Specular lighting

Specular light is light which is coherently reflected without scattering. The best example of an object with no ambient or diffuse reflectivity but high specular reflectivity is a mirror.

When you look into a mirror, what you see is the reflection of light that has previously been reflected or emitted by other objects. We can create a pure mirror by defining an fplane having a material with diffuse and ambient = {0, 0, 0} and specular = 1.0.

Therefore in a raytracing system, if a ray hits an object with a non-zero specular reflectivity it is necessary to reflect or bounce the ray to see what it hits next. If that object also has a non-zero specular reflectivity it is necessary to bounce the ray again.

This process continues until the bounced ray:

- hits no object
- hits an object with no specular reflectivity.
- travels so far that the effect of further bounces is negligible
Reflecting a ray

Basic physics says: the angle of incidence (the angle the incoming ray makes with the normal at the hitpoint) is equal to the angle of reflection.

```c
vec_reflect(
    vec_t *unitin,          /* unit vector in incoming direction of the ray */
    vec_t *unitnorm,        /* outward surface normal */
    vec_t *unitout);        /* unit vector in outgoing direction reflected ray */
```

Let

\[ U = -unitin \]
\[ N = unitnorm \]

Then

\[ U + V = 2 N \cos(T) \]
where \( T \) is the angle between \( U \) and \( N \)

\[ \cos(T) = U \cdot N \]

so

\[ U + V = 2 N (U \cdot N) \]

and

\[ V = 2 N (U \cdot N) - U \]
The updated raytrace function:

```c
void ray_trace(
    model_t *model,
    vec_t *base,        /* location of viewer or previous hit */
    vec_t *dir,         /* unit vector in direction of object */
    drgb_t *pix,        /* pixel return location */
    double total_dist,  /* distance ray has traveled so far */
    object_t *last_hit) /* most recently hit object */
{
    object_t *closest;
    double    specref = 0.0;
    double    mindist;
    drgb_t    thispix = {0.0, 0.0, 0.0};

    if (total_dist > MAX_DIST)
        return;

    Find the closest object that the ray hits, and if there is a hit:
    Add the distance from base of the ray to the hit point to total_dist
    and do ambient and diffuse lighting as before and divide by total_dist

    closest->getspecular(&specref);      /* see if object has specular reflectivity */
    if (specref is not 0)
    {
        drgb_t specint = {0.0, 0.0, 0.0};
        compute direction, ref_dir, of the reflected ray.
        ray_trace(model, closest->last_hit, ref_dir, specint, total_dist, closest);
        scale specint by specref
        add specint to thispix
    }
    add thispix to pix
}
```

The hitloc lies on the surface of closest. We can't allow another hit at distance 0.
Specular glints

Glints are another form of specular reflection. A glint occurs when a specific light is reflected from the surface to the viewpoint (or source of a reflected ray). In the following example we see glints produced by a greenish light in the lower left and a redish light at the upper right.

The center of the glint is the location at which a ray arriving from the light is reflected about the normal directly into the eye of the viewer. We will provide a model that allows us to control how tightly the reflection is focused.
Modifications to ray tracing data structures

The only modification to the classes that is necessary is the addition of the shininess exponent to the material_t. It is a single double precision value and must be added to the parser and printer of the material_t. If the shininess value is not specified it should be set to 0.0. The material_getshine() function retrieves the shininess value. While we are at it, we also add a transparency attribute that we will describe later.

class material_t
{
    friend material_t *material_getbyname(model_t *, char *);

public:
    material_t(){};
    material_t(FILE *in, model_t *model, int attrmax);
    void getambient(drgb_t *dest);
    void getdiffuse(drgb_t *dest);
    void getspecular(double *spec);
    void getshine(double *shiny);
    void gettrans(double *trans);
    char *material_getname();
    void material_print(FILE *out);

private:
    int cookie;
    char name[NAME_LEN];
    drgb_t ambient;          /* Reflectivity for materials */
    drgb_t diffuse;
    double specular;
    double shininess;
    double transparency;
};
A sample input file

The input file used to produce the blue sphere with the two glints is shown below. The *larger* the value of *shininess* the *more focused (smaller)* the glint will be. Visually realistic values tend to be large.

camera cam1
  {  
    pixeldim 800 600  
    worlddim 8 6  
    viewpoint 4 3 4  
  }

material blue
  {  
    diffuse 0 0 8  
    specular 1.0  
    shininess 50.0  
  }

light pinkfront
  {  
    emissivity 6 5 5  
    location 10 8 4  
  }

light cyanfront
  {  
    emissivity 4 6 5  
    location -2 0 4  
  }

sphere shadowmaker
  {  
    material blue  
    center 4 3 -6  
    radius 4  
  }
The model for producing glints

The model is closely related to the one underlying specular reflection. Because the angle of incidence is equal to the angle of reflection, an incoming light ray from light 1 will be directed toward the view point. In contrast, an incoming ray from light 2 will be reflected on the other side of the surface normal from the ViewPt.

