# Lightning: A System for Reusing Ray Trace Data for Fast Lighting of 3D Scenes

Jennifer K. Stewart

Department of Computer Science
Brown University
Providence, Rhode Island 02912

**CS-98-09**
August 1998

# Lightning:
# A System for Reusing Ray Trace Data for Fast Lighting of 3D Scenes

Jennifer K. Stewart

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science
in the Department of Computer Science at Brown University

August, 1998

Professor John Forbes Hughes
Advisor

# Table of Contents:

## Abstract

The most time-consuming part of lighting 3D scenes in computer graphics is the rendering and re-rendering artists have to do as they edit the lights and their parameters. Most lighting editors have some features to expedite this process, such as low resolution or fast shading render previews, but these typically fail to generate exactly the same shadows, highlights, and reflections that appear when the scene is rendered normally. We present a method for ray tracing 3D scenes where these characteristics are rendered just as accurately as with traditional ray tracing, but in up to 98% less time. By storing information from an initial, expensive, ray trace of the scene, we can re-render subsequent changes to the lights in a small fraction of the time it takes to do a regular ray trace, and far more accurately than using a low resolution or fast shading render preview.

# 1 Introduction

## 1.1 Motivation

Effectively lighting 3D scenes in computer graphics is essential for achieving realism, depth, mood, and atmosphere. Because we are working with digital instead of real-world objects, we must take extra care to light objects in such a way as to make them look real instead of fake, 3D instead of flat, and tangible instead of digital. For example, compare the two images below. It is quite obvious that the scene on the left is computer generated; however, for the scene on the right it is not so clear. The *only* difference between these two scenes is the lighting model.

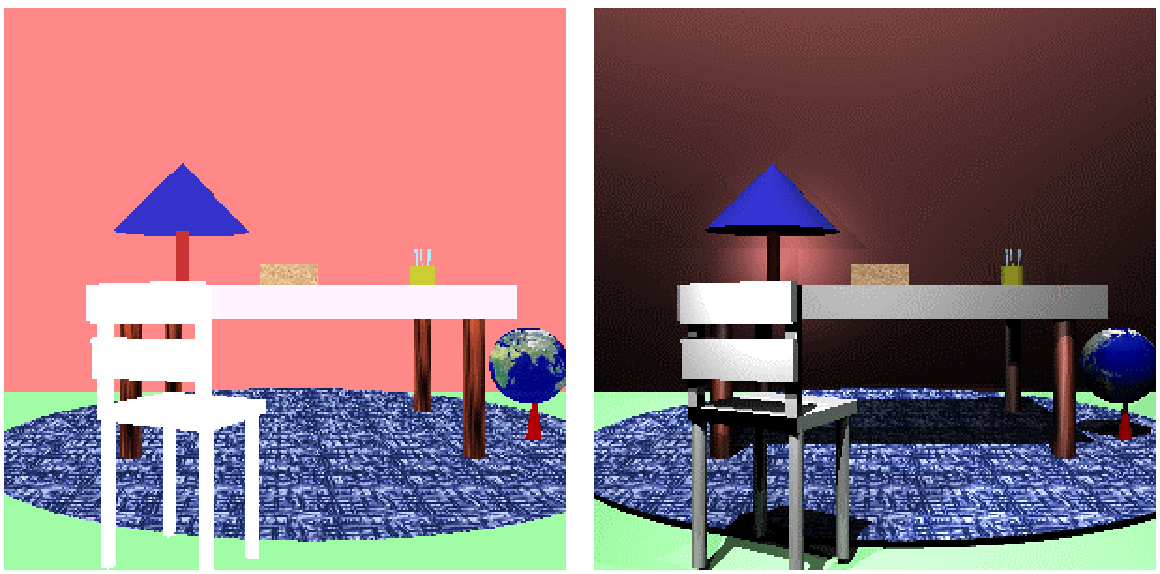


*Figure 1: We rendered the scene on the right with our system, Lightning, and the scene on the left with a less complex lighting equation than ours uses.*

Animators and artists can spend days lighting a single shot to achieve believable effects, often using as many as twenty or thirty lights. Each of these lights needs to be adjusted for the proper position, direction, color, intensity, attenuation, angle, dropoff, or other characteristics[1]. Rendering and re-rendering while tweaking these parameters can be extremely time consuming. Typically, an artist generates low resolution or fast shading render previews to quickly see how his or her lighting edits are affecting the scene. However, the shadows, highlights, and reflections these produce are often very different from those a normal render would generate, or worse, are not present at all. Consequently, the artist may discover unexpected or undesirable results only later when

[1] Attenuation refers to how fast a light's influence dissipates as objects get farther away from it. Angle refers to how wide a spotlight's cone-shaped range of influence is. The dropoff rate refers to how fast a spot light's intensity falls off from the center of its area of influence. Each of these is explained in more detail, with examples, in section 4.

he or she decides to wait for a complete render. Our goal was to create a lighting tool that would allow an artist to re-render a scene multiple times as s/he makes light edits, each time generating *exactly* the same results that a traditional ray trace would, but in a small fraction of the time. Because an artist (also called lighter, animator, or technical director) typically does not edit camera or object positions, we can take advantage of the fact that only the lights are changing. We do this by first storing information from an initial, expensive, ray trace of the scene, where each pixel's color is determined from a complex lighting equation. Then, after an artist changes some lighting parameters, we re-render the image by recomputing only a portion of that equation for each pixel, depending on which light the user edited and what type of changes he or she made. We call our light editor "Lightning."

### 1.2 Related Work

A number of researchers have developed ways to speed up ray tracing. Some well-known techniques include using bounding volume hierarchies [1] or octrees [2] to expedite intersection testing, or using parallel processing to render multiple pixels simultaneously [3]. Each of these schemes could be incorporated into our system, but is orthogonal to our overall goal of improving the rendering times for re-lighted models. Furthermore, we were interested in implementing an automatic system for improving rendering times, whereas bounding volume hierarchies and octrees often require user tweaking to operate effectively.

More directly related is work that researchers have done to speed up ray tracing after a user has made edits to the scene. Brière and Poulin provide a method for updating only the portion of a ray-traced image affected by changes rather than re-rendering the entire image [4]. While effective, their approach requires the maintenance of complicated data structures and is more appropriate for scenes where not only lights, but also objects and surface properties are being changed. Furthermore, if a light changes, it often affects almost everything in the scene (at least to some degree) so storing which lights affect which regions is not necessarily worth the expense. In addition, if a light's position changes, then many additional regions may now be affected anyway. Séquin and Smyrl present an impressive method for re-rendering a scene quickly after surface characteristics and light attributes have changed [5]. Using very little memory, they store pixel data so that they can perform only minimal recalculations in order to re-render. The drawbacks to this technique are that it does not handle changes to light position, direction, or type, nor does it permit adding or deleting lights.

### 1.3 Document Overview

This document contains four distinct parts. First, we explain traditional ray tracing and how our method is an improvement over it in a light-editing environment. Next, we describe the user interface to the editor including how to change lights and their parameters. Third, we explain the implementation details and results. Specifically, we describe how Lightning quickly re-renders a scene using different methods depending on what type of light changes an artist made. We also compare the speed of rendering these changes with that of traditional, naïve ray tracing. Finally, we discuss conclusions and future work. Appendix A shows a block-diagram description of the code, and Appendix B shows additional results and scenes lit with our system.

# 2 Project Description

This program is based on traditional ray tracing, which includes the ability to render images with shadows, reflections, and diffuse and specular lighting. First, we review how traditional ray tracing works. Then we discuss how our implementation is different and why it is useful.

## 2.1 Traditional Ray Tracing

Ray tracing is a photorealistic rendering method pioneered by Appel [6] and used for years to create pictures of 3D scenes. The algorithm generates shadows and specular highlights quite realistically, and when run recursively also generates reflections. The basic algorithm for ray tracing is this: for every pixel in the image, shoot a ray from the eye point through the pixel and into the virtual 3D world. Test the ray for intersection with every object in the scene. The object to be drawn at this pixel is the one that is intersected and is closest to the viewer. If this object is not reflective, color the pixel based on this object's color and the lights that hit this point on the object (see the next section for the lighting equation). To determine which lights hit this point on the object, shoot a ray from this point to each light in the scene. If a light is not blocked by another object, then it is visible and contributes to the color; otherwise the object is in shadow from this light. The final color of the pixel is a based on the diffuse and specular contributions of all the point's visible lights combined with the color of the object itself. If no object was intersected, color the pixel the background color (usually black).

If the object is reflective, we still calculate the diffuse and specular contributions of each light to this object's color as described above, but we also must account for the contribution of the object it is reflecting. For example, if this object is silver and is reflecting a blue object at this point, then its color there will be a mixture of silver and blue. To determine this, we shoot a reflection ray out from the intersection point into the virtual world to see what, if any, object it is reflecting (see Figure 2). The ray is defined as the incoming ray mirrored about the surface normal. On the first pass the incoming ray is the ray from the eye point to the object; after that, it is the reflected ray from the previous object. If this new ray intersects an object, we calculate *its* color the same way we did for the original object. We use the lighting equation to combine the contribution of all its visible lights with its own object color. Then, if this object is itself reflective, we shoot yet another ray into the scene and begin again. If not, we stop. With reflective objects, this process can continue recursively indefinitely, but in most cases two or three levels deep is sufficient.

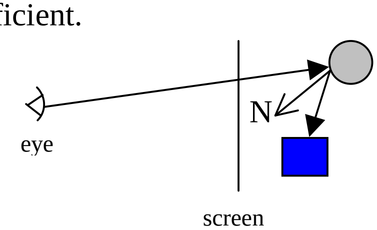



*Figure 2: The ray fired through this pixel hits the silver object. Since the silver object is reflective, so we shoot another ray from it, which hits the blue object, which is not reflective.*

In our implementation, the user can dynamically adjust the recursion level up to a maximum of ten. For more a detailed explanation of how ray tracing works, see [7], [8], or one of any number of sites that discuss it on the World Wide Web, such as [9].

### 2.2 The Lighting Equation

In our implementation, we handle the first rendering of any scene just like a traditional ray trace, except that we store for later use much of the data we compute. The lighting equation we use to determine the color at each pixel is no different from what an implementation of a traditional ray trace might use. However, lighting equations do vary from one implementation to another depending on the features it handles as well as the program's goals, so we will describe ours explicitly. For example, ray tracers that handle transparency or bump mapping would compute additional information that ours does not. Ray tracers that handle only directional and point lights, or those that do not account for attenuation, would compute less information than we do. The lighting equation we use is a variation based on the one presented in *Computer Graphics, Principles and Practice* [7] and the Java3D lighting equation [10]. It handles three types of lights: point, directional, and spot. A point light has a position but no direction; it radiates equally in all directions. A directional light has a direction but no position; it is considered to be an infinite distance from the scene and is often used to simulate sunlight. A spotlight has a direction and a position, and a cone-shaped area of influence. Using these kinds of lights, the color (illumination) $I$ at a given pixel is calculated as follows:

$$I = \left[ k_{ambient} + k_{diffuse} * \sum_{i}^{numlights} l_{diffuse} \right] * \left(1 - k_{reflect}\right) + k_{specular} * \sum_{i}^{numlights} l_{specular} + k_{reflect} * C_r$$

where:

| | | | | |
|---|---|---|---|---|
| $k_{ambient}$ | = | Object ambient color scaled by the user-defined scene ambience (default 1) | | |
| $k_{diffuse}$ | = | Object diffuse color, based on material and texture map (if any) | | |
| $k_{specular}$ | = | Object specular color | | |
| $k_{reflect}$ | = | Object reflectivity | | |
| $k_{intersection}$ | = | Point of intersection on the object at this pixel | | |
| $l_{diffuse}$ | = | $[\, l_{color} * l_{intensity} * \left(L \bullet N\right) ] * atten_i * spot_i$ | = | Light's diffuse contribution |
| $l_{specular}$ | = | $[l_{color} * l_{intensity} * (V \bullet R)^{k\_exponent} ] * atten_i * spot_i$ | = | Light's specular contribution |
| $l_{intensity}$ | = | Light intensity | | |
| $l_{color}$ | = | Light color | | |
| $l_{position}$ | = | Light position (point and spot lights only) | | |
| $l_{direction}$ | = | Light direction (spot and directional lights only) | | |
| $l_{dropoff}$ | = | Light dropoff rate (spot light only) | | |

| | | | | |
|---|---|---|---|---|
| $atten_i$ | = | $\min\left(\frac{1.0}{c_c + c_l d + c_q d^2}, 1.0\right)$ | = | Attenuation coefficient for light *i* (point and spot lights only) |
| $c_c, c_l, c_q$ | = | User defined constants representing the constant, linear, and quadratic attenuation rates, respectively | | |
| $d$ | = | Distance from the point on the object to the light source | | |
| $spot_i$ | = | $[\max((-L \bullet l_{direction}), 0)]^{l_{dropoff}}$ | = | Spot light coefficient for light *i* (spot lights only) |
| $L$ | = | $l_{position} - k_{intersection}$ | = | Light vector (point and spot lights) |
| $L$ | = | $-l_{direction}$ | = | Light vector (directional lights) |
| $N$ | = | Normalized surface normal at the point of intersection | | |
| $V$ | = | Normalized incoming direction vector (initially the view vector) | | |
| $R$ | = | $normalize(2N * (N \bullet -L) - (-L))$ | = | Normalized reflection vector |
| $C_r$ | = | Recursive color (the color computed by calling this function recursively) | = | Reflective value |

*Equation 1: The Lighting Equation*

This equation needs to be computed for each pixel in the scene where an object is intersected. For a 300 by 300 image, that's as many as 90,000 times. Notice that the diffuse and specular terms are essentially the summation of each light's diffuse and specular contributions, times the object's own diffuse and specular colors, respectively. The fact that each light contributes its color independently will be a key factor in allowing fast updates for a user editing a light with our system.

## 2.3 Project Goal

Now that we have examined how traditional ray tracing works, we can explain how and why our implementation is different. While ray tracing is a wonderful way to render scenes photorealistically, it has the major drawback that it is slow. Complex scenes can take minutes or even hours to render because of the object intersection tests and the lighting equation that are computed at every pixel.

Imagine being an artist or animator whose job it is to light a scene. You begin by putting in a few key lights and then rendering. You wait. Once the image is finished rendering, you see that one of them causes shadows where you don't want them, and also is not bright enough. You change the light's position and intensity, then render again. You wait. Now it is causing an unwanted highlight on a character's face. You move the light again, and change its color to a softer hue. You render again, and wait. The color is too blue.

