Assignment P2 : RAY CASTING

Computer Graphics 2 (15-463)

Due 12 March 1996 by end of day

For this assignment you will implement a ray casting program for spheres and polygons that uses hierarchical bounding volumes as an optimization. Ray casting is one-level (non-recursive) ray tracing: at each pixel you determine what surface point is visible and shade it.

Basics

Your program will have the following steps:

1. Read a 3-D scene file. We will provide the scene files and a routine, `scene_read`, to parse scene files. This routine will create data structures describing the hierarchy of spheres and polygons with materials, lights, and bounding volumes, and the 4×4 viewing matrix.
2. Loop over output pixels, for each generating a ray in world space.
3. Test each ray for intersection against spheres and polygons in the scene, using bounding volumes as an optimization, and determine the nearest intersection point (if any).
4. If the ray hits a surface, shade the intersection point using Phong's illumination formula (given below). Otherwise, if the ray hits nothing, use the background color. The resulting RGB color is used as the pixel value.
5. Write pixels to a TIFF picture file.

Thus, the input of your ray caster is a scene file and the output is a TIFF picture file. You can view the picture with programs such as our own `xplay`, or standard programs such as `xv`, `xview`, `xloadimage`, `display`[1], or `imgview`[2].

The source code we provide is in *classdir*`/pub/src/p2`. See the `README_P2` file there. We will provide three scene files describing simple scenes in *classdir*`/pub/scene`. For this assigment, you'll need the files `min.sc`, a single polygon and a single sphere illuminated by a single light; `table.sc`, a dodecahedron and sphere on a table; and `room.sc`, a room containing a fractal thing. See *classdir*`/www/assts/scene.ps` for documentation on scene file format (also available via the Web).

Programming Tips

These files already have camera transformations built into them. If you want to view the scene from different angles, you can split the scene file into two parts as has been done in `min.cam.sc` and `min.nocam.sc`, and modify just the former. To support this, it is suggested that your `raycast` program accept multiple scene files as arguments, e.g. `raycast min.sc` or `raycast min.cam.sc min.nocam.sc`, calling `scene = scene_read(scenefilename, scene)` on each file. On the first call, pass in `scene==NULL`, but on subsequent calls to `scene_read`, pass along the scene pointer returned from the previous call.

After calling `scene_read`, the matrix that transforms world space to screen space (a.k.a. the "viewing matrix") is available in `scene->worldtoscreen`. The requested width and height of the output picture

[1] on SGI's, in `/usr/local/magick/bin`
[2] on SGI's only

are available in `scene->width` and `scene->height`. See the source files `scene.h` and `scene.c` for more details.

You'll need to have hierarchical bounding volumes working in order to generate good pictures of `room.sc`, since it contains 2,457 polygons and spheres, and it would probably take hours to generate a picture without them. The `scene_read` function creates a hierarchy of `GROUP`, `SPHERE`, and `POLYGON` objects, where the `GROUP` nodes contain objects between matching `push` and `pop` commands from the scene file. It computes a bounding sphere (`bsphere`) and bounding box (`bbox`) for every `GROUP` node. (This way of doing hierarchical bounding volumes, where they are derived from the transformation hierarchy, is not always the best way to do things, but it's simple). You can do bounding volume testing of a ray against a `GROUP` using either of these volumes (`bsphere` is probably less code, but perhaps slower). As you trace rays, calculate the average number of ray-surface intersection tests done (include spheres, polygons, and bounding volume tests). You may need to put some time into low-level optimization of your ray-sphere, ray-polygon, and ray-bound intersection code in order to generate pictures of `room.sc` in reasonable time.

In assignment P3, you will extend your program to do shadows, recursive ray tracing (good for chrome & glass) and antialiasing. Since these modifications imply even more calls to your intersection routines, you'll need these to be fast for later, too.