Notice that if the light is reflected directly toward the view point, then the sum of the vector pointing toward the light and the vector pointing toward the viewpoint (more precisely the source of the ray!) is a vector that is perfectly aligned with the surface normal.
Therefore the algorithm works as follows:

```c
void light_t::add_glint(
object_t *hitobj,
vec_t    *base,      // base of original ray
vec_t    *dir,       // unit vector in direction to light
drgb_t   *pixel)
```

- If the shininess of the surface is 0 return.
- Compute a unit vector from the hitpoint to the light.
- Compute a unit vector from the hitpoint to the source of the ray.
- Take the sum of these two vectors and convert the sum to a unit vector.
- Compute \( \text{dot} \) = the dot product of the unit sum with the unit normal at the hitpoint. This is our base measure of how close the line to the viewpoint is to the actual direction the light is reflected.
- Raise \( \text{dot} \) to the power of shininess. Since the dot product is \(\leq 1.0\), raising it to a large power will tend to reduce it. This is why a large value of shininess produces a more focused glint.
- Scale the emissivity of the light by \( \text{dot} \) times the specular reflectivity of the hit object's material.
- Add the scaled value to \( \text{pixel} \).

Technically speaking we should also scale by \(1/(\text{distance to light})\). But that tends to excessively reduce the visibility of the glint.
Where does the glint code go?

The *glint* effect must be computed for *every hitpoint and every light*. This is also true of diffuse illumination. Thus, the most reasonable place to put it is:

- in a new method that is invoked near the end of illuminate `illuminate()`.

```c
pix_sum(pixel, diffuse, pixel);
add_glint(hitobj, base, dir, pixel);
return;
}
```

How do we access the *base of the ray*??

We must pass it from raytrace, through add diffuse to illuminate :-(
Antialiasing with sub-pixel sampling.

Aliasing is an effect in which edges that should appear smooth actually appear jagged because of the finite size of pixels. One approach to anti-aliasing is to artificially induce an intensity gradient near any edge. One way to do this is via random sub-pixel sampling.

In this approach, the world coordinate space is partitioned into a collection of non-overlapping squares in which the actual pixel associated at the square is located at the center. Multiple rays are fired at each pixel with the direction of the ray randomly jittered in a way that ensures in passes through the proper square. The value actually stored in the image is the average of all the pixel values computed.

Magnified view without antialiasing

![Magnified view without antialiasing](image1)

Magnified view with 16 pixel averaging

![Magnified view with 16 pixel averaging](image2)
The easiest way to jitter the direction of the ray is to jitter the coordinates of the pixel through which it passes. This is easy to do.

```c
void camera_t::getdir(int x, int y, vec_t *dir)
{
    vec_t world;
    double dx = x;
    double dy = y;

    if (AA_SAMPLES > 1)
    {
        dx = randomize(dx);
        dy = randomize(dy);
    }

    /* Create world coordinates using (dx, dy) */
```

The `randomize()` function

Most systems provide library functions that generate streams of pseudo random numbers. The `random()` function is the one we will use. It returns an integer value between 0 and 0x7fffffff. To perform pixel randomization the value must be converted to a `double` in the range [-0.5, 0.5] which is then added to the input pixel coordinate. This can be done by:

- converting the random integer to a `double`
- dividing the `double` by 0x7fffffff
- subtracting 0.5 from the `double`.

In the transformations, do not use the symbol `RAND_MAX`. 
**Modifications to `image_create()`**

The `make_pixel` function must generate AA_SAMPLES directions and call `ray_trace()` AA_SAMPLES times. It is necessary that `ray_trace()` add to and not set the contents of the pixel pointer passed to it. After all the calls to `ray_trace()` have been made, the final pixel value must be scaled by 1.0 / AA_SAMPLES.

```c
static inline void make_pixel(model_t *model, int x, int y) {
    vec_t raydir;
    vec_t viewpt;
    drgb_t pix = {0.0, 0.0, 0.0};
    camera_t *cam = model->cam;
    int i;

    cam->getviewpt(&viewpt);
    for (i = 0; i < AA_SAMPLES; i++) {
        cam->camera_getdir(x, y, &raydir);
        ray_trace(model, &viewpt, &raydir, &pix, 0.0, NULL);
    }

    pix_scale(1.0 / AA_SAMPLES, &pix, &pix);
    cam->store_pixel(x, y, &pix);
    return;
}
```
Partial transparency

Partial transparency of objects is somewhat less easy to implement than specular reflection. We will be confronted by two aspects of the problem:

- allowing objects that are normally obscured because they are located behind other objects to become visible.
- allowing lights that would normally be occluded to provide partial illumination of the intervening objects are all partially transparent.

One could argue that if we were to do specular reflection correctly then it would be even more difficult than partial transparency. Recall that we did not deal with the possibility of light being reflected from a shiny surface illuminating an object, but in the real world it surely does that.

