CS 314: More Acceleration, Starting Shading

Robert Bridson

October 16, 2008

1 Further Exploiting BVHs

Last time we figured out how to build a reasonably good BVH based on axis-aligned bounding boxes, at least for most inputs if not all. We can use it to accelerate casting a ray and looking for intersections: a ray should check the root node, and if it intersects, recursively check all the children or if it's at a leaf check the actual geometry stored there, keeping track of the nearest intersection found so far. (And remember that usually the recursion is handled explicitly with a stack, not function calls.)

We can do a bit better however. Since we're looking for the *first* intersection, as soon as we've found any intersection with actual geometry, we can skip over any bounding box that the ray hits after that parameter value (recall also that our primitive ray vs. box test actually calculated the interval of parameter values where the ray goes through the box). This could be an enormous savings: if a million triangles are inside a bounding box that we skip because it's "behind" the current nearest intersection, that's a million-fold speed-up on that branch of the BVH.

In fact, this is another of the reasons that some experts think raycasting may yet end up used more in real-time graphics—the Z-buffer algorithm doesn't have the same possibility for speed-up. Define the **depth complexity** to be the total number of intersections between a ray and the geometry in the scene: call it d. In the ideal case, if the BVH performs as well as we would like it to, raycasting can find the first intersection in $O(\log d)$ time. For Z-buffer, however, we have to rasterize every single triangle in view, so we'll take time $O(d)$ to get that same first intersection.

It's not yet clear if depth complexity really is scaling up so fast in real-world applications that we'll hit a point in the future where raycasting is actually faster (since it's harder to implement in hardware than Z-buffer), but some people are convinced. There are also many special cases where we can apply tricks to get similar speed-ups in Z-buffer, so it's not quite as black-and-white as I've made it out to be:

for example, if the camera is in a closed room on a massive level in a game, only the geometry of the room has to be rasterized since it will be impossible to see any of the rest of the level. This is sometimes included in the category of **occlusion culling**: techniques for efficiently "culling" or skipping geometry that we can guarantee won't show up in the image because it's blocked ("occluded") by other geometry. One special case worth mentioning is **portal culling**, which is designed for indoor scenes: figuring out that if the camera is in room A, even if the windows or doors are open, there's no sight line that would allow it to see into room B (which is maybe around a bend in the corridor, for example), and therefore room B needn't be rasterized.

A further tweak to the raycasting acceleration that we can apply is to test the children of a BVH node in order of their first intersection with the ray. This makes it more likely we'll hit the first (or at least a relatively near) intersection early on, and can then cull out more nodes in the BVH afterwards.

Finally, the BVH also has an important use in Z-buffer rendering too. Any triangle that ends up outside the view frustum or view volume is pointless to send for rasterization. To avoid this unnecessary expense, we can traverse the BVH as well, skipping over any node whose bounding box doesn't intersect the view frustum (and thus possibly skipping lots of geometry). This is called **view frustum culling**, and is almost always a very good idea to implement.

2 Shading Diffuse Surfaces

We now have worked out an awful lot of details on rendering, from rasterization to transformations to efficient raycasting, but we haven't yet talked about what color to make pixels that are successfully rasterized or have a ray which intersects the geometry. Earlier on we implicitly assumed the user would have specified an RGB colour for us—perhaps interpolated from the vertices using barycentric coordinates—but now we'll properly tackle the question. This is the subject of **shading**: figuring out what colour to return along a ray or what colour a pixel rasterizes to.

This is a big subject, that can overlap with both the physics of light transport (how does light interact with a surface) and the artistry of defining a look that communicates to the viewer what the artist wants. We'll stick with the more physical side of it for now.

We see colour on a surface because light is emanating from it. Where does that light come from? It either could be emitted directly from the surface (if the surface is glowing) or could be transferred from light arriving from elsewhere that illuminates the surface. Things get a little complicated because in the second case, the incoming light might be shining directly from a light source or it could be from another surface in the scene.

