WrapMode::TILE to get the appropriate tiling behavior.

As with texture coordinates, a good way to debug texture mapping is to turn off the radiance estimate and just draw the BSDF::evaluate value on screen.

### 3.3 Recursive Ray Tracing

Conceptually, the back tracer has two methods: estimateLo and estimateLi. EstimateLi just calls estimateLo with the other direction at the intersection of the ray and the surface (review the Rendering Notes if this idea isn’t immediately familiar).

So far, we’ve been estimating the scattered radiance at a point using the photon map. This works well for BSDFs that are diffuse, meaning that they don’t have any really sharp peaks. If the BSDF has a sharp peak it is called specular. The problem with sharp peaks is that we’re really unlikely to have a photon that happens to come in at exactly the direction that matches the sharp peak. In
other words, our current algorithm doesn’t sample sharp peaks well. If we shot trillions of photons (like nature does!), photon mapping would get the right result. But for the low numbers of photons that we’re dealing with, we need a trick to correctly sample these specular peaks.

To handle specular surfaces, your backwards tracer now needs to compute estimateLo as:

\[
L_o = L_e + L_s \quad \text{← what we’ve already got}
\]
\[
L_s = L_d + L_m + L_t \quad \text{← new definition of “scattered light”}
\]

Where:

- \( L_o \) = outgoing radiance
- \( L_s \) = scattered radiance
- \( L_d \) = diffuse scattered radiance (estimated from the PhotonMap)
- \( L_m \) = mirror reflected radiance (recursive call to estimateLi with the mirror reflection direction)
- \( L_t \) = transmitted radiance (recursive call to estimateLi with the refracted direction)

Since not all surfaces need the diffuse, mirror, or transmissive term, you should extend your BSDF class so that the back tracer can ask a BSDF which of the three kinds of rays it will return non-zero values for. E.g., for a Lambertian surface, we don’t want to bother computing specular samples, since a Lambertian BSDF is not specular.

Rename your original BSDF “scatter” method to something like scatterOrAbsorb, to better represent its true nature. Then, add scatterMirror and scatterTransmit. Scatter mirror scatters the photon in the mirror reflection direction and returns the mirror reflectance probability. That is, it will always scatter the photon—it is not using Russian roulette. Scatter transmit always scatters the photon in the transmission direction (which, in the absence of refraction, is the same as the incoming direction).

Now write new BSDF subclasses that implement these methods. By the end, you should have LambertianBSDF, TexturedLambertianBSDF, MirrorBSDF, and TransmitBSDF. Note that the original scatter method needs to support mirror and transmissive scattering for the appropriate subclasses. MirrorBSDF::scatterOrAbsorb can call scatterMirror and TransmitBSDF::scatterOrAbsorb can call scatterTransmit so that you don’t implement the same scattering code twice in each class. The catch is that we still have to implement the “OrAbsorb” part. So MirrorBSDF::scatterOrAbsorb should look something like:

```c
if (rand <= rho_mirror) {
    get direction from scatterMirror
} else {
    absorb
}
```

Note that you’re now scattering during backward tracing. Before we only scattered during forward tracing.

Real transmissive surfaces always both reflect and transmit (and almost all have at least a little diffuse scattering as well!) You are not required to implement a BSDF that has that property, although doing so as an extension on the specification or in a final project is not very difficult—just make a new BSDF subclass that implements both scatterMirror and scatterTransmit. Here the scatterOrAbsorb method has several cases:
if (rand <= rho_diffuse) {
    diffuse reflect
} else if (rand <= rho_diffuse + rho_mirror) {
    mirror reflect
} else if (rand <= rho_diffuse + rho_mirror + rho_transmit) {
    transmit
} else {
    absorb
}