It is suggested that steps 3 & 4 above be modularized into a routine that takes a ray as argument and returns a color, e.g.

```
trace(Ray *ray, Color *col)     /* ray: input; col: returned */
```

Shading

The shading formula used[3] should be Phong's illumination model[4]:

$$\mathbf{I} = \mathbf{C}_{mat} k_{dr} \left[\mathbf{I}_a + \sum_i \mathbf{I}_i \max\{(N \cdot L_i), 0\} \right] + \sum_i \mathbf{I}_i k_{sr} \max\{(N \cdot H_i), 0\}^e$$

where the summation is over all light sources, boldface denotes an RGB triple or color[5], italic capital denotes a 3-D direction vector, lower case denotes a scalar, and "$\cdot$" denotes dot product. General variables are:

$\mathbf{I}$ is the returned radiance of the ray (a color)
N is the surface normal at the intersection point

Light source parameters are:

$\mathbf{I}_a$ is the radiance of ambient light (a color)
$\mathbf{I}_i$ is the radiance of light source i, (a color)
L_i is the direction of light i from the intersection point, computed as the difference of the light's position and the intersection position
H_i is the halfway vector for Phong's illumination model:
$H_i = (L_i + V)/|L_i + V|$ where V is direction of the viewer,
i.e. V is the opposite of the incident ray direction

[3] See Foley et. al equations (16.14), (16.20), and discussion of halfway vector H on page 731. To simplify, we're assuming light source attenuation factor = 1 and a white specular color.

[4] Note that Phong illumination is not the same thing as Phong shading. The former is a formula for computing diffuse and specular reflection of any geometric surface type; the latter is a technique for interpolating normal vectors across polygons to make them look curved, not faceted.

[5] "Color" is a vague term, but in computer graphics it usually means spectral samples or color coordinates of something in some color space. Sometimes it specifically means RGB spectral samples of radiance (a.k.a. intensity), sometimes it specifically means RGB spectral samples of reflectance.

Surface material parameters are:

$\mathbf{C}_{mat}$ is the reflectance of the intersected surface's material (a color)

k_{dr} is the coefficient of diffuse reflection

k_{sr} is the coefficient of specular reflection

e is related to surface roughness and controls highlight size

Light source parameters and material parameters are available in the structures `Light` and `Material` described in `scene.h`.

Shading Notes

All direction vectors should be unit vectors (make them unit length, if you don't know a priori that they are). The max with zero is a backfacing test. The product of $\mathbf{C}_{mat}$ with the radiance quantity $[\mathbf{I}_a + \cdots]$ is a product of two 3-vectors. This product should be done componentwise, e.g. multiply the reds to get the red component, the greens to get the green component, etc. Typically, $k_{dr} + k_{sr}$ and each component of $\mathbf{C}_{mat}$ lie between 0 and 1. If the computed radiance is $\mathbf{I} = (I_r, I_g, I_b)$ then your final pixel values should be $(255 * I_r, 255 * I_g, 255 * I_b)$, but since the radiances ($\mathbf{I}_i$ and $\mathbf{I}$) are not necessarily bounded, you should check each component of the pixel value for overflow, pegging at 255.

What to Submit.

Submit in your `p2` directory:

1. Source code to your `raycast` program, (called "`raycast`", please).
2. A working executable of your `raycast` program that we will be able to run (executable on either an SGI or Sun4).
3. TIFF picture files for `min.sc`, `table.sc`, and `room.sc` created by your raycaster. (Call them `min.tiff`, `table.tiff`, `room.tiff`).
4. A new scene, perhaps a modified version of `table.sc` where you have changed the camera and some of the objects (positions, colors, and lights). You could use the `mcmod` modeler, if you like, and create something totally new. Call the new scene file `new.sc` and create a picture of it, `new.tiff`.
5. A text file, `README`, where you summarize briefly what you've done, and what each of the files in your directory is. Tell us how to run your program on `min.sc` to regenerate `min.tiff`. Also tell us, for `room.sc`, the type of machine you were running on (be specific), how long it took to generate `room.tiff`, and the average number of surface intersection tests per ray.

Notes: feel free to modify the data structures in `scene.h`. You will probably want to pre-compute various information (plane equations, etc) for polygons, for instance. You can also modify `scene.c` and `matrix.[ch]` if you like. Optimization is optional, not required; we're not grading on speed. Reference pictures are provided in *classdir*`/pub/pix/p2`. You will not be penalized for minor differences in shading.

11pm 27 Feb. 96