For a physical look at shading, we generally talk in terms of **materials**, where specific materials include anodized aluminum, house paint, polished oak wood, glass, etc. An object made of any of these will interact with light at its surface in particular ways, which we need to model on the computer.

The simplest and most common material model of is called the **Lambertian** model (or **diffuse** shading, or **matte** material) after the mathematician Lambert. Most non-shiny materials are reasonably well approximated as being Lambertian, though it is an idealization that doesn't really exist in the world. The idea is that an incoming light ray should be scattered equally in all directions: when a photon hits a Lambertian surface, it bounces off in a random direction chosen uniformly from the entire hemisphere of possible directions off the surface.[1] Contrast this to a perfect mirror, which we'll cover next time: in a mirror, an incoming light ray bounces off only along a single outgoing direction, the mirror reflection of the ray.

The Lambertian model is particularly simple because the outgoing light (the colour of the surface) doesn't depend on what direction you're looking at it from—all outgoing light directions are equal—but it's not entirely trivial because it *does* depend on the incoming light. Our first step is measuring how much light is illuminating the surface.

This isn't simple. Our question "how much light" is imprecise: how do we measure the amount of light on surface? There is a whole range of scientific measures of light, defined in the fields of radiometry and photometry; we'll sidestep these technical units and hopefully stick to just intuitive mental pictures.

The first good thing to think about is to visualize light energy as coming in photons: the total amount of light illuminating a surface could be thought of as the number of photons which hit it. (Obviously we should probably specify per unit of time, making this a measure of *power* rather than energy.) But that isn't quite what we want here: if a surface receives 1000 photons in one second but is the size of the moon, we'd expect to see any point on the surface as much darker than if the surface was the size of a sesame seed. So when we talk about the amount of light hitting a point on the surface, we really should be thinking of measuring it as the number of photons per unit time and **per unit area**. It's really the *density* of photons hitting the surface that counts.

Now, suppose at the point on the surface we're interested in, light rays (i.e. photon paths) are coming in parallel, opposite a unit-length direction $\vec{l}$ which points towards the light source. Draw a side view diagram, along a plane spanned by $\vec{l}$ and the unit-length surface normal $\hat{n}$. Suppose the photon paths are separated by a distance l: then $1/r$ is a measure of the density (photons per unit area) ignoring the spacing of photon paths in the third direction, which will be constant while we work through this.

[1] Note the word hemisphere: all directions from a point in empty space would cover a sphere, but for a point on an opaque surface only the directions that don't go into the object are allowed, which is half a sphere: a hemisphere.

If the surface is being struck by the photons head-on, i.e. $\hat{n} = \vec{l}$, then the density of photons on the surface will be exactly the same as the density of incoming light: photons will strike at a distance l apart in this side view. On the other hand, if the light direction is parallel to the surface (i.e. orthogonal to the normal) then no photons will hit the surface at all. At directions in between, the density of photons will vary.

We can work out exactly how by using simple trigonometric argument: draw the right triangle where one leg is the distance r between two photon paths, another leg is along a photon path, and the hypotenuse lies on the surface. The length of the hypotenuse gives us the distance that photons are separated when they hit the surface. If θ is the angle between the light direction and the surface then the length of the hypotenuse is:

$$\frac{r}{\sin\theta}$$

Therefore the light density on the surface is proportional to $1/(r/\sin\theta)$, in other words $\sin\theta$ times the incoming light intensity. With a little more trigonometry, we can simplify that to the cosine of the angle between the light direction and the surface normal, which is given from the dot-product

$$\vec{l} \cdot \hat{n}$$

We multiply this by both the incoming light intensity and the surface material's diffuse property (how much light it reflects back uniformly, as opposed to just absorbing it and getting hotter) to get the final shaded value:

$$CL(\vec{l} \cdot \hat{n})$$

Here C is the surface's material property, L is the incoming light intensity. We can extend this to be calculated for each colour component individually, making C and L RGB vectors and doing the multiply element-by-element.