You turn down the blue component and render again, and wait. Now you are satisfied, so you move on to the other key lights and make similar changes, re-rendering as you go. Finally you are happy with the key lights, so you move on to adding fill lights. These lights are for filling in unwanted shadow areas with soft, diffuse light. You decide you need two of them: one on the character's hand and another on a clock that is on the desk. You put a spotlight in, adjust the parameters as you think they should be, and render. You wait. The light is too dark. You adjust the intensity and the attenuation, then re-render and wait. It still is not right, so you decrease the dropoff rate and re-render. You wait. It still looks wrong, so you decide it might work better with a different type of light. You change it to a point light, then re-render and wait. You need to change the attenuation again, now that you are using a different kind of light. You render and wait. Finally it looks good, so you move on to the second fill light. You place it in the scene, adjust some parameters, then render. You wait. The light is far too big. You reduce its angle of influence then render and wait again. Now the size is about right, but the clock you were trying to brighten now looks washed out and the shadows look funny. You change its direction, reduce the intensity, and take out some color. After a couple more tries, you finally get it right. Now you can move on to rim lights[2]…

Clearly, lighting is a time consuming process, the biggest bottleneck of which is rendering and re-rendering. Even the most simple scenes require multiple lights, each of which needs its position, direction, light type, color, intensity, attenuation, dropoff, and/or angle values set and tweaked to achieve just the right effect. For example, the simple scene in Figure 3 required nine lights:

[2] Rim lights are used to add brightness to an object or character's silhouette, emphasizing its outline in order to add the illusion of depth.

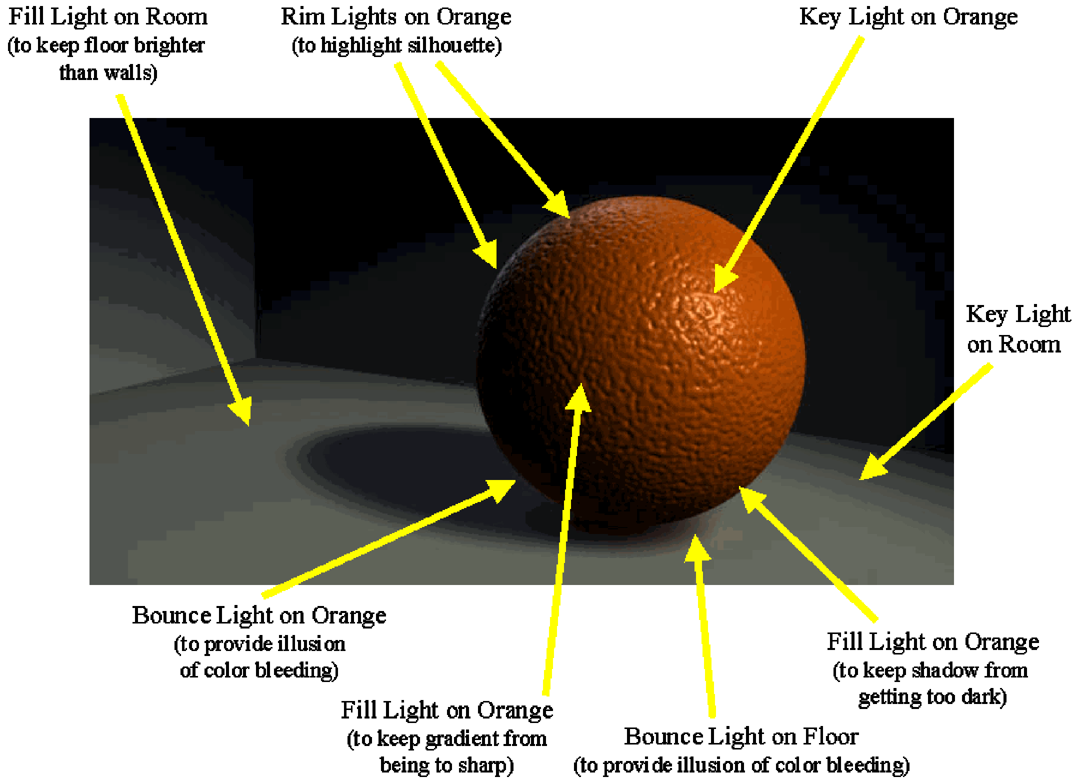



*Figure 3 shows an orange and the various effects that the lights, carefully placed, give the scene. The image was rendered with Pixar's RenderMan.*

While the effects are subtle and may seem unimportant, without this attention to detail the orange would look flat and unrealistic. This image was rendered with Pixar's RenderMan, not a ray tracer, but the principles (and the problem) are the same. With a naïve rendering approach, not saving information from one rendering to the next, the artist has to wait for a complete re-render after s/he makes even the smallest change to a light, in order to accurately see its effect. To compensate, lighters frequently make multiple changes at a time (for example changing color, intensity, and attenuation) before re-rendering, but this makes it more difficult to see which parameter is affecting what, and consequently which need to be changed if the desired effect is not achieved. Other options are to re-render a low-resolution or cropped version of the image, or use a fast lighting algorithm such as Phong shading while making changes. However, as previously mentioned, these methods do not generate the same shadows, reflections, and highlights that a full ray trace would, if they generate them at all. The goal of our project was to create a lighting editor that would give the artist the *exact* image that a full ray trace would produce, but at interactive rates.

### 2.4 Reusing Ray Trace Data for Fast Re-rendering

In order to speed up the process of lighting a 3D scene, we implement a tool specifically for lighting so that we can take advantage of the fact that the only objects that are changing are the lights. This is not an unreasonable restriction, as in most cases by the time an artist is handed a scene, the shot has already been set up; objects other than lights will not be changing. In some cases, an artist will also edit material properties at this point in the production process, but we leave that functionality as an option for future work now that this technique has proven successful.

Recall that computing the lighting equation (which includes, indirectly, intersection testing) at each pixel is what makes a ray tracer slow. However, many of the elements of this equation do not change when lights and their parameters are adjusted. Therefore, we should not redundantly recompute them when we re-render light changes. Specifically, when a user is editing one light, the following components remain constant for each pixel and are saved in our system for reuse:

- The object intersected at that pixel (if any)
- The point of intersection on that object and its corresponding surface normal
- The object's diffuse color or texture map color at that point
- The shapes intersected by the recursively reflected rays if the object is reflective
- The contribution to the pixel's color of every *other* light in the scene

The first key point is that we do not need to perform intersection tests to recalculate which object is visible at every pixel. The second key point is that we do not need to recalculate the contribution from all the other lights in the scene when only one is being edited. This is especially valuable in high-quality, complex scenes, which often have twenty or more lights.

Our algorithm works by performing a traditional, expensive ray trace of the scene on the first pass, while also storing, for each pixel, the object intersected, the intersection point, the normal and diffuse or texture map color at that point, and the shapes intersected by the recursively reflected rays if the object is reflective. In addition, we store the contribution of each light *separately*. In fact, we store each light's diffuse and specular components separately. Thus, when a user renders changes s/he has made to a light, we access the diffuse and specular terms for that light alone, and only recalculate those values. Then, we compose the new values into the lighting equation, which has otherwise remained unchanged. Essentially, the new image is a composite of the old ray-traced image minus the old contribution of the edited light, plus its new contribution. The user is not permitted to edit more than one light at a time without re-rendering in between (though s/he can make multiple types of edits to that light).

When we recompute a single light's contribution for composition into the lighting equation, we do not even necessarily calculate its entire diffuse and specular terms in full. For example, if only the intensity has been changed, we can simply divide the light's diffuse and specular contributions by the old intensity, then multiply them by the new intensity. We do not need to recompute the attenuation coefficient, the spot light coefficient, the dot product between the object normal and the light vector, etc. These and other implementation details are described in more detail in section 4, "Implementation and Results." The key point is that not only do we narrow down the lighting equation to recalculate only the contribution of the light that was edited, we also narrow down how much of that light's contribution needs to be recomputed.

By default, our re-render is not recursive and therefore does not update reflections. In most cases, this is sufficient to give the artist the necessary feedback to evaluate his or her lighting changes, especially because most objects are not reflective anyway. Indeed, most artists would not want to wait for reflections to be updated when they are simply making incremental changes to a light. Therefore, by default our system renders images that are completely accurate (in the sense that they are the same as a regular ray trace) except for reflections. However, when a user wants a 100% accurate ray trace, s/he simply presses the "Update Reflections" button and the system performs the recursive calculations. Even our recursive ray tracing is faster than traditional recursive ray tracing because we have stored the intersected objects for each recursively reflected ray. We still need to recalculate the point of intersection, the normal, and the color contributions of the lights for each recursive layer, so it is slower than our usual re-render, but we do not need to perform the most expensive operation of a recursive ray trace– object intersections. In theory, we could store at every recursive layer all the same information we do for the top level of recursion, but this would make the memory requirements of the program unacceptably high. Already, just storing the information we do uses a lot of memory, which is one of the program's disadvantages (see section 5.2). Still, we feel that the time/space tradeoff is acceptable for this type of application, since memory is inexpensive compared to an artist's time.

# 3 User's Guide

The README that accompanies Lightning contains specific details about its installation and setup. This section will describe for prospective users how to use the application itself. It introduces the various interface features and functions of the system and, because it is meant as a guide for users, it refers, throughout, to "you," meaning the potential user.

## 3.1 Running the Program and Loading a file

To run the program as an application, type:

```
java -mx60m gui
```

To run the program as an applet using appletviewer, type:

```
appletviewer -J-mx60m gui.html
```

You must run the program from its root directory because it uses relative paths to access texture maps, widget icons, etc. It is necessary to allocate extra memory (using the `-mx60m` option) in order for the program to run effectively, since we store so much data at every pixel. Usually 60 megabytes is plenty for a 200 x 200 image, which is the default size. Often 100 megabytes are needed for rendering a 300 x 300 image. Also, this program runs much better if you are using a Just In Time Compiler.

Once the program is open, you can load a file by pressing the "Load File" button on the Canvas Window. On a PC, you will need to click on the "jen" folder to get into the directory where all the scene files are. On a Sun, you will already be in it. Select a .jen file. The .jen file format describes the objects in the scene and their characteristics such as position, rotation, non-uniform scale, shininess, and diffuse, specular, ambient (emissive), and reflective colors. Lights and their characteristics are also defined in the .jen file. Different types of lights have the following different attributes:

**Directional Light:**

Intensity, Color, Direction

**Point Light:**

Intensity, Color, Position, Attenuation

**Spot Light:**

Intensity, Color, Position, Direction, Attenuation, Angle and Dropoff Rate

When you click “OK,” Lightning parses the data and creates a Java3D scene in the right canvas, and a corresponding ray traced image of the scene in the left canvas. Now you are ready to begin editing the lights.

### 3.2 How to Manipulate Lights and Change Their Parameters

Figure 4 shows a screen shot of a file that has just been loaded. The image on the left is the ray-traced image. The image on the right is the Java3D image. The Java3D image looks different because Java3D uses a fast shading algorithm, and treats texture maps as emissive color (meaning it is always drawn to its full brightness, regardless of the lights).

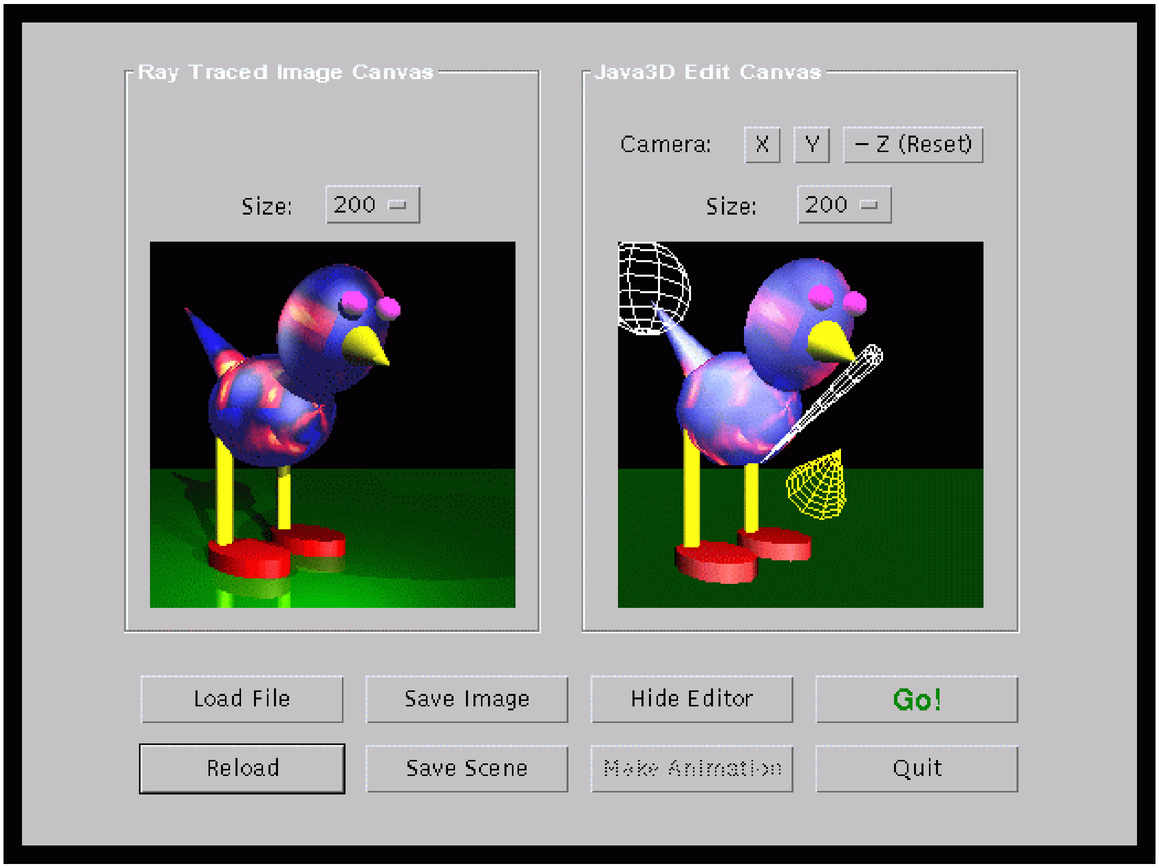



*Figure 4: Lightning's Canvas Window*

To begin working, click on a light in the Java3D canvas. Lights are represented with wireframe geometry: a sphere for a point light, a short fat cone for a spot light, and a long skinny cone for a directional light. In this scene we have all three, and the spotlight is selected. The light turns yellow when selected, and its attributes appear in the Property Window (see Figure 5).

To change a light's position or direction, you simply manipulate it in the Java3D canvas. To rotate it, click and drag it with the left mouse button. Dragging it to the right will rotate it to the right, likewise to the left, up, and down. To zoom the light in and out, click on it with the middle button and drag (if you are not using a mouse with three buttons, press the Alt key and use either the left or right mouse button to get the behavior of the middle button). Dragging the mouse up will zoom the light into the scene, away from you. Dragging down will zoom the object towards you. Finally, to translate the light in the viewing plane, click and drag the object with the right mouse button, and it will follow your cursor. You may notice that you are not allowed to rotate a point light. This is because a point light has no direction and therefore rotating it would be meaningless. However, you are allowed to move a directional light even though it has no "position." This is for convenience as you may want to simply move it out of your way; it will have no effect on the rendering.

