ANTI-ALIASING TO SMOOTH OUT EDGES

Unit 18 notes, pages 11-12

1. code is given in notes for addition to camera::getdir
2. the “randomize” function can be added in camera::getdir as well, where:
   a. it could take the pixel coordinate in as an argument
   b. the function random( ) is called (that function returns a long, which is an int)
   c. convert that number that gets returned to a double
   d. divide that double by the hex # in the notes
   e. then subtract .5 from it
   f. return (that number + the pixel coordinate)
3. code is given in notes for make_pixel function in image.c
4. also, don’t forget to add that line specified in the makefile: CFLAGS = -DAA_SAMPLES=1
   • you can change that to other numbers, but the larger the number, the longer it will take for the program to run, so try smaller numbers, like 7-10
5. I think with AA_SAMPLES in the makefile, that means take it out of ray.h. (try commenting it out in ray.h to see if it works)
**SPECULAR LIGHTING**

Unit 18 notes, pages 1-3 (glints continue on through page 9)

1. make sure in ray.h that material_t has a private data member called specular

2. also, notice in ray.h that material_t has a class method called getspecular - so that will need to be implemented in material.cpp (I think it's already be there? Check and make sure.)
   a. material::material_getspecular works similarly to material_getambient where the argument sent in will be assigned the value of specular
   b. there is also object_t::getspecular() method implemented in object.cpp

3. from the notes, page 3 - updating the ray_trace() function:

   ```
   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, mindist to total_dist
   and do ambient and diffuse lighting as before and divide by total_dist
   
   in other words: (nothing really new here - this is the same that we have had in ray_trace)
   ```
   
   ```
   call to find_closest_object (sending last_hit in as 4th argument)
   check to see if the object that is returned is null
   if it is, return
   add mindist to total_dist
   getambient
   add_illuminate
   pix_scale
   
   closest->getspecular(&specref); /* see if object has specular reflectivity */
   /* “closest” is the object returned by find_closest_object - many of you called it “obj” instead of “closest” */
   
   if (specref is not 0)
       |
   |    drgb_t specint = {0.0, 0.0, 0.0};
   |    /* declare “ref_dir”, of type of vec.t */
   |    /* compute direction, ref_dir, of the reflected ray. * vec.reflect( ) function call */
   |    ray_trace(model, closest->last_hit, ref_dir, specint, total_dist, closest);
   |    scale specint by specref /* pix.prod( ) */
   |    add specint to thispix /* pix.sum( ) */
   |
   |
   add thispix to pix /* the same pix.sum( ) that we have had at the end of ray_trace*/
   ```
**ADDING GLINTS**

Unit 18 notes, pages 4-9

1. make sure in ray.h that material_t has a private data member called shininess

2. also, notice in ray.h that material_t has a class method called getshine - so that will need to be implemented in material.cpp (I think it's already be there? Check and make sure.)
   a. material::material_getshine works similarly to the other getter methods where the argument sent in will be assigned the value of shininess

3. parsing of the shininess attribute will need to be taken care of

4. the notes suggest adding code for glints in a new function called add_glint that is invoked near the end of illuminate()

5. or you could add code right in the illuminate() function at the bottom.

6. either way, following the pseudo code on page 8 of the notes:
   a. you’ll need a variable declared to hold the shininess value from the hit object; if it’s zero, return
   b. compute a unit vector from the hitpoint to the light (probably already done towards the top of the illuminate function, some students called it dir)
   c. compute a unit vector from the hitpoint to the source of the ray (viewpoint)
      • base (one of the arguments) is the viewpoint; the object has the hit point
      • then, vec_diff to get the vector
      • then, vec_unit to unitize that vector
   d. vec_sum for both of those vectors
   e. vec_unit to unitize that sum
   f. compute dot = the dot product of the unit sum with the unit normal at that hitpoint. this is the base measure of how close the line to the viewpoint is to the actual direction the light is reflected.
   g. raise dot to the power of shininess - the variable from step a. above - and save result into the shininess variable. you can use the pow( ) function to raise something to a power (part of the math library).
   h. scale the emissivity of the light by dot times the specular reflectivity of the hit object’s material
      • use getspecular (a material_t method) to get the specular value from the hitobj - store into a variable
      • scale emissivity by (dot time specular)
   i. add the scaled value to the pixel
**ADDING TRANSPARENCY**

