A.A. For each pixel in your screen or window, you have to do a mathematical calculation called "casting a ray into the scene". When the ray hits an object in the scene, you might have to cast multiple recursive rays, depending on how many lights there are, whether the object is reflective, whether it's translucent, etc. That is, your first ray might branch into b rays. And each of those recursive rays might branch into b recursive rays. And so on. In principle, there is no end to the recursion. In practice, we cut it off at some depth d. So there's something like $\mathcal{O}(b^d)$ work being done. That's expensive. And we haven't even talked about how the time complexity depends on the number p of pixels (linearly) or the number n of scene objects or triangles (linearly at each recursion). The algorithm can be parallelized — especially over the pixels in the screen — but this parallelization is not widely available in practice, and it does nothing to reduce the complexity in theory.

[To go beyond what I expect of a CS 311 student: Let $T(d)$ be the number of ray-body intersections needed to color a single pixel of the screen at recursion depth cutoff d. Suppose that we use our usual shading model with ℓ shadow rays (one for each light), one mirror ray, and one transmission ray. Then $T(d)$ can be expressed using a recurrence relation

$$\begin{aligned} T(d) &= n + \ell n + (n + T(d-1)) + (n + T(d-1)) \\ &= (\ell + 3)n + 2T(d-1) \end{aligned}$$

with initial condition $T(0) = (\ell + 1)n$. If we ignore the $(\ell + 3)n$ term, then the solution is $T(d) = (\ell + 1)n2^d$, so the overall time complexity is $\mathcal{O}(p\ell n2^d)$. Incorporating the $(\ell + 3)n$ term makes the running time even greater, although it's still $\mathcal{O}(p\ell n2^d)$ I think. Try to fill in the details.]

A.B. The triangle rasterization algorithm makes a Crucial Assumption: The scene can be broken into triangles that are independent of each other. That is, when you're rendering one triangle, you don't need to know anything about what's happening with other triangles. This assumption is not realistic; for example, it requires us to handle shadows using an expensive add-on called "shadow mapping". If p is the number of pixels, n the number of scene objects or triangles, and ℓ the number of lights, then the time complexity is something like $\mathcal{O}(pn\ell)$ without shadow mapping or $\mathcal{O}(pn\ell^2)$ with shadow mapping on all lights. In any event, there is no exponential complexity as in ray tracing. Also, massively parallel hardware for triangle rasterization is mature and widely available, which decreases the practical running times, even if it has no effect on the theoretical complexity.

[In practice, not many lights are shadow-mapped, to keep the complexity at $\mathcal{O}(pn\ell)$. For example, on the last day of class I showed a scene from *Deus Ex: Human Revolution* using more than 40 lights, only two of which were shadow-mapped. Notice that this $\mathcal{O}(pn\ell)$ matches the ray-tracing complexity $\mathcal{O}(p\ell n2^d)$ when $d = 0$. In a sense, ray tracing lets us escape the confines

of the Crucial Assumption, at a cost that is exponential in how far we want to go beyond those confines.]

B.A. [Many students responded to the prompt by offering other algorithms, that did not follow the requested structure. I am not asking about algorithms for box intersection; I am asking about *this* algorithm for box intersection.]

We begin by transforming the ray data $\vec{e}$, $\vec{d}$ into local coordinates of the box.

For the $X =$ `left` plane, compute the intersection time t for $\vec{x}(t) = \vec{e} + t\vec{d}$. (This t exists because we are ignoring the special case. It might not be in $[t_{\text{start}}, t_{\text{end}}]$. By the way, we're solving $e_0 + td_0 = \ell$, so we get $t = (\ell - e_0)/d_0$, but I'm not grading you for algebra.) Based on d_0, determine whether that intersection is entering or exiting. If it's entering, then set t_{start} to $\max(t_{\text{start}}, t)$. If it's exiting, then set t_{end} to $\min(t_{\text{end}}, t)$. After this operation, $[t_{\text{start}}, t_{\text{end}}]$ is the time interval that the ray spends in the half space whose boundary is the $X =$ `left` plane. If $t_{\text{start}} > t_{\text{end}}$, then the interval is empty, so return `rayNONE`.

Similarly, for each of the other planes, compute the intersection time t, use the local $\vec{d}$ to decide entering vs. exiting, and update t_{start} and t_{end}. Return `rayNONE` if the interval is empty.

At the end of this process, $[t_{\text{start}}, t_{\text{end}}]$ is the non-empty time interval that the ray spends inside the box. There are two cases. If t_{start} is still `rayEPSILON`, then the camera is inside the box, so return `rayEXIT` with t_{end}. Otherwise, return `rayENTER` with t_{start}.

B.B. The basic idea is simple: Remember which side of the box was hit, so that you know its local normal. Return that local normal transformed to global coordinates.

Here's how we can incorporate the idea into our algorithm above. Maintain two more variables, $\vec{n}_{\text{start}}$ and $\vec{n}_{\text{end}}$. Whenever you update t_{start} or t_{end}, also update $\vec{n}_{\text{start}}$ or $\vec{n}_{\text{end}}$ respectively. For example, if your first intersection, with the $X =$ `left` plane, causes t_{end} to be updated, then at the same time set $\vec{n}_{\text{end}}$ to $(-1, 0, 0)$. At the end of the algorithm, the local normal is $\vec{n}_{\text{start}}$ if you're returning t_{start} or $\vec{n}_{\text{end}}$ if you're returning t_{end}. Before you return the normal, transform it from local to global coordinates.

C. Assume for a moment that all translucent bodies have index of refraction 1. Then we could modify our usual shadowing algorithm by casting the shadow ray toward the light source recursively, tracing its path through translucent bodies. If the shadow ray hits an opaque object, then the fragment is in shadow as usual. If it hits only translucent objects, then some light reaches the fragment. The amount of light is the light's original color modulated by $(\vec{c}_{\text{trans}})^{\ell}$ for each transmission, as we did for translucency. We could even let our recursive shadow rays branch to handle mirroring. This whole process would be expensive but not much harder than what we currently have. [The time complexity would increase from something like $\mathcal{O}(2^d)$ to something like $\mathcal{O}((\ell + 2)^d)$. Also, one student wisely points out that textured light, which we

implemented in rasterization, is a special case of this problem.]

The problem is dramatically harder if we account for indices of refraction other than 1. When we cast a shadow ray from a fragment toward a light source, we do so because we assume that the light (if any) will take a straight path from the source to the fragment. The presence of translucent bodies means that light can take a bent path instead. Which bent path? It's hard to know. There is only one straight path, but there are infinitely many bent paths. So in which direction do we even cast our shadow ray?

I think that we would have to use something like distribution ray tracing, which was discussed briefly in class. Send out many rays from the fragment. For each one, do the recursive ray tracing outlined above. So we're adding the expense of recursive ray tracing, but more importantly we're adding the expense of distribution ray tracing. On the plus side, once we're doing distribution ray tracing, we can probably improve some of our other lighting effects.