To change other attributes, use the widgets in the Property Manager. Because different types of lights have different attributes, certain widgets may be disabled, depending on what type of light you pick.  We have picked a spotlight, which has all the attributes:


*Figure 5: Lightning's Property Editor*

Three sliders with corresponding text boxes control the Red, Green, and Blue components of the light color. Attenuation, Intensity, Angle, Dropoff, and Scene Ambience are all controlled with arrow widgets (in case you are unfamiliar with these terms, they are explained in section 4, "Implementation and Results," or refer to chapter 16 in [4]). To increase the value of one of these parameters, drag the arrow up, or click on the top half of the arrow to increment by a fixed amount (0.10). Drag down or click the bottom half of the arrow to decrement the value. Alternatively, you can type a precise value in the text box. You may also add or delete a light, or change the light type. You may notice that sometimes certain buttons are disabled. For example, the "Go" button is disabled if you have not made any lighting edits since the last rendering since there is nothing new to re-render.  In the above screen shot, the "Add Light" button is disabled because the user has made changes to the spotlight, so s/he is not allowed to add a new light before rendering these changes by pressing "Go" (recall that you can not edit more than one light at a time without re-rendering in between).

### 3.3 Re-rendering

The lighting changes you make are visible in real time in the Java3D canvas because Java3D uses a fast shading algorithm. This gives you a good approximation of the lighting effects you are achieving, which already is an improvement over many lighting editors where you blindly manipulate lights in a wireframe scene. Those environments provide no visual feedback until you explicitly re-render. In Lightning, you have constant, approximate real-time feedback in the Java3D scene, then when you want to update the ray-traced image, you simply press the "Go" button. The ray-traced image is intentionally not updated automatically as changes are made, in order to allow you to make multiple changes before re-rendering. When you finally do re-render, the ray trace update is far faster than it would be to re-ray trace the entire image, because of all the information we saved from the previous configuration (see section 4, “Implementation and Results,” to see just how much faster). Recall that re-rendering only updates the top-level color of the ray-traced image; it does not continue recursively since recalculating reflections is usually not essential feedback for editing lights. When you want to see the effect of your changes on reflections, simply press the "Update Reflections" button after re-rendering. You can also change the recursive depth if you want more or less reflection.

### 3.4 Undo

The program has unlimited undo and redo capabilities for each light as you edit it. Notice that any changes you make using undo or redo after rendering with "Go" must be again re-rendered. As stated previously, the ray trace update is intentionally not done automatically, to allow you to make multiple changes, including undoing and redoing, before re-ray tracing. If for any reason you want to completely start over with the file you are working on, you can just press "Reload" on the Canvas Window.

### 3.5 Saving the Scene

If you like the changes you have made to your scene, you may want to save them. You can do this by pressing the "Save Scene" button in the Canvas Window. This will save your scene as a .jen file (the scene file format) in the directory from which you are currently running the application. Then, you can load the file at a later date and return to the scene exactly as you left it. If you just want to save the picture, not the scene itself, you can press the "Save Image" button in the Canvas Window. This will save the image out to a .xpm file in your current directory.

### 3.6 Canvas Size and Camera Manipulation

In the Canvas Window, there are pop-up menu controls for changing the sizes of both the Ray Traced Image Canvas and the Java3D Edit Canvas. Naturally, resizing the Ray Traced Image Canvas will require a complete new ray trace since the number of pixels will have changed. However, once this is finished you may continue working normally. The Java3D Edit Canvas also has camera controls. You can position the camera in the Java3D scene such that it is looking down the X, negative Y (bird’s-eye view), or negative Z axis by pressing the “X,” “-Y,” and “-Z (Reset)” buttons, respectively (the initial viewpoint is looking down the negative Z axis). You can also use the mouse to manipulate the viewpoint by holding down the Shift key. With any button, a mouse drag that begins mostly vertical (moving towards the top or bottom of the canvas) will zoom the camera in and out. A mouse drag that begins mostly horizontal (moving from left to right or right to left) will result in a camera pan. Finally, a mouse drag that begins close to the edge of the canvas will result in a camera trackball manipulation (the scene moves as if it is enclosed in a virtual sphere). At any point you can return to the original camera position by pressing the "- Z (Reset)" button.

Note that moving the camera in the Java3D scene will have no effect on the ray-traced image. This is because we assume that the "shot" is set up, just as we assume that the objects in the scene (other than the lights) are fixed. This is a reasonable assumption because, in the real world, when an artist is given a scene, the object and camera positions have typically already been decided. Our system could easily be extended in the future to provide this functionality; however, note that a camera move would require a complete re-ray trace since the data stored at each pixel would become invalid.

### 3.7 Animating

Sometimes it is useful to plot out a motion path for a light across several frames, for example to simulate a sunset. To do this, select a light and press the "Make Animation" button on the Canvas Window. If the button is disabled, you must re-render first by pressing “Go.” To record a motion path, press the "Record" button in the Make Animation window, then move the light around as you wish. When you are finished, press "Stop.” Now you can view your animation by pressing "Play,” and it will continuously re-render the ray-traced image as the light moves. You can save your motion path out to a file by pressing "Save Animation.” You can load a path at any time by pressing the "Load Animation" button. The resolution number parameter shown in the window determines how often the light's transform is recorded as you move the light. The default is every 10 frames that are rendered in the Java3D window. Increase the number for more coarse animations; bring it down to 1 for the finest granularity. When you are finished animating, press "Close" and your scene will be restored to the state it was in before you began.  Naturally, if your scene is very complex the animating times will be slow, and only the simplest scenes can actually be animated in real time. Still, this can be a useful tool for rendering a light in many different positions and it is much faster than using traditional methods.

## 4 Implementation and Results: Changing Light Attributes

As stated previously, only one light may be edited at a time. For example, a user can not change the color of one light then another without re-rendering in between (this is not a full ray trace, though– just a re-render). The way the re-rendering generally works is to go through every pixel and recompute the diffuse and specular contributions of the selected light, then recompose them into the lighting equation. The algorithm varies somewhat depending on the type of change the user made to the light, so each change will be discussed individually below. Each is handled differently so as to perform the minimum number of calculations. We also discuss how we recompute composite changes when, for example, the user has changed both a light’s color and position.

## 4.1 Color and Intensity

Color and intensity are the simplest types of changes to recompute, and both are handled the same way so we will only explain it for intensity. Recall the formula for the diffuse and specular contributions of a light:

$$l_{diffuse} = [l_{color} * l_{intensity} * (L \bullet N)] * atten_i * spot_i$$
$$l_{specular} = [l_{color} * l_{intensity} * (V \bullet R)^{k\_exponent}] * atten_i * spot_i$$

These are the only parts of the lighting equation affected by *any* change to a light, and we only need to recompute them for the light that has changed, not for all the other lights in the scene. For an intensity change, we store in each light the intensity value the last time it was ray traced, as well as the new intensity value. When the user re-renders, we can simply divide the light's diffuse and specular contributions by the old intensity value, then multiply them by the new intensity value. Once we have these new diffuse and specular terms, we can compose them into the lighting equation with all of the other values, which have remained constant:

$$I = \left[ k_{ambient} + k_{diffuse} * \sum_{i}^{numlights} l_{diffuse} \right] * (1 - k_{reflect}) + k_{specular} * \sum_{i}^{numlights} l_{specular} + k_{reflect} * recursive\_color$$

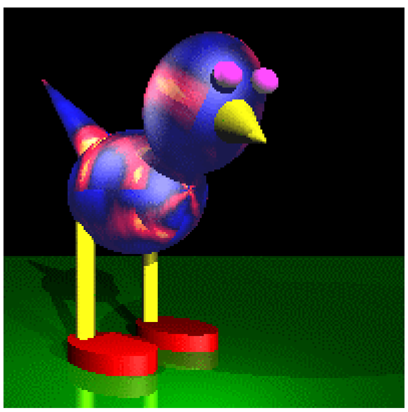
Changed for one light          Changed for one light

But why should we explicitly divide by the old intensity and multiply by the new intensity at every pixel? Instead, we first calculate the ratio of the new intensity to the old intensity and multiply by this value, saving up to 180,000 divisions for a 300 by 300 image. To further optimize, we do not even necessarily do this for every pixel. We can trivially reject pixels that have no object intersection (these are stored as null pixels and are painted the background color, black). We can also trivially skip pixels that are not affected by this light. If, in the previous ray trace, we determined that a given light was not visible from this point, we stored its diffuse color as a negative value. This acts as a flag when re-rendering, indicating that the light does not affect this pixel and therefore would not change due to a change in its color or intensity. Therefore, we only recalculate intensity for pixels that have an object visible at that point *and* where that object's intersection point is visible to the selected light.

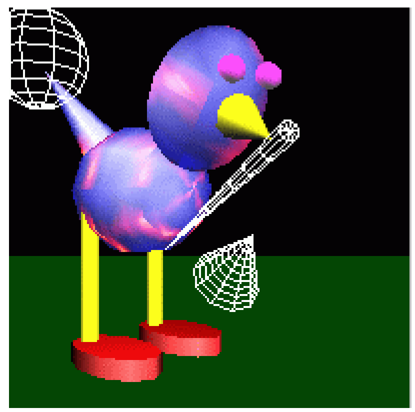
Here and in the following sections we show ray trace examples for a scene with four spheres, two cones, four cylinders, and one plane. It has three lights: one each of point, directional, and spot. It took 22.1 seconds to render the first time with our system, compared to 18.6 seconds for a traditional ray tracer. The extra time we spend is for accumulating and storing the data, which will make re-rendering faster as we edit the lights. When we compare re-rendering times to ray tracing times below, we are comparing them to the 18.6, traditional ray trace speed, to show how much faster our system is than a naïve renderer.  In each case, the rendering times given are for a 300 by 300 image, though the ones we display here are 200 by 200 because of space constraints. Each value presented is an average of three runs of the same test with varying values, running on a 300 MHz Sun Ultra2, single processor machine, with 256 megabytes of RAM. Initially, all the rendering times we give are for a default re-render, which does not recompute reflections. Then, in section 4.8, “Reflections,” we show that re-rendering with reflections adds just 0.6 seconds to the rendering time for this scene.  In Appendix B we show additional images and results, including ones for more complex scenes.

Figure 6 shows the scene after the directional light’s color has changed from white to red. This change took 0.36 seconds to re-render, a 98% savings over re-rendering with a naïve, traditional ray tracer. We also show an intensity change for the same light; it too was re-rendered with a 98% time savings.

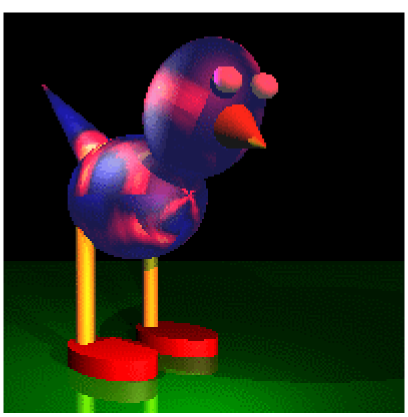
**Original Image**



**Java3D Version of the Original Scene (to show light positions)**



**Re-rendered Image: Color Change**



**Re-rendered Image: Intensity Decrease**

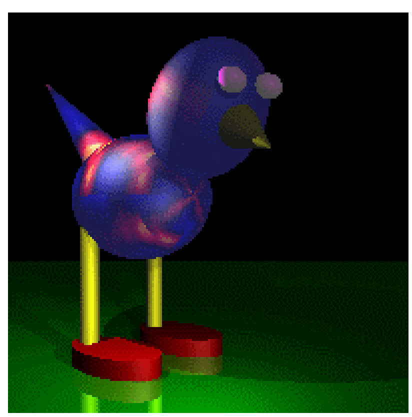


*Figure 6: The top row shows the original image and the Java3D window of this scene. In the bottom left image, the directional light’s color has changed to red. In the bottom right image, its intensity has been reduced.*

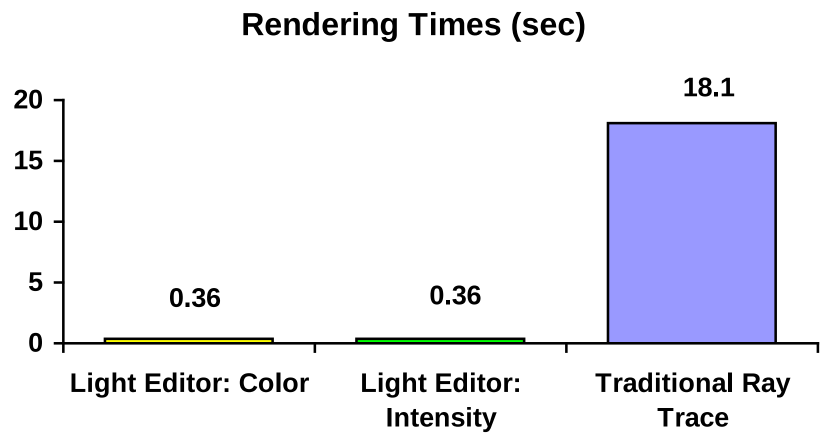



*Figure 7: We re-render color and intensity changes in less than half a second, compared to 18 seconds for a regular ray trace.*

Changes to color and intensity are the fastest kind to re-render, since no object intersections need to be recomputed and all we do to the light's diffuse and specular contributions is multiply each by the ratio of new to old intensity. This amounts to just two multiplies at every pixel affected by the light. However, there is one exception. What if the old intensity value was zero? In this case, we can not simply multiply by the ratio of new to old intensity because the result is undefined. In addition, if the old intensity was zero then the entire diffuse and specular contributions of this light were zero, so scaling them up by the new value would not change them. This is a special situation where we are faced with the "worst case" scenario: we need to recompute the total diffuse and specular contributions of the selected light. While slower than simply doing two multiplies per pixel, it is still much faster than a traditional ray trace because only one light's contribution is being computed. The rest of the lighting equation remains constant.

### 4.2 Attenuation

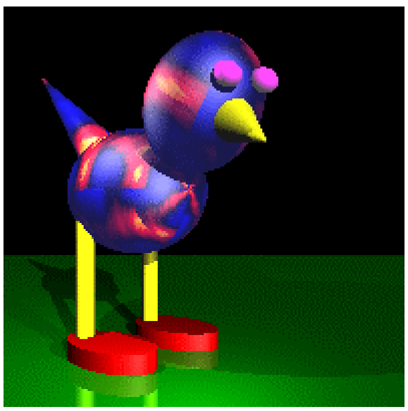
An attenuation change is slightly more complicated to handle than a color or intensity change. Recall the formula for the attenuation coefficient for a light *i*:

$$atten_i = \min\left(\frac{1.0}{c_c + c_l d + c_q d^2}, 1.0\right)$$

This coefficient is designed to scale the diffuse and specular contributions of a light depending on how far away it is from an object. The farther away a light is from an object, the less light the object should get from it. As such, it is meaningful only for point and spotlights, since directional lights are treated as infinitely far away and therefore do not attenuate. Re-rendering after an attenuation change is essentially the same idea as re-rendering after an intensity change: we divide the diffuse and specular contributions of the light by the old attenuation value, then multiply them by the new attenuation value at each pixel affected by the light. This is done in the same way whether the change is to $c_c$, $c_l$, or $c_q$. However, we can not save time by computing the ratio of the new attenuation to the old attenuation up front, as we did for intensity, since attenuation differs at every pixel. The attenuation coefficient is a function of *d*, the distance between the intersection point and the light source, which is different at every point. In addition, we do not store the old attenuation value (since it is inexpensive to compute, but expensive to store since it differs for every pixel), so we need to calculate both the old and new attenuation values before we can divide and multiply them by the light's contributions, respectively.