Unit 18 notes, starting on page 13

1. make sure in ray.h that material_t has a private data member called transparency (similar to shininess for glints); it’s default value of 0.0 means opaque = no transparency; 1.0 means invisible = complete transparency

2. also, notice in ray.h that material_t has a class method called gettrans - so that will need to be implemented in material.cpp (I think it’s already be there? Check and make sure.)
   a. material::material_gettrans works similarly to the other getter methods where the argument sent in will be assigned the value of transparency
   b. there is also object_t::gettrans() method implemented in object.cpp

3. parsing of the transparency attribute will need to be taken care of

4. from the notes, page 15 - updating the ray_trace( ) function - the following should go immediately after the call to add_illumination( ) and before the specular code:

```cpp
BEGIN RED

closest ->gettrans(&trans);
/* see if object has transparency */

if (trans is not 0)
{

drgb_t transint = {0.0, 0.0, 0.0};  
/* declare “ref_dir”, of type of vec_t */
compute direction, ref_dir, of the reflected ray. /* vec_reflect() function call */
ray_trace(model, closest->last_hit, ref_dir, specint, total_dist, closest);
 scale specint by specref /* pix_prod() */
 add specint to thispix /* pix_sum() */
}

add thispix to pix /* the same pix_sum() that we have had at the end of ray_trace*/

END RED
```

```cpp
closest->getspecular(&specref);
if (specref is not 0)
{

drgb_t specint = {0.0, 0.0, 0.0};
/* declare “ref_dir”, of type of vec_t */
compute direction, ref_dir, of the reflected ray. /* vec_reflect() function call */
ray_trace(model, closest->last_hit, ref_dir, specint, total_dist, closest);
 scale specint by specref /* pix_prod() */
 add specint to thispix /* pix_sum() */
}

add thispix to pix /* the same pix_sum() that we have had at the end of ray_trace*/
```

```
if (trans is not 0)
{

drgb_t transint = {0.0, 0.0, 0.0};  
/* diffuse color of transparent object */
pix_scale for scaling thispix by (1.0 - trans)
ray_trace(model, closest->last_hit, dir, &transint, total_dist, closest);
scale transint by trans /* pix_scale() */
compute diffusecolor and maxpix /* as in light_t::illuminate */
scale diffusecolor by (trans / maxpix)
multiply (component-wise) transint by scaled diffusecolor /* pix_prod() */
 add transint to thispix /* pix_sum() */
}
5. from the notes, page 16 - updating the light_t::illuminate() function, it says to do the following:

```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

    pix_copy pixel into diffcolor
    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
}
```
1. The spotlight_t is a derived class of light_t. The class definition looks like:

```cpp
class spotlight_t: public light_t
{
    public:
        spotlight_t(FILE *, model_t *, int);
    virtual void printer(FILE *);
    virtual int vischeck(vec_t hitloc);
    private:
        double theta; // half angle in degrees
        vec_t point; // point the centerline hits
        vec_t dir; // unit vec centerline direction
        double costheta; // cosine of the cone's half angle
};
```

2. There are only 3 methods for the spotlight that need to be implemented. The constructor:
   a. must parse the point and theta parameters
   b. It must compute a unit vector in the centerline direction and store it in dir
      • vec_diff with the location (comes from light_t) of the spotlight and the point (from the spotlight_t)
      • vec_unit to unitize that vector
   c. and convert theta to radians and store its cosine in costheta. The C math library supplies a cos( ) function that can be used. i.e. costheta = cos(theta);
      • you can look up the formula for converting degrees to radians; don't forget - M_PI is a predefined variable that you can use for the value of Pi.

3. The vischeck( ) method of the spotlight_t will need to be implemented. The spotlight will illuminate the hit location if and only if a vector from the location of the spotlight (light) to the hit location lies inside the spot cone. The vischeck( ) method will check for that.
   a. compute a unit vector from the location (comes from light_t) of the spotlight to the hitloc (hitloc is the argument passed into the vischeck( ) method)
      • first vec_diff to find the vector
      • then vec_unit to unitize that vector
   b. take the dot product of this vector with a unit vector in the direction of the centerline of the spotcone i.e. vec_dot with the unitized vector from step