An optional transparency factor is added to the `material_t`. Its default value is 0.0 which means not transparent at all.

```cpp
private:
    int     cookie;
    char    name[NAME_LEN];
    drgb_t  ambient;       /* Reflectivity for materials */
    drgb_t  diffuse;
    drgb_t  specular;
    double  shininess;
    double  transparency; /* 0.0 -> opaque : 1.0 invisible */
};
```
In the above image page this is the material definition for the gold vases and the see through sphere

```plaintext
material gold
{
  ambient 1 1 0
  diffuse 4 4 0
  transparency 0.2
  specular 0.4
}

material see_thru
{
  diffuse 4 4 4
  transparency 0.6
}
```
**Required modifications**

**Modifications to material_t**

The following modifications to the `material_t` class are needed to support transparency

- Then material constructor must parse transparency attribute
- New `material_t::material_gettrans()` and `object_t::gettrans()` methods

**Modifications to ray_trace**

The following modifications to the `ray_trace()` function are also required. This code should be inserted immediately AFTER `add_illumination()` is called and `thispix` is scaled by `1 / total_dist`.

```cpp
closest->gettransparency(&trans); /* see if object has transparency */
if (trans is not 0)
{
    drgb_t transint = {0.0, 0.0, 0.0};  // intensity of pass thru ray.
    drgb_t diffcolor = {0.0, 0.0, 0.0};  // diffuse color of transparent object
    call ray_trace recursively keeping the same ray direction but letting the new base of the ray be the current hit point.
    scale thispix by (1.0 - trans)
    scale transint by trans
    compute diffcolor and maxpix;  // as in light_t::illuminate
    scale diffcolor by trans / maxpix
    multiply (component-wise) transint by scaled diffcolor
    add transint to thispix
}
```
Modifications to light_f::illuminate()

The tests for occlusion must be modified in a fairly significant way. It is potentially necessary to identify all of the objects that lie upon the path from the hit point of the object being illuminated back to the light. For each object that lies on the path and is partially transparent it is necessary to modify the emissivity of the light in a way that reduces its intensity and may change its color. A correct solution should work something like this:

```cpp
getemiss(emiss);  // make copy of emissivity that we can change
baseobj = hitobj; // starting point for search
workdist = dist;  // make copy of dist we can change

while (1)
{
    drgb_t *diffcolor; // diffuse color of object in the way
    double maxpix;     // maximum value of diffcolor[] array.
    double trans;      // transparency of intervening object

    find closest object along the path
    if there isn't one or it's beyond the light
        break;

    compute trans;
    if (trans == 0)
        return;

    /* Light is occluded by partially transparent object */
    compute diffcolor and maxpix;
    scale diffcolor by trans / maxpix
    multiply (component-wise) emiss by scaled diffcolor

    update baseobj
    update workdist
}
```
A single partially transparent finite plane

camera cam1
{
  pixeldim 640 480
  worlddim 8 6
  viewpoint 4 3 8
}

material white
{
  diffuse 16 16 16
}

material green
{
  diffuse 0 8 2
  transparency 0.6
}

plane backwall
{
  material white
  normal 0 0 1
  point 0 0 -8
}

plane middle
{
  material green
  normal 0 0 1
  point 2 1.5 -5
  xdir 1 0 0
  dimensions 4 3
}

light left
{
  location 0 3 3
  emissivity 5 5 5
}

light right
{
  location 8 3 3
  emissivity 5 5 5
}
The resulting image

The gray background is the backwall as illuminated by the two lights with neither light passing through the finite plane. The light green area on the left side of the picture is produced by rays that do not hit the fplane but for which the light on the right is occluded by the transparent fplane. The bright green rectangle with the dark stripe in the middle is produced by rays that pass through the transparent fplane and then strike the back wall. The dark stripe in the center is covers pixels on the back wall for which the finite plane occludes both the left and right lights. The thin even darker green strips above and below the fplane are backwall pixels created by rays that miss the fplane but for which both lights are occluded by the fplane.

<table>
<thead>
<tr>
<th>Region</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ray passes through fplane</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>Fplane occludes right side light</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Fplane occludes left side light</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>
An example with two overlapping partially transparent objects

camera cam1
{
  pixeldim 640 480
  worlddim 8 6
  viewpoint 4 3 8
}

material white
{
  diffuse 16 16 16
  ambient 0 0 0
}

material cyan
{
  diffuse 0 8 8
  transparency 0.6
}

material yellow
{
  diffuse 8 8 0
  transparency 0.6
}

plane backwall
{
  material white
  normal 0 0 1
  point 0 0 -8
}

fplane middle
{
  material cyan
  normal 0 0 1
  point 3 1.5 -5.05
  xdir 1 0 0
  dimensions 4 3
}

fplane middle2
{
  material yellow
  normal 0 0 1
  point 1 1.5 -5
  dimensions 4 3
}

light middle
{
  location 4 3 3
  emissivity 10 10 10
}
Here the yellow fplane is on the left and the cyan fplane is on the right. The overlapping area is green as it should be. The darker edges that comprise the border are pixels for which rays do not pass through the fplanes, but for which the fplanes do occlude the single centrally located light.