In this scene, the point light's attenuation was changed from (0.1, 1, 0.1) to (0, 1, 1), making the light seem darker. Here, we achieve a 98% time savings by re-rendering with this system instead of with a traditional ray trace:

**Original Image**



**Re-rendered Image**

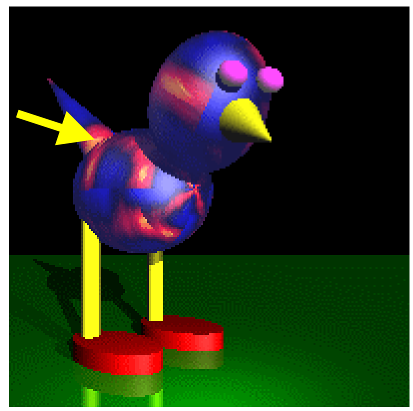


*Figure 8: The attenuation level has increased in the image on the right, making the light less intense on the bird's back.*

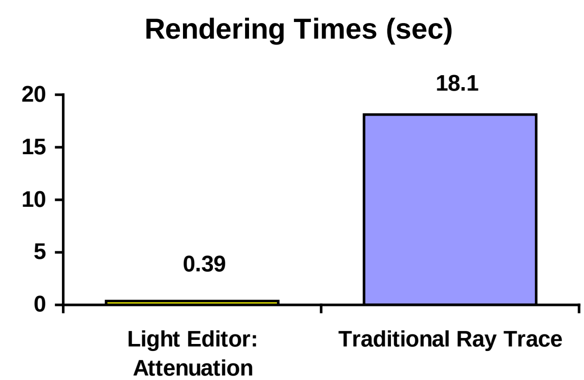



*Figure 9: Lightning can re-render this attenuation change in under a half a second, compared to 18 seconds for a regular ray trace.*

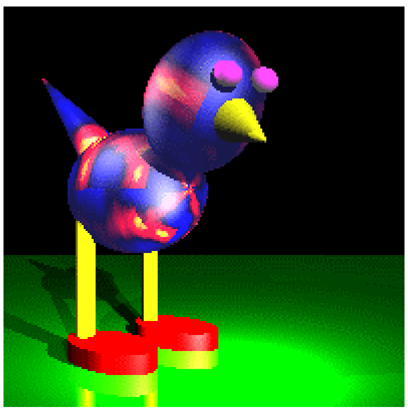
As before, we are occasionally faced with the special case where the previous value was zero. If this is true, we must again compute the total diffuse and specular contributions for the selected light since dividing by the old value is undefined. However, we still have the advantage that we can trivially reject pixels where no object is intersected or where the object intersected is not affected by this light.

## 4.3 Spot Light Angle

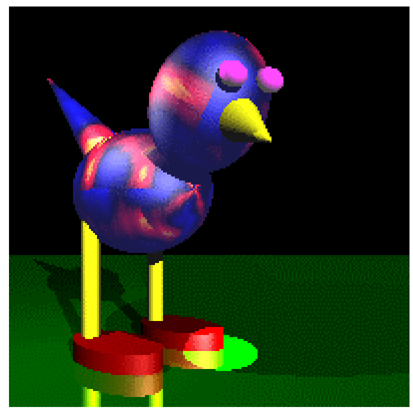
An angle change refers to an increase or decrease in the cone-shaped area of influence of a spotlight. If the angle has been made smaller, we simply go through the pixels previously affected by this light and remove the diffuse and specular contributions for those points that are no longer affected. If the angle has been made bigger, we must check all pixels to see if they are now affected, and if so, compute the total diffuse and specular contributions of the light– but only for those pixels that were not previously affected. Those that were affected before will have unchanged values.

Figure 10 shows the results both of increasing and decreasing the angle:

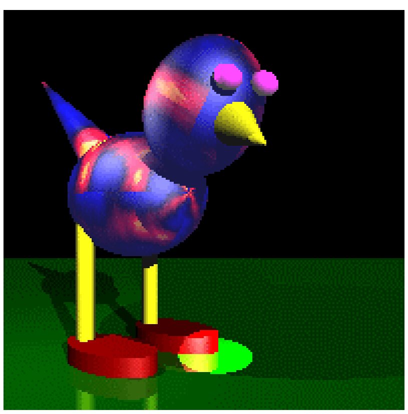
**Original Image**



**Re-rendered Image: Angle Decrease**



**Original Image**



**Re-rendered Image: Angle Increase**

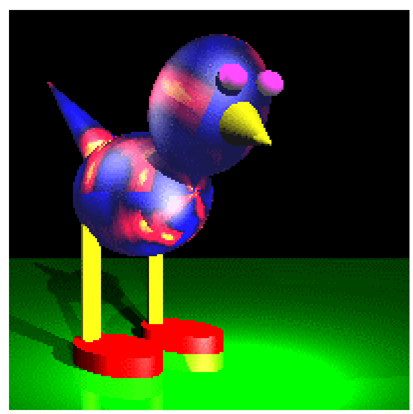


*Figure 10: In the top row, we re-render after an angle decrease. In the bottom row, we re-render after an angle increase.*

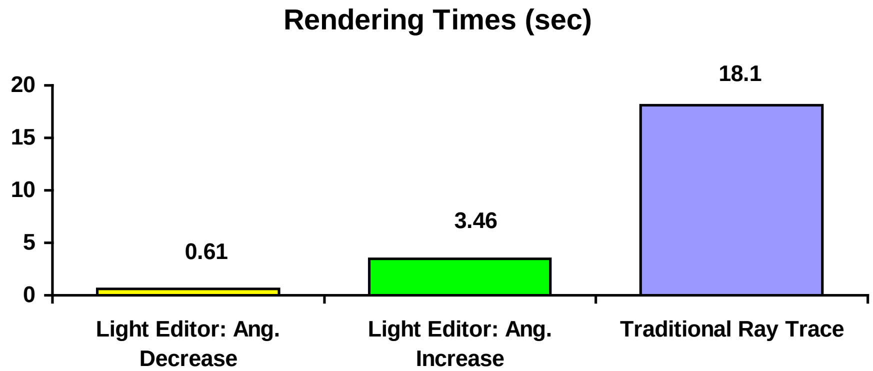



*Figure 11: An angle increase is notably slower than an angle decrease, since new pixels may now be affected.*

Here we save 97% of the rendering time for an angle decrease and 81% for an angle increase, compared to regular ray tracing.

## 4.4 Spot Light Dropoff Rate

Like angle, dropoff rate is only meaningful for spotlights. The dropoff rate refers to how fast a spot light's intensity falls off from the center of its area of influence. When a change is made to a spot light's dropoff rate, we use the new value of $l_{dropoff}$ to recompute the spot light coefficient:

$$spot_i = \left[\max\left(\left(-L \bullet l_{direction}\right), 0\right)\right]^{l_{dropoff}}$$

Then, we divide the diffuse and specular contributions by the old spot light coefficient and multiply them by the new one, as usual. As for attenuation, we must compute both the old and the new values in order to update the light's contribution, since we do not store the spot light coefficient separately for each pixel. However, we can still save calculations by trivially rejecting pixels not affected by this light. Note that the definition we use for "affected" is that this light *potentially* affects the point at this pixel. It does not necessarily mean that the point gets any illumination from it. For example, consider the scene in Figure 12, where Spotlight A is visible from Point B because B is in A's cone-shaped region of influence, and is not blocked by (in shadow from) any other object. However, A's dropoff rate has made its illumination fall off such that Point B is outside of A's region of illumination:

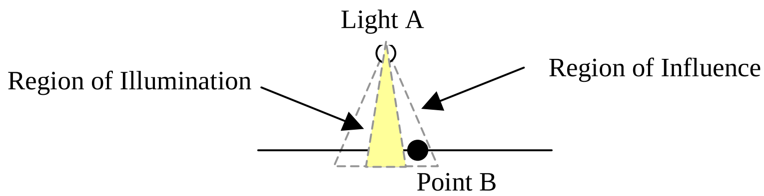



*Figure 12: In this example, Light A's dropoff rate is such only the yellow region is illuminated. However, Point B is still considered "affected" by Light A, and is not trivially rejected when re-rendering a dropoff change (because it might now be illuminated).*

Defining "affected" this way (as "is potentially affected by this light") allows us to trivially reject all pixels not affected by this light when re-rendering a dropoff change. If we had instead defined it as "gets illumination from this light," then we could not reject unaffected pixels when re-rendering a dropoff change, in case the point should now be illuminated.

The spotlight on the left has a low dropoff rate, and was re-rendered on the right with a high dropoff rate. Here we achieve a 95% time savings over traditional ray tracing:

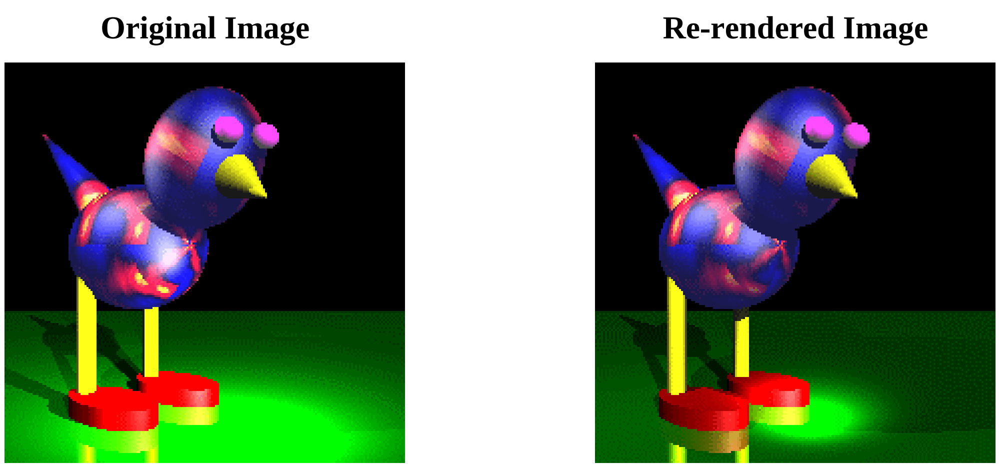



*Figure 13: We use Lightning to decrease the spotlight's dropoff rate, making its illumination fall off faster from the center.*

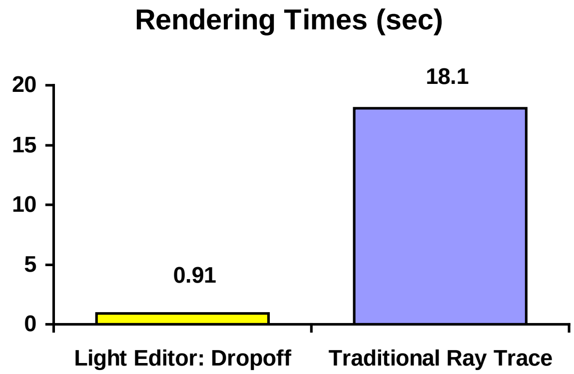



*Figure 14: We re-render this scene after a dropoff rate change in under a second.*

## 4.5 Position, Direction and Light Type

In most of the cases above, we were able to save a lot of computation time by trivially rejecting pixels not affected by the light being edited. However, this time-saving technique cannot be used when a light's position or direction has changed because new points may now be affected. It also cannot be used when we change a light's type because each have different rules about whether or not they are considered visible from a given point and are therefore affected. For example a point light and a spot light can both have the same position, but a point light will affect more surface points because it radiates equally in all directions– its effect is not cut off by a cone angle. As a result, when a user changes from one light type to another or s/he physically manipulates a light, a re-render must perform the "worst case" calculation. That is, it must recompute the light's total diffuse and specular contributions. But remember that the majority of the lighting equation is still reusable: the object intersected at each pixel, the intersection point, the surface normal and diffuse or texture map color at that point, the shapes intersected by the recursively reflected rays if the object is reflective, and the contributions of all the other lights in the scene. In addition, we can still trivially reject pixels, which have no object intersections.

In the first pair of images in Figure 15, we have changed the position of a point light from the upper left to the middle, so that now the underside of the bird is brighter, and the top of the bird is darker. In the second pair, we have changed the direction of a directional light, from pointing towards the left to pointing towards the right front (notice the different shadows). In the third pair, we have changed the light type from directional to point, which results in a darker image because of its attenuation and position (which the directional light did not have).

