NPR Light Implementation Notes

Janet Davis

March 16, 2000

1 Introduction

NPR Light displays models using non-photorealistic rendering techniques, including tone-based shading and edge outlines. In addition, models may include metal-shaded or stippled surfaces. The implementation uses OpenGL for the rendering engine and has a simple GLUT-based graphical user interface. Models are defined in plain text files.

2 Rendering

A model is rendered in up to four passes. In the first pass, objects are shaded with their base colors and the warm and cool tones that indicate surface normal direction. In the second pass, edges are drawn. The scene is then loaded into the accumulation buffer with a factor of 0.5. The color and depth buffers are cleared, and specular highlights are rendered. The edges are rendered again to ensure that they are fully visible in the accumulated scene, and the scene is loaded into the accumulation buffer with a factor of 0.5. The scene is then returned to the frame buffer and displayed. Each pass is encapsulated in a display list to increase rendering speed.

2.1 Shading

Gooch, et. al. present a shading model in which tone is used to indicate surface normals. Surfaces that face the single light are shaded with a warm color such as yellow; surfaces that face away from the light are shaded with a cool color such as blue. This follows the conventions of technical illustration.

If $\hat{n}$ is the surface normal and $\hat{I}$ is the direction to the light, then warm and cool colors C_{warm} and C_{cool} are blended according to the angle between the surface normal and the light direction:

$$C = \left(\frac{1 + \hat{I} \cdot \hat{n}}{2}\right) C_{cool} + \left(1 - \frac{1 + \hat{I} \cdot \hat{n}}{2}\right) C_{warm}$$

The colors C_{warm} and C_{cool} are calculated using several factors. The colors C_{blue} and C_{yellow} are saturated blue and yellow. Each material has a base color C_{base}. Global parameters b, y, α, and β modulate these colors. The result is a linear blend between blue/yellow and black/object-color tones:

$$C_{cool} = b \cdot C_{blue} + \alpha C_{base}$$
$$C_{warm} = y \cdot C_{yellow} + \beta C_{base}$$

I implemented Rademacher's shading model, which is similar but not identical. The most significant difference is that shading is controlled with per-object rather than global parameters. The model is as follows:

$$\begin{aligned} C = k_a * C_{base} \quad &+ \quad (\hat{I} \cdot \hat{n})(C_{yellow} - (C_{base} \cdot k_1) \cdot k_2 \\ &+ \quad ((-\hat{I}) \cdot \hat{n})(C_{blue} - (C_{base} \cdot k_3) \cdot k_4 \end{aligned}$$

All values, except for C_{yellow} and C_{blue}, are per-object parameters. This model can be simulated in OpenGL as follows. The ambient light has color $k_a * C_{base}$. Two lights are created, one at the actual light position and one at that position reflected across the origin. The first light has diffuse color $(C_{yellow} - (C_{base} \cdot k_1) \cdot k_2$, for the warm shading. The second light has diffuse color $(C_{blue} - (C_{base} \cdot k_3) \cdot k_4$, for the cool shading. The ambient and specular terms for both lights are set to zero. The object can then be rendered with white ambient and diffuse material colors to get the correct lighting.

The default value for k_a is 0.9, while k_1 through k_4 have a default of 1.0. All of these parameters, as well as the base color, can be set by the user. I used $C_{blue} = (0, 0, 0.4)$ and $C_{yellow} = (0.4, 0.4, 0)$. The diffuse colors of the two lights are also scaled down by 1/3 in order to produce a more reasonable result. The light position is above and to the right, at $(5, 5, 2)$.

2.2 Specular highlights

Because the combination of specular highlights with negative diffuse lights may result in some jarring artifacts, specular highlights are rendered in a separate pass. Ambient and diffuse lights are disabled entirely, as is the specular component of the light at the reflected position. The light at the real light position is given a white specular component. All objects have black ambient and diffuse material properties. Their specular term is a shade of gray specified by the user. Their shininess can be specified by the user as well.

2.3 Edges

Object silhouettes are drawn with the first and simplest technique described by Raskar and Cohen. Front face culling is enabled, the depth function is set to `GL_LEQUAL`, and polygons are translated slightly forward using `glPolygonOffset()`. The entire scene is re-rendered under these conditions, giving silhouette edges with fairly even thickness.