For the specular scene, I positioned the rectangular mirror using the following code. You may need to adjust the vertices based on the scale of your box. The fromXYZYPRDegrees method chooses a coordinate frame based on the (x,y,z) position and yaw, pitch, and roll angle measures.

```
CFrame cframe = CFrame::fromXYZYPRDegrees(-1.4,-0.55f,-0.7,45,-8,0);
Vector3 v0 = cframe.pointToWorldSpace(Vector3(-1.2f, 1.5f, 0));
Vector3 v1 = cframe.pointToWorldSpace(Vector3(-1.2f, -1.5f, 0));
Vector3 v2 = cframe.pointToWorldSpace(Vector3(1.2f, -1.5f, 0));
Vector3 v3 = cframe.pointToWorldSpace(Vector3(1.2f, 1.5f, 0));
s->insert(RectangleEntity::create(v0, v1, v2, v3, mirror));
```

My back tracer estimateLo looks something like the code below. Note that my code contains extra support for refraction with the eta variables that you don’t need, and that you should use my code more as a reference than as starter code to copy—you’ll end up spending a lot of time debugging the difference in our conventions if you use it directly. See also the Rendering Notes description.
Radiance3 BackTracer::estimateRadiance(
    const Ray& worldRay,
    float refractiveIndex,
    int backBouncesAllowed,
    Vector3& normal,
    float& depth) {
    Hit hit;
    float distance = inf();
    m_scene->intersect(worldRay, distance, hit);
    depth = distance;
    normal = hit.normal;

    static const Radiance3 ZERO(Color3::black());

    // Paths of the form L[S|D]*DE
    Radiance3 diffuseScattered(ZERO);

    // Paths of the form LE
    Radiance3 emitted(ZERO);

    // Paths of the form L[S|D]S+E
    Radiance3 reflectedSpecular(ZERO);
    Radiance3 transmittedSpecular(ZERO);

    if (distance < inf()) {
        // Outgoing light direction
        Vector3 w_o = -worldRay.direction;

        // We hit some object. Estimate the radiance that
        // was reflected from that object
        // back along this ray.
        if (! hit.bsdf->allDeltas()) {
            diffuseScattered = photonMapRadianceEstimate(hit.bsdf,
                hit.position, w_o, hit.texCoord, n, refractiveIndex);
        }

        if (hit.emissionFunction.notNull()) {
            // We hit an emitter
            emitted = emitterRadianceEstimate(hit.emissionFunction,
                w_o, hit.texCoord, hit.normal);
        }

        if (hit.bsdf->hasDeltas() && (backBouncesAllowed > 0)) {
            // Recursive bounce
            Ray recursiveRay;
            Color3 reflectivity;

            // Reflection (returns 0 for reflectivity if absorbed)
            hit.bsdf->scatterMirror (hit.normal,
                eta_i, w_o, hit.texCoord,
                eta_o, recursiveRay.direction,
                reflectivity);

            if (reflectivity.sum() > 0.0f) {
                recursiveRay.origin = hit.position +
                    hit.normal * 0.00001f;
                reflectedSpecular = estimateRadiance(recursiveRay,
                    refractiveIndex, backBouncesAllowed - 1) *
                    reflectivity;
            }

            // Refraction
            hit.bsdf->scatterTransmit(hit.normal,
                eta_i, eta_o, w_o, hit.texCoord,
                refractiveIndex,recursiveRay.direction,
                reflectivity);

            if (reflectivity.sum() > 0.0f) {
            }
recursiveRay.origin = hit.position - hit.normal * 0.00001f;
transmittedSpecular = estimateRadiance(recursiveRay, refractiveIndex, backBouncesAllowed - 1) * reflectivity;
}

} else {
    // Hit background
    emitted = BACKGROUND_COLOR;
    normal = Vector3::unitZ();
}

return diffuseScattered + emitted + transmittedSpecular + reflectedSpecular;

---

**Project 0x02 Groups**

**theta**
- 09twb
- 09kaw_2

**iota**
- 09cmz
- 10dpf

**kappa**
- 10msl
- 09ack_2
- 09sb

**lambda**
- 09msg
- 09wkj

**mu**
- 09twb
- 09ajs

**xi**
- 10kl
- 09jmc

**09pzh**
- 09pzh

svn co svn://graphics-svn.cs.williams.edu/371/2/ray1-GROUP

Your username and password are the same as last week.