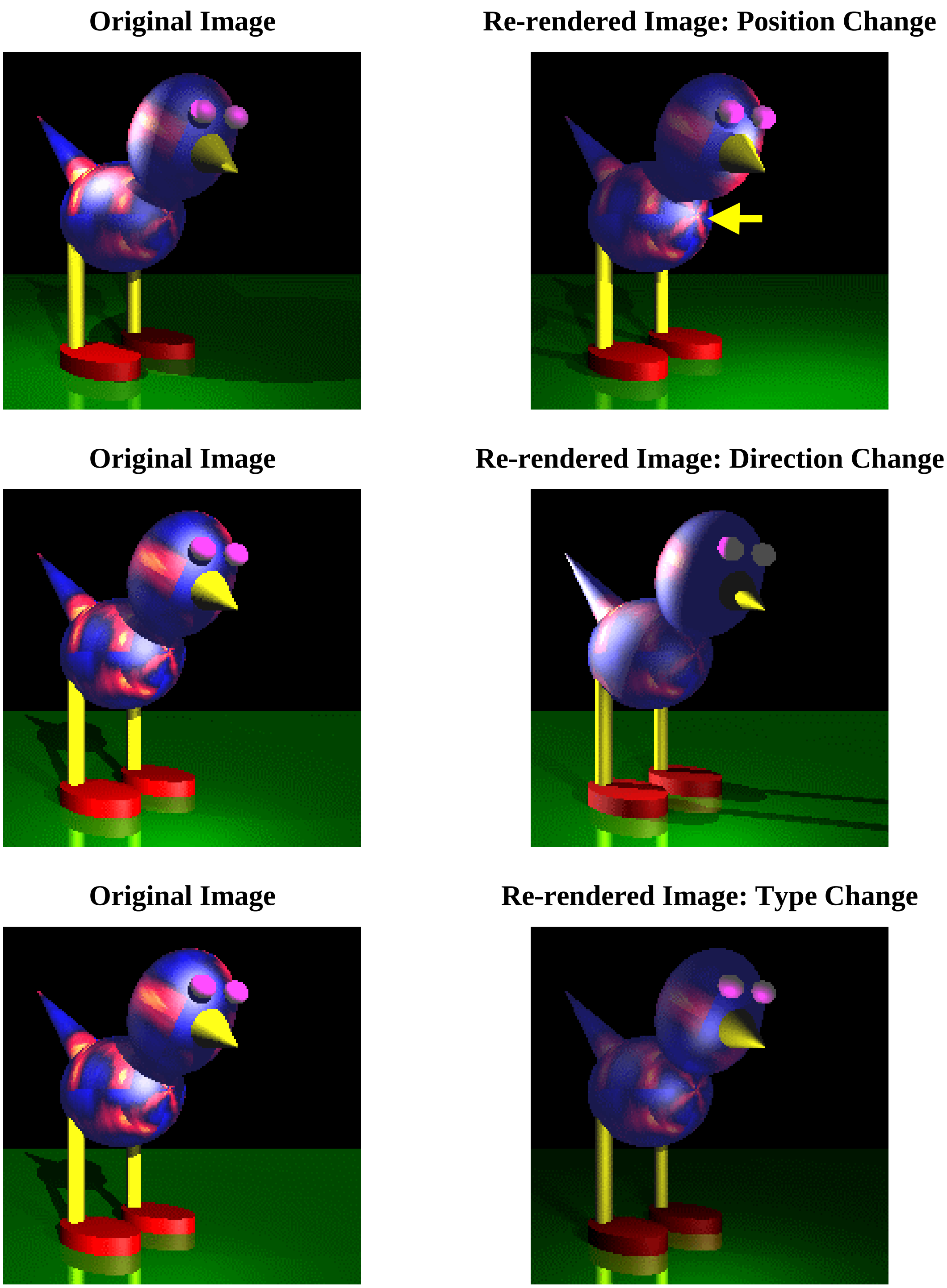



*Figure 15: In the first row, we re-render after a point light position change. In the second row, we have changed the directional light's direction. In the third row, we have changed the directional light to a point light.*

.

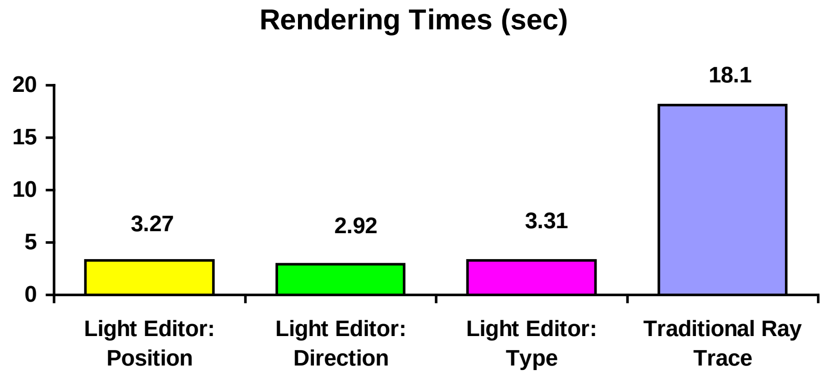



*Figure 16: Position, direction, and light type changes all take roughly the same amount of time, since they all need to compute the light's diffuse and specular contributions in total.*

For these types of changes, we see less of a time savings than for those edits which do not require recomputing the total contribution of the edited light. Still, we save over regular ray tracing since the data for all the other lights in the scene is unchanged. For this scene, we save 82% for the position change, 83% for the directional change, and 82% for the type change.

### 4.6 Adding and Deleting Lights

Deleting a light is obviously a very simple re-rendering calculation– we just subtract the contribution of that light. However, there is also some overhead associated with the change, since we must reallocate and shrink the arrays that store the diffuse and specular light contributions at each pixel. Still, in this example we achieve a 95% time savings over regular ray tracing when we delete the directional light (notice how much darker the bird is):

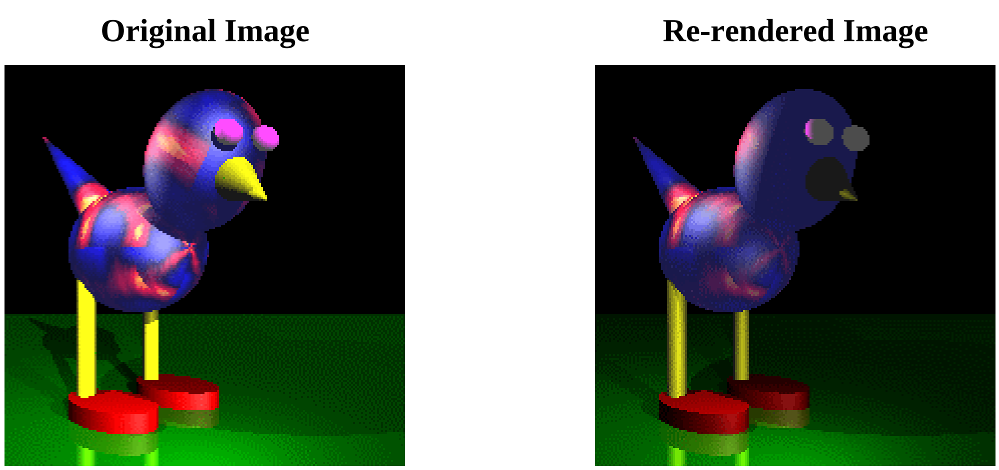



*Figure 17: We re-render after removing the directional light.*

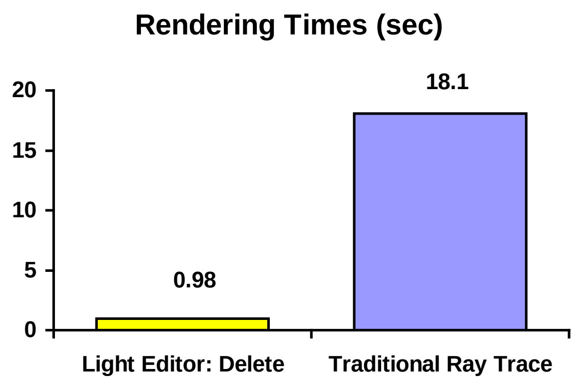



*Figure 18: Deleting a light is an inexpensive change to re-render.*

Adding a light also requires array management overhead, in this case to extend the length of the arrays. Obviously, we must also calculate the new light's total diffuse and specular contributions because we have no previous data for it. These values are then composed into the lighting equation as usual. Adding a new light is the most expensive light edit in our system, but we still achieve significant savings over regular ray tracing. Just how much we save is directly related to how many lights are in the scene. If we were adding a light to a scene that just had one other light, we would save roughly just 50% of the computation time of a regular ray trace, since half of the lights' contributions do not need to be computed. If we were adding a light to a scene that had nine lights in it already, we would save roughly 90% over rendering with a traditional ray tracer. Here we add a light to a scene with three other lights, and achieve a 71% savings (notice how the bird is brighter):

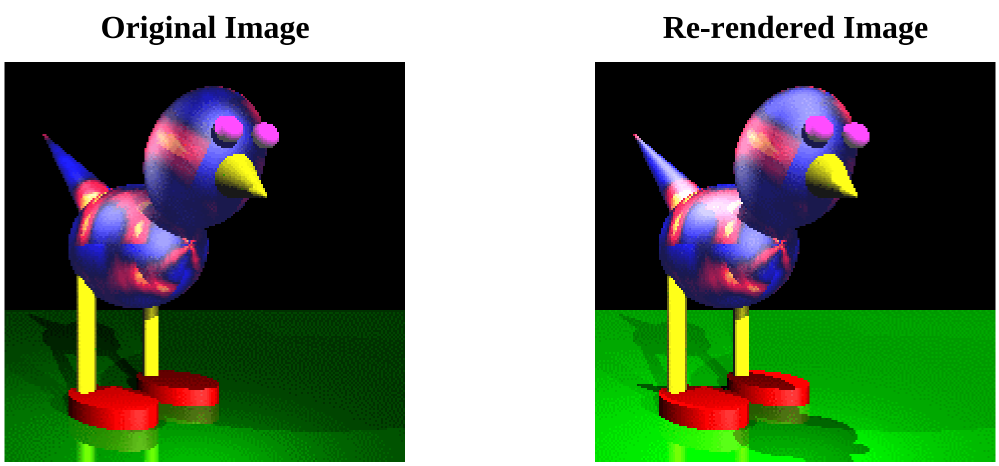



*Figure 19: We have added a directional light.*

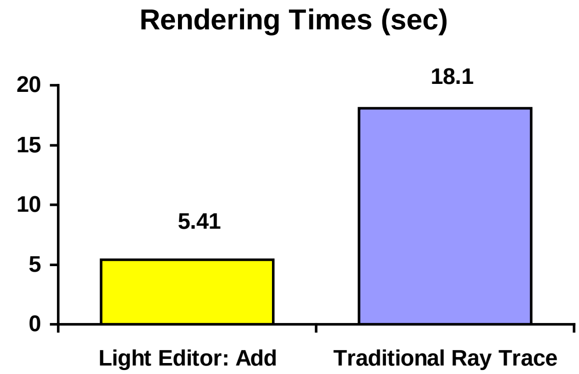



*Figure 20: Adding a light is the most expensive change to re-render with Lightning, yet it is still significantly faster than performing a traditional ray trace.*

We considered using a linked list or some other data structure more suited to dynamic allocation to hold the lights' diffuse and specular contributions at each pixel, but we concluded that the cost of traversing the list in all other situations, where dynamism is not needed (such as for color or attenuation changes) outweighed the benefit of making adding and deleting lights faster. Indeed, changes to light parameters occur far more often then adding and deleting lights does. For example, consider that for every new light a user adds s/he then makes numerous changes to its parameters to get it to look right.

### 4.7 Scene Ambience

Scene ambience is a special case because the lights in the scene are actually irrelevant to its effects. It simply determines how bright we want ambient colors to be. However, it is still a parameter that should be in a lighting editor because it can affect the overall brightness of the scene. Ambient "light" is an approximation of multiply-reflected light in a scene, which cannot quickly be computed by a lighting equation like that given in section 2.2. In the real world, an object that is not directly lit by any light is often still visible because of light that bounces off of the objects around it. Therefore, an ambient light term is often used in computer graphics to ensure that objects defined by the lighting equation as completely in shadow are not rendered as total blackness.

When the user changes the scene ambience, all s/he is really changing in the lighting equation is $k_{ambient}$, the value of the object's ambient color scaled by the scene ambience. Therefore, all we need to do is recompose the lighting equation for every pixel with this new scaled value. We do not need to update the values of the contributions of any of the lights, and we can reject pixels where there is no object intersection, as usual. Here we change the scene ambience from 1.0 to 0.1, which darkens those objects which have ambient color (everything but the feet, in this scene). This re-render gave us a 98% time savings over regular ray tracing.

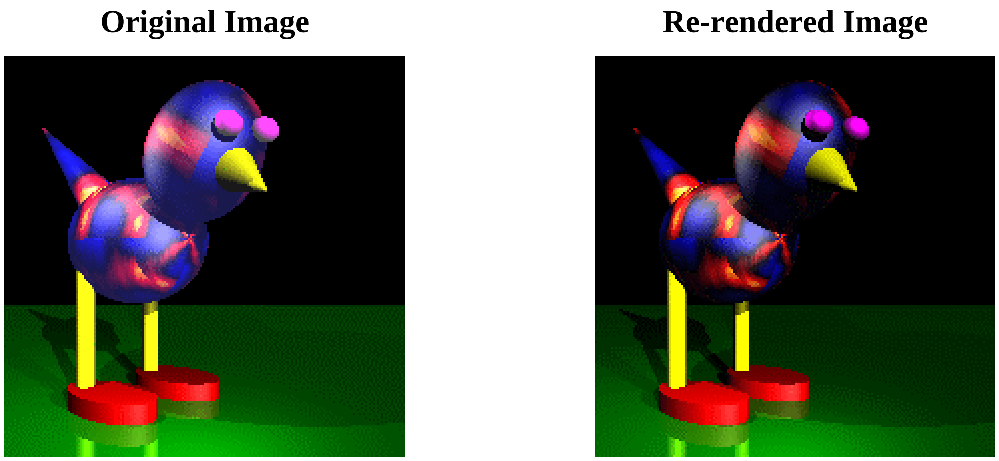



*Figure 21: The scene ambience has been reduced, making objects with ambient color less bright.*

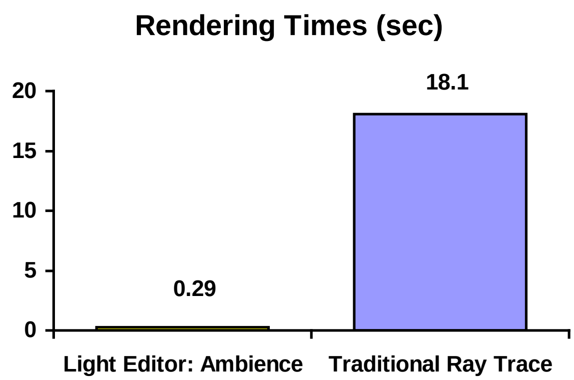



*Figure 22: We can re-render after an ambience change in 2% of the time it takes to do a regular ray trace.*

### 4.8 Reflections

Rendering the top recursive level is generally good enough to provide an artist with sufficient feedback to evaluate the light changes he or she is making. The re-rendered image is just as accurate as a traditional ray trace, except for the reflections. To give at least an approximation of the reflections, the old reflective value is saved and reused in the lighting equation, since in most cases it will have changed little, if at all:

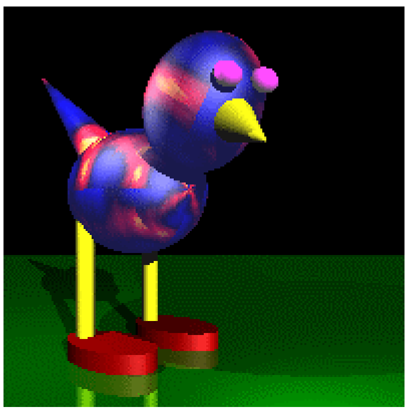
$$\text{reflective value} = k_{reflect} * C_r$$

$k_{reflect}$ is obviously unchanged since it is an object property, and the recursive color $C_r$ has only changed for reflective objects directly or indirectly affected by the light being edited. Still, there are times when we need the re-rendered image to be exact, including reflections. To achieve this, the user presses the “Update Reflections” button and we recompute the $C_r$ term and compose it back into the lighting equation.