2.4 Metals

Gooch et. al. observe that, in technical illustrations, metal objects are shaded as if very anisotropic. They produce this effect using a striped texture map in which stripe centers are luminances uniformly chosen from the interval [0.0,0.5], and the luminances are interpolated linearly between the stripe centers. They use twenty stripes. These stripes are mapped along the axis of minimum curvature, parallel to the milling axis, and the stripe closest to the light source is overwritten with white.

My metal rendering technique basically follows these guidelines. I generate a one-dimensional texture map of 16 32-pixel wide stripes. The stripes are generated as above. The texture is blended equally with the tone-shaded component of the surface along the axis of minimum curvature. I found overwriting the stripe nearest the light with white to be impractical under arbitrary transformations. Using a moderately bright specular color (0.7) and a low shininess (10) gives plausible results for cylinders and flat surfaces. Spheres, however, look very improbable.

Note that a single texture map is generated for all metal-shaded objects, so that it is possible to line up the stripes on adjacent objects and give the impression of continuity.

2.5 Stippled surfaces

I experimented with stippled surfaces. The texture maps for stippling are 256×256 arrays of floating point luminosity values, computed as follows. With probability $1 - threshold$, a pixel is set to 1. Otherwise, the pixel is set to $1 - difference$, if binary stippling is desired, or a value between $1 - difference$ and 1 if variable stippling is desired. The texture is mipmapped using `gluBuild2DMipmaps`. The magnification function is `GL_LINEAR` and the minification function is `GL_LINEAR_MIPMAP_LINEAR`, in order to give smooth transitions as the object is moved. The `GL_PERSPECTIVE_CORRECTION_HINT` is also enabled.

The texture may be used to modulate the tone-based shading, which darkens some pixels, resulting in something that looks like pencil stippling. Or, the texture can be blended equally with the results of shading, so that the shading is lightened for some pixels. With large, white specular highlights, light shading, and a high value of $threshold$ or $difference$, the results resemble colored pencil shading.

The texture mapping function, whether to use a binary or a continuous texture map, and the values of $difference$ and $threshold$ are all supplied by the user for a given stipple material.

3 Models

I developed a simple model specification language. In this language, materials must be declared and named before they are used. The user can specify values for the base color with `red`, `green`, and `blue`. The user may also specify `ka`, `k1`, `k2`, `k3`, `k4`, `shininess`, and `specular`. If any of these values are not specified, reasonable defaults are provided. A metal or stippled material may be generated using the `Metal` or `Stipple` keyword rather than the `Material` keyword. For stippled materials, the user may optionally specify `threshold`, `difference`, `vary_difference`, and `stipple_function`.

The language allows the hierarchical application of `rotate`, `translate`, and `scale` transformations. An object may be included at any point in the hierarchy. Objects include spheres, cylinders, cones, tori, helices, cubes, and GLUT teapots. (Arbitrary polygon meshes could, of course, be added to this list.) The user may specify a material and a number of geometric parameters for each object. Texture coordinates will be generated for all but the cube. Most objects are drawn using GLU or GLUT; helices and tori are drawn using GLE, the GL extrusion library, which is distributed with GLUT 3.6 and up.

The objects have parameters as follows:

`Sphere`	`radius`
`Cube, Teapot`	`size`
`Cone`	`radius, height, capped`
`Cylinder`	`radius` or `radius1` and `radius2, height, capped`
`OpenCylinder`	`radius` or `radius1` and `radius2, height, thickness`
`Torus`	`innerRadius, outerRadius`
`Helicoid`	`rToroid, startRadius, drdTheta, startZ, dzdTheta, startTheta, sweepTheta`

The language is case-insensitive.

Sample scenes are available, and the lex and yacc specifications are included with the source code.

4 The User Interface

The user starts `nprlight` by giving the command

```
nprlight <model-name>
```

The model is initially displayed with only shading enabled. However, shading, edge lines, and specular highlights can all be toggled using a right-click pop-up menu. The model can be rotated by left-dragging the mouse.