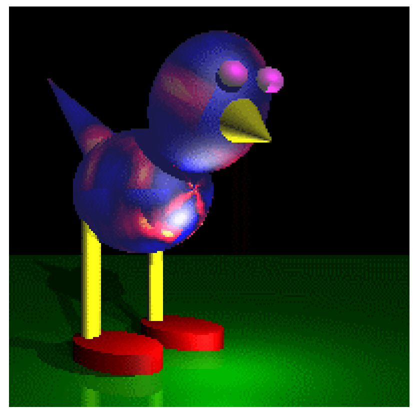
To recompute the $C_r$ term, we use the list of shapes we have stored which indicate the object intersected by each recursive ray. We need to compute the complete lighting equation at every recursive layer (except the first), but we save time by not having to perform intersection tests to determine to which object it applies. As stated previously, we could, in theory, store all the lighting equation information for every recursive layer like we do for the top layer, but the memory requirements would be too large. Instead, we picked the most expensive calculation, object intersection, and saved that result for each recursive ray, so it is still faster than traditional ray tracing. The list of intersected objects has length ten, the maximum level of recursion the user can set.

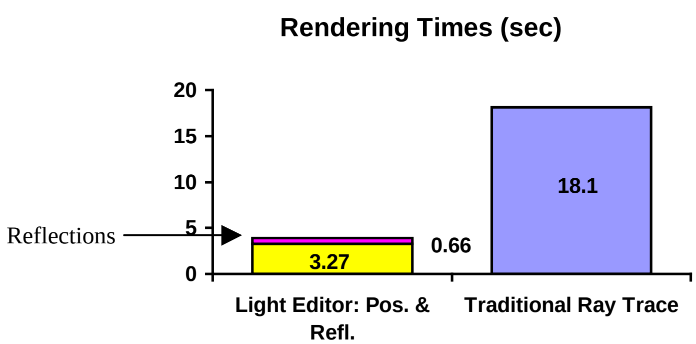
In Figure 23, the point light's position has been moved from the upper right to near the bird's feet. First we do a normal re-render, by pressing "Go," then we update the reflections.

**Original Image**



**Re-rendered Image: No Reflection Update**






**Re-rendered Image: Reflections Updated**

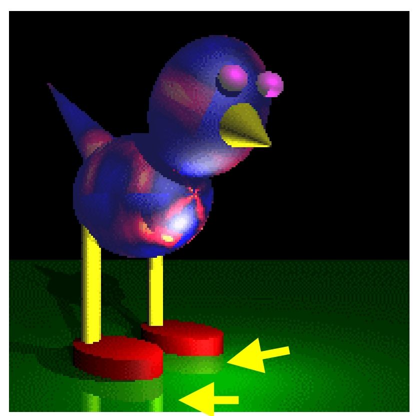


*Figure 23: Looking closely on the floor by the bird's feet, one can see the effect of updating the reflections. Notice how the image in the upper right, where the user has not yet updated reflections, still gives a useful representation of the effect of the point light's position change. However, as the graph on the lower left shows, updating reflections in this scene only adds a small fraction to the total rendering time.*

### 4.9 Composite Changes

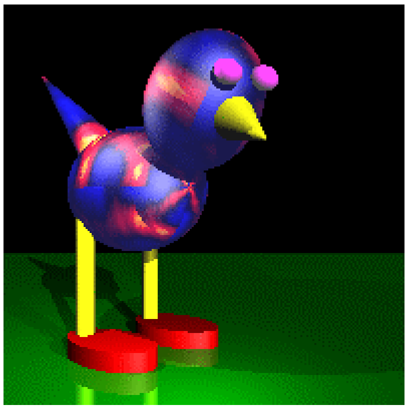
Often, a user makes not one but multiple changes to a light's parameters before re-rendering. The question is, how can we recompute the minimum necessary data while still accounting for every change? Recall that some changes, such as position, require what we call a "worst case" recalculation– we have to compute the total diffuse and specular contributions of the edited light. Still, this "worst case" is not nearly as expensive as a traditional ray trace; it is merely the worst case in our application. If we have both added a light and then changed its type, both of which require this "worst case" evaluation, we do not want to compute it twice. Doing it for one will take care of it for the other, since the first recalculation will use all of the most up-to-date data about the light, including its type. Likewise, if we have changed a light's position as well as its color, we do not want to calculate the worst case (for the position change), then scale by the ratio of new color to old (for the color change). The color change will have already been accounted for in the first computation since the new color value will have been used during the position change recalculation. Another example is if the user first changed a light's color, then deleted the light. We do not want to bother computing the color change since the light no longer exists. Clearly, the operations and the order in which they are performed are critical, not only for speed but also for accuracy in some cases. The pseudo code for re-rendering the ray-traced image is as follows (null pixels are those which are always painted black because no object was intersected):

```
if light was added
     for every pixel != null
          pixel.updateAdd()
else
     if light was deleted
          for every pixel != null
               pixel.updateDelete()
     else
          if there was a light type change, position
          change, or direction change
               for every pixel != null
                    pixel.updateFull()
          else
               if there was an ambience change
               or an angle change
                    for every pixel != null
                         pixel.update()
               else
                    for every pixel != null
                         if pixel is affected by this light
                              pixel.update()
```

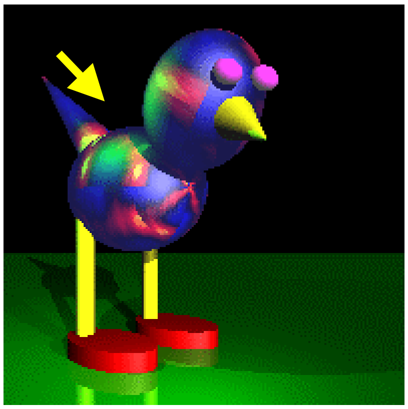
The functions `pixel.updateAdd()`and `pixel.updateDelete()`reallocate the arrays of diffuse and specular light contributions and then recompose the lighting equation with the new, or without the old, light's values, respectively. The function `pixel.updateFull()` performs the worst case recalculation, where the "full" diffuse and specular term is recomputed for the edited light then composed back into the lighting equation. The function `pixel.update()` is a more generic update method which determines within the pixel what needs to be recalculated, and can thus calculate the effects of multiple changes together. For example, if just the light's color and intensity were changed, `pixel.update()` would be called and would scale the light's diffuse and specular contributions as described in section 4.1. However, if a light's color, intensity, and position were changed, then everything would get taken care of in the `pixel.updateFull()` function. Notice that if the changes were just to color, intensity, dropoff, and/or attenuation, then we can not only trivially reject background pixels, but also those not affected by the edited light (see the last three lines).

In the first example in Figure 24, the user has made changes to the color, intensity, and attenuation of the point light in the scene. Re-rendering these changes takes 0.3 seconds, still a 98% time savings over traditional ray tracing, despite the fact that multiple parameters changed[3]. In the second example, the user has changed the spotlight's direction (it is now pointing up at the bird), angle, intensity, and dropoff rate. Here we re-render in just 3.4 seconds, an 82% time savings over regular ray tracing.

**Original Image**



**Re-rendered Image: Composite 1**



[3] This number is smaller than the re-rendering times for color, intensity, and attenuation times in sections 4.1 and 4.2 because this light happens to affect fewer pixels than the lights that we edited in those examples do.

**Original Image** | **Re-rendered Image: Composite 2**

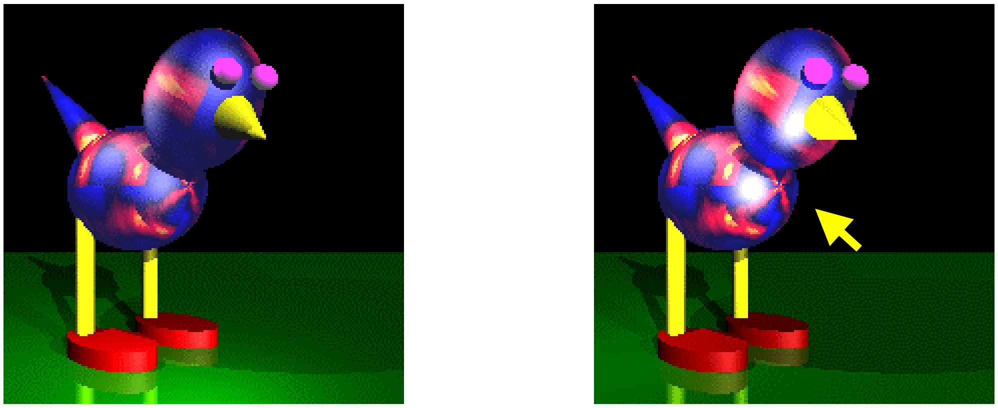


*Figure 24: In the first pair of images (previous page), the user has changed the point light's color, intensity, and attenuation. In the second pair, the user has changed the spotlight's direction, angle, intensity, and dropoff rate.*

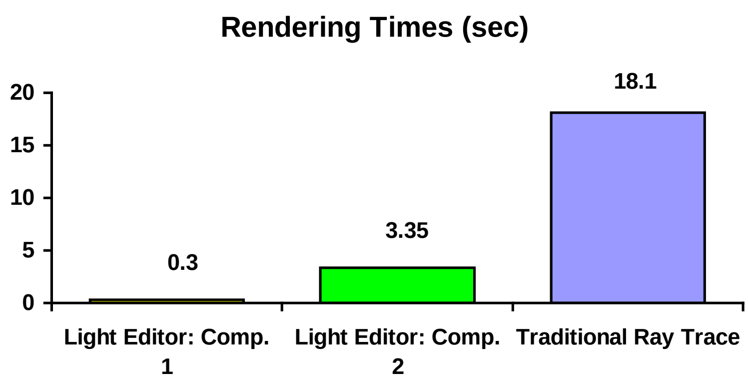



*Figure 25: Even composite changes are re-rendered in a small fraction of the time it takes to do a traditional ray trace.*

Even though we do not re-render reflections by default when we update the ray-traced image, we can still provide the user with an image that is *exactly* what a full rendering would generate, in a small fraction of the time, if s/he presses the "Update Reflections" button. If we simply add the "Update Reflections" time to the re-rendering times of all the examples given above, we can see that the total time is still far less than it takes to perform a complete ray trace:

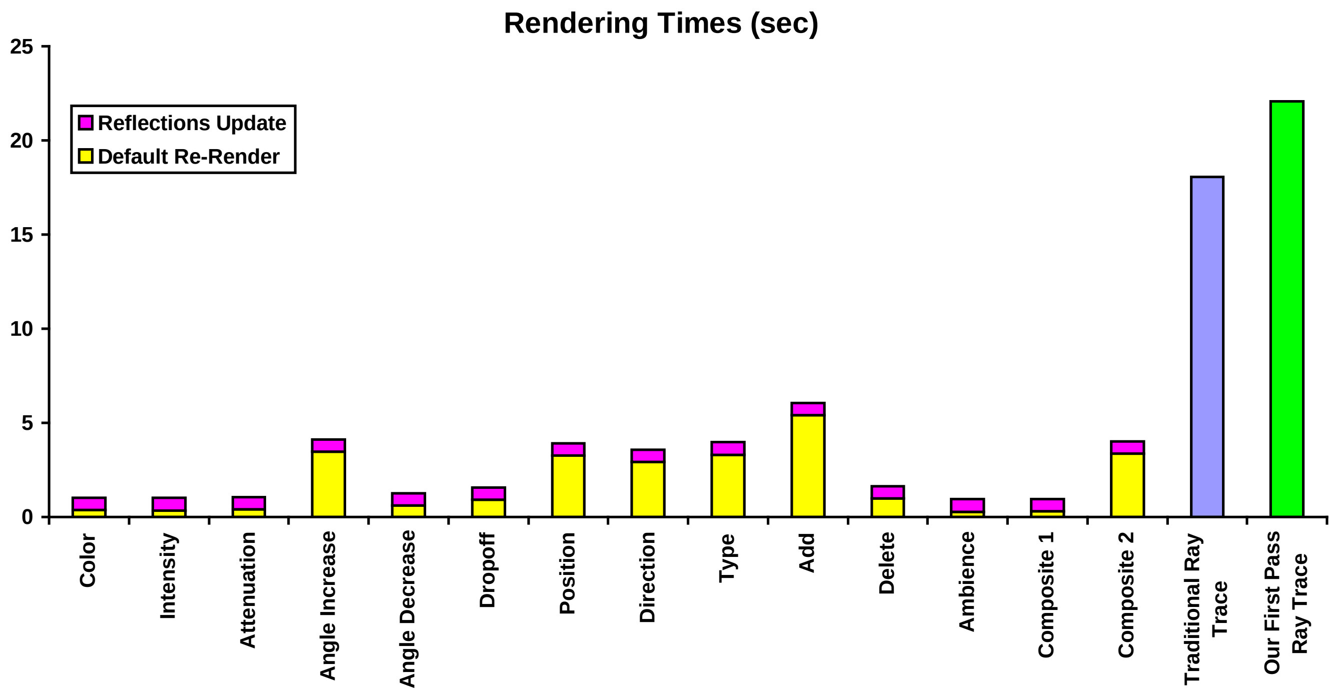



*Figure 26: Rendering with reflections gives the user the* exact *same image as rendering with a traditional ray trace, but in far less time.*

# 5 Future Work and Conclusion

## 5.1 Advantages

The biggest advantage of Lightning is that it re-renders scenes in a small fraction of the time that it takes to re-render them with regular ray tracing. Most lighting editors do not take advantage of the previous image when rendering a new image after an artist has made changes to a light. While many systems do provide mechanisms for rendering quickly, such as generating low resolution, cropped, or fast-shaded images, these methods fail to generate the same shadows, highlights, and reflections that a full rendering would. This makes it difficult for the artist to evaluate his or her changes both accurately and interactively. In contrast, we provide a system which renders lighting changes in far less time than with regular ray tracing, but just as accurately, with the exception of reflections which can also be rendered quickly when desired.

## 5.2 Disadvantages/Future work

The most obvious drawback to the system is also what gives it its strength: the restriction that only the lights may be edited. However, we believe that people whose sole job it is to light scenes, as in a film or commercial production environment, should have a tool designed specifically for them rather than using tools that also handle modeling and animating, if it can make their work significantly faster. At the very least, their tool should be able to go into a lighting mode, to take advantage of the fact that only the lights are changing. Still, it would be helpful if material properties, not just lights, could be edited since the two combined are what really determines the illumination of a scene. The other disadvantage of the program is that it has a very large memory requirement. While it does not increase significantly with the complexity of the scene or the number of lights (as in [4]), it increases linearly with the number of pixels. Each pixel requires 250 bytes of storage for a scene with 4 lights and 10 shapes, which means it takes at least 23 megabytes to store the data for a 300 x 300 image, plus the expense of running the application itself. Future implementations could include data compression techniques, perhaps by grouping neighboring, similar pixels as in [5]. Other areas for future work include adding more features to the system, such as bump mapping, anti-aliasing, camera position changing, and the ability to handle more complex shapes. Finally, we could compose other ray tracing optimizations into our system, such as using bounding volumes to reduce intersection tests.

### 5.3 Conclusion

We have presented a method for quickly re-rendering scenes with results comparable to traditional ray tracing, but in a small fraction of the time. Artists and animators, who need to re-render scenes dozens of times as they make changes to the lights, could benefit greatly from this kind of system, which significantly reduces the time it takes to light a scene. Clearly, taking advantage of information from previous renders when making any kind of incremental changes to a scene has the potential to reduce re-rendering times considerably. We have shown that for lighting changes, re-rendering times can be reduced by up to 98%.

# 6 Appendix A: Software Architecture

We chose to implement the editor in Java using Java3D in part because we wanted it to be multi-platform. In addition, the 2D widgets and 3D geometry provided in the class libraries allowed us to focus on the algorithm during implementation, rather than on lower-level details. While C++ was roughly twice as fast in prototype tests, Java's performance was good enough to provide the interactivity we desired. Furthermore, the important proof of our concept lies in the *relative* difference between traditional ray tracing and rendering with our system, both of which we implemented in Java.

In *Diagram I* on the following page we show a high-level overview of the Java classes we used to implement our lighting editor. The notation is a variant of standard Object Modeling Technique (see legend). Our class names begin with a lower-case letter, while Java classes begin with an upper-case letter. We do not show lower level utility classes and classes pertaining more to the user interface than to the algorithm itself. The classes across the top are the three main engines driving the system. `gui` is the Applet or Application itself, which manages the graphical user interface. `render` is the ray trace canvas which includes the functions for ray tracing and redoing the ray trace after the user makes changes to the lights. `j3dSceneCreator` creates a Java3D scene that has a one-to-one correspondence to the ray-traced scene, plus wireframe representations of the lights, which can be manipulated. At the bottom we show the pixel class, which holds the most important functions: those for quickly updating the pixel color depending on the changes the user made to the lights, and based on the data we store there from previous renderings.

## Code Structure:

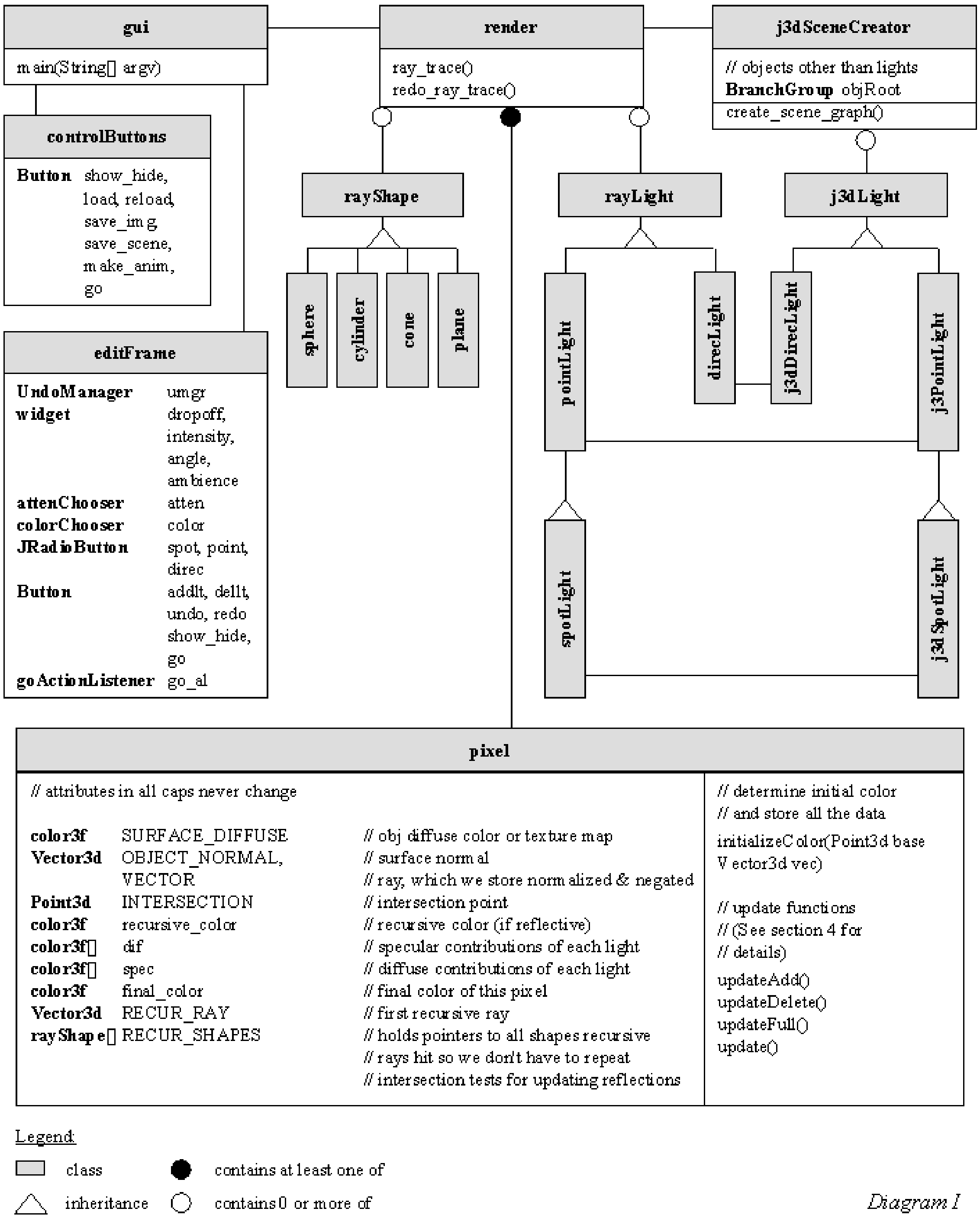


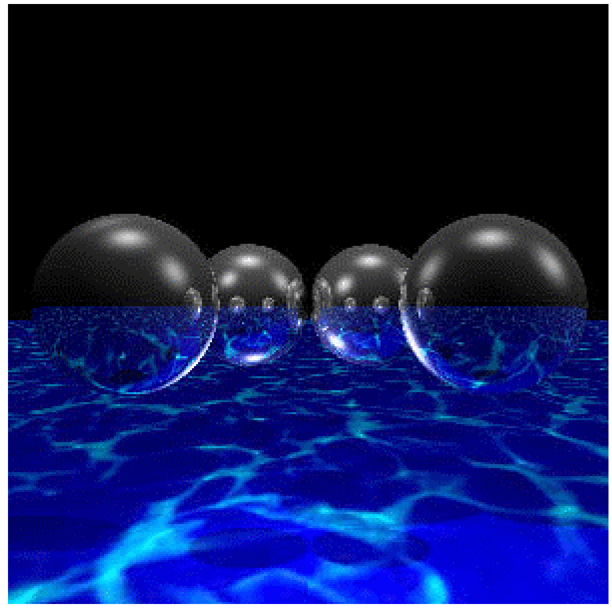

*Diagram I*

# 7 Appendix B: Results and Images

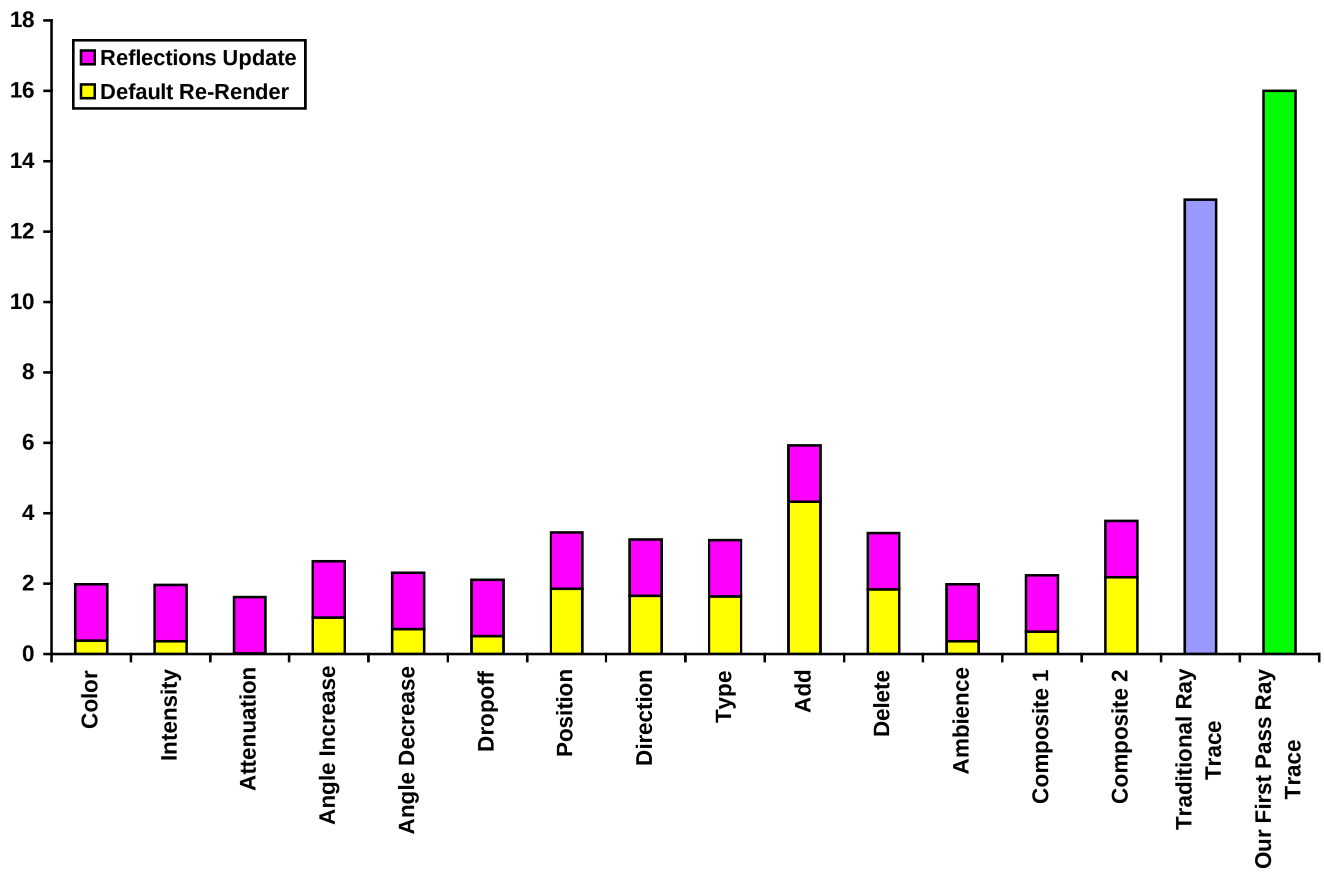
As in section 4, the images on the following pages were all rendered with our system on a 300 MHz Sun Ultra2, single processor machine, with 256 megabytes of RAM. The tests were all performed on images size 300 by 300 pixels, though comparable relative results were found on images of different sizes. Each value is an average of three runs of the same test with varying values, and both traditional ray trace and our first pass ray trace rendering times are included for comparison against re-rendering times. Notice how some of the changes, such as color, intensity, light deletion, ambience, and attenuation do not significantly increase with the complexity of the scene. The re-rendering times for these changes depend primarily on the number of pixels, not the number of objects in the scene, since they require no object intersection testing.

## Example 1: Reflective Balls

| | |
|---|---|
| Spheres | 4 |
| Cones | 0 |
| Cylinders | 0 |
| Planes | 1 |
| Point Lights | 1 |
| Spot Lights | 1 |
| Directional Lights | 2 |



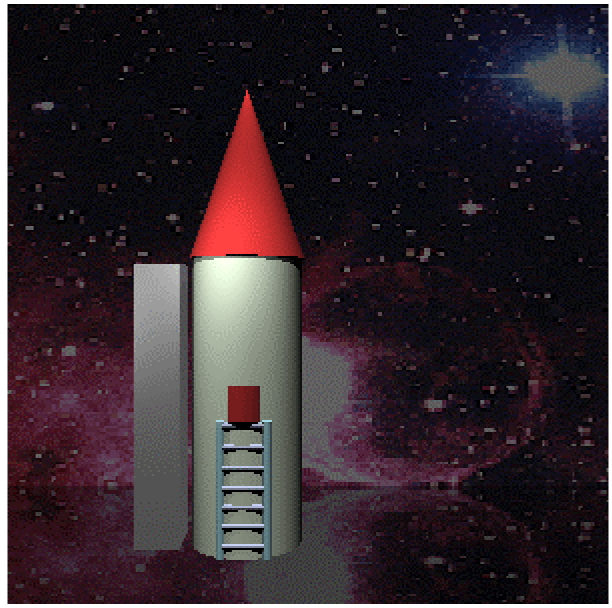
Rendering Time (sec)



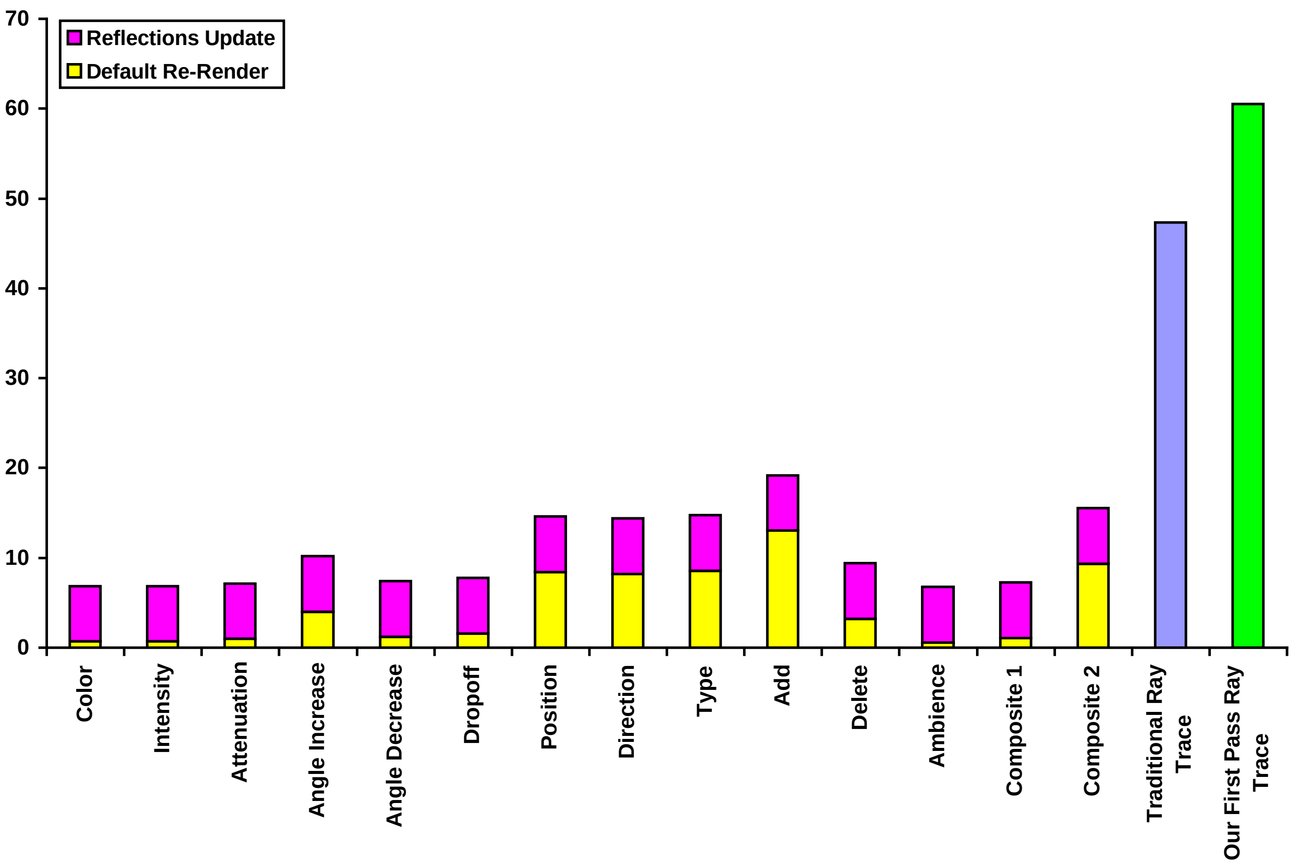

Type of Light Edit

## Example 2: Space Ship

| | |
|---|---|
| Spheres | 0 |
| Cones | 1 |
| Cylinders | 11 |
| Planes | 8 |
| Point Lights | 2 |
| Spot Lights | 1 |
| Directional Lights | 1 |



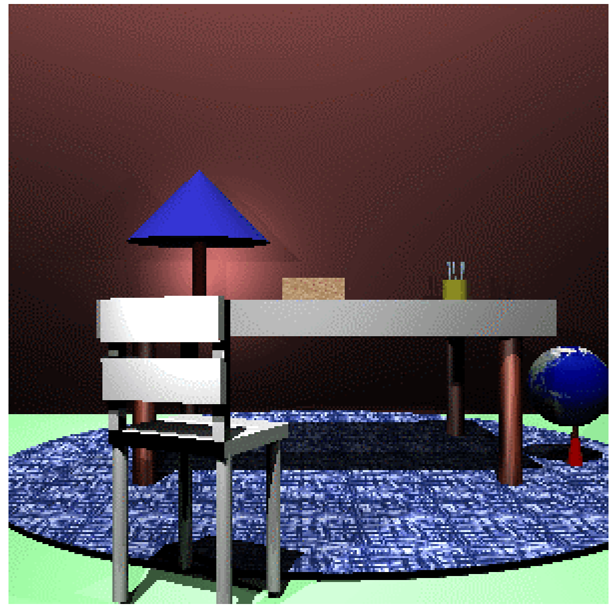
Rendering
Time (sec)



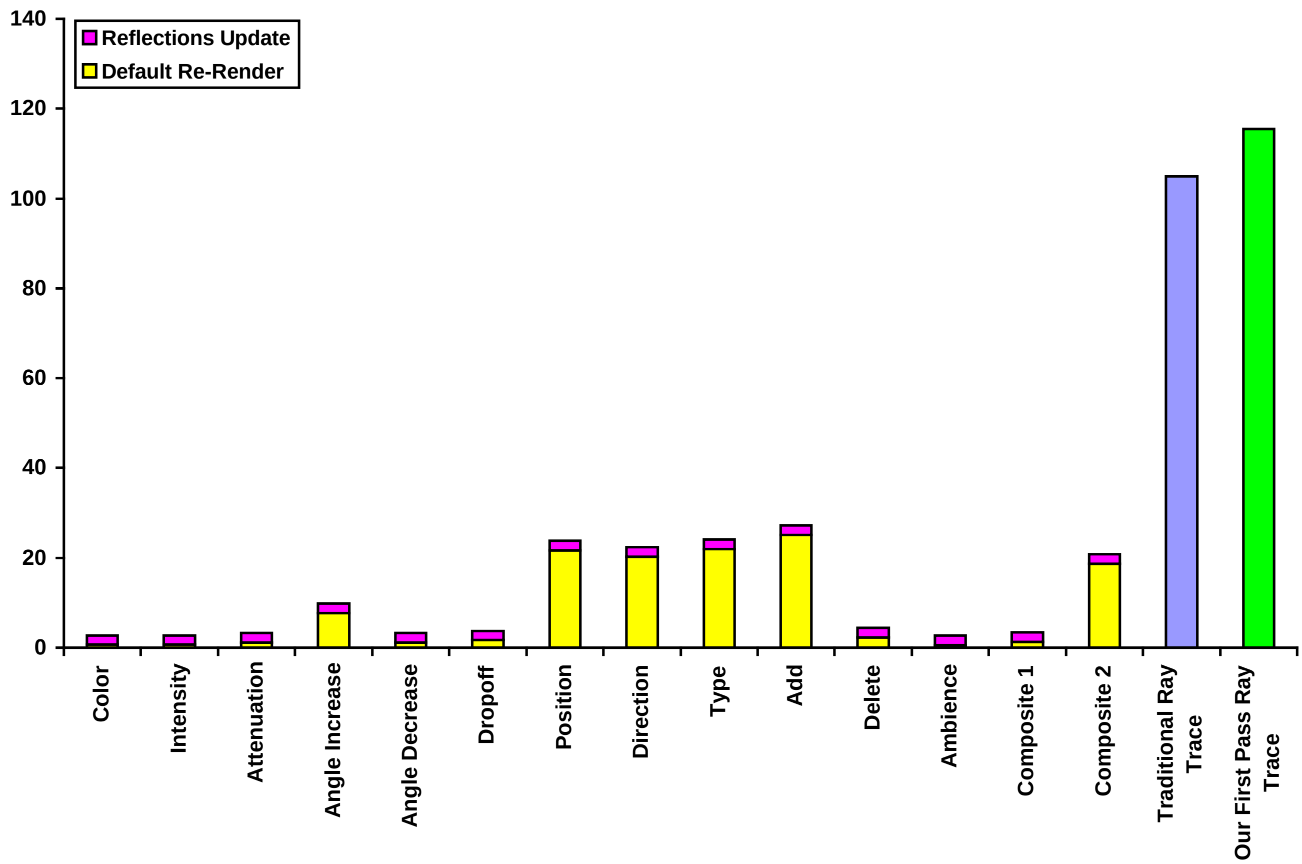

Type of Light Edit

## Example 3: Desk

| | |
|---|---|
| Spheres | 1 |
| Cones | 2 |
| Cylinders | 16 |
| Planes | 44 |
| Point Lights | 2 |
| Spot Lights | 1 |
| Directional Lights | 1 |



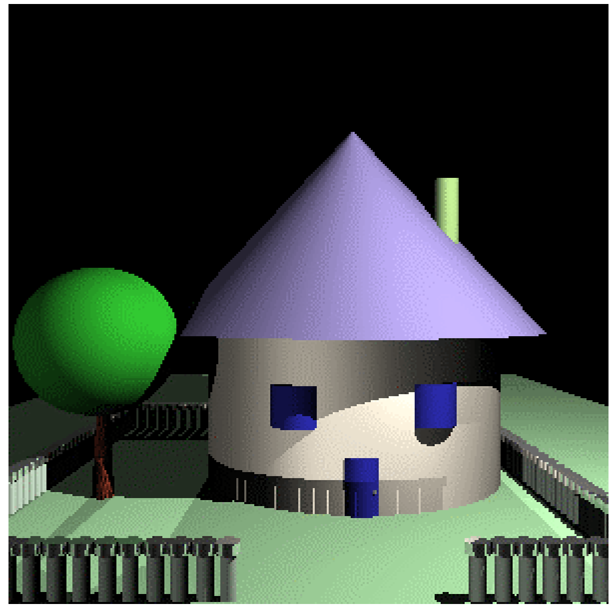
Rendering Time (sec)



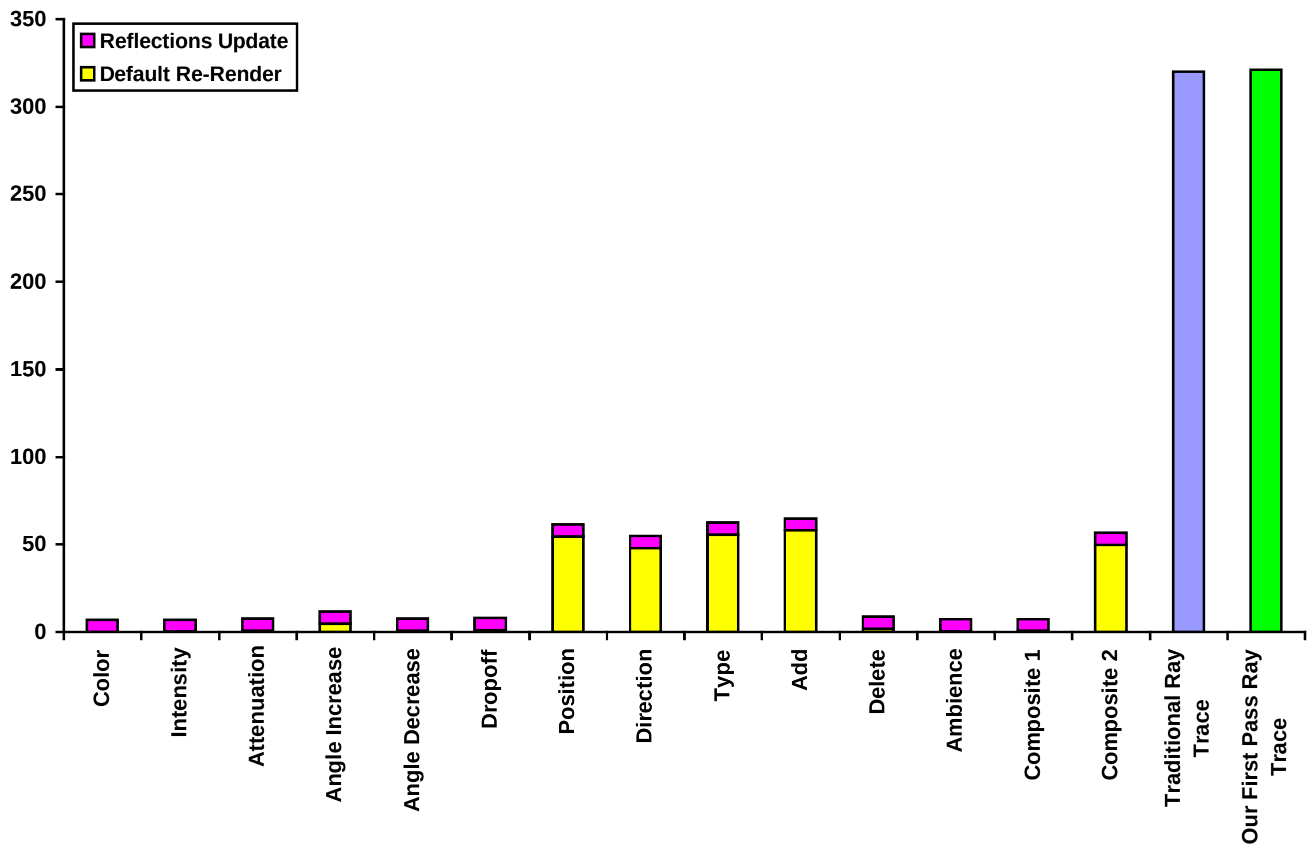

Type of Light Edit

## Example 4: House

| | |
|---|---|
| Spheres | 1 |
| Cones | 2 |
| Cylinders | 266 |
| Planes | 1 |
| Point Lights | 1 |
| Spot Lights | 1 |
| Directional Lights | 2 |



Rendering Time (sec)




Type of Light Edit

## Acknowledgements:

Special thanks to my advisor John Hughes for not only helping with the design and implementation of this project, but for inspiring its name one day by describing my ray trace updates as “lightning fast.”  Also many thanks to Jeff White and my parents for all their help and support. Thanks to Loring Holden for enabling Brown University's Sketch program to model .jen scenes [11].

## References:


[1] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. In *CG&A*, 7(5), pp. 14-20, May 1987.

[2] A. S. Glassner. Space Subdivision for Fast Ray Tracing. In *IEEE Computer Graphics and Applications* vol. 4, no. 10, pp. 15-22, Oct. 1984.

[3] J. Eyles, S. Molnar, J. Poulton, T. Greer, A. Lastra, N. England, and L. Westover. Pixel Flow: The Realization. In *Proceedings of the 1997 Siggraph/Eurographics Workshop on Graphics Hardware*, pp. 57-68, Aug. 1997.

[4] N. Brière and P. Poulin. Hierarchical View-dependent Structures for Interactive Scene Manipulation. In *Proceedings of SIGGRAPH '96*, pp. 83-90, Aug.1996.

[5] C. Séquin and E. Smyrl. Parameterized Ray Tracing. In *Proceedings of SIGGRAPH '89*, pp. 307-314, July 1989.

[6] A. Appel. Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the Spring Joint Computer Conference*, pp. 37-45, 1968.

[7] J. Foley, A. van Dam, S. Feiner, and J. F. Hughes. *Computer Graphics, Principles and Practice*. Addison-Wesley, Reading, MA, 1996.

[8] A. S. Glassner. *An Introduction to Ray Tracing.* Academic Press, London, 1989.

[9] http://www.cm.cf.ac.uk/Ray.Tracing/ (otherwise known as "The Mother of All Ray Tracing Pages")

[10] H. Sowizral, K. Rushforth, and M. Deering. *The Java3D API Specification.* Addison-Wesley, Reading, MA, 1998.

[11] R. Zeleznik, K. Herndon, and J.F. Hughes. SKETCH: An Interface for Sketching 3D Scenes. In *Proceedings of SIGGRAPH '96*, pp. 163-170, Aug. 1996.

[12] G. Drettakis and E. Fiume. A Fast Shadow Algorithm for Area Light Sources Using Back Projection. In *Proceedings of SIGGRAPH '94*, pp. 223-230, July 1994.

[13] A. J. Stewart and S. Ghali. Fast Computation of Shadow Boundaries Using Spatial Coherence and Backprojections. In *Proceedings of SIGGRAPH '94*, pp. 231-238, July 1994.

[14] G. Elber. Low Cost Illumination Computation Using an Approximation of Light Wavefronts. In *Proceedings of SIGGRAPH '94*, pp. 335-342, July 1994.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[16] H. Fuchs, J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F. Brooks Jr., J. Eyeles, and J. Poulton. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. In *Proceedings of SIGGRAPH '85*, pp. 111-120, July 1985.

[17] R. Cook, L. Carpenter, and E. Catmull. The Reyes Image Rendering Architecture. In *Proceedings of SIGGRAPH '87*, pp. 95-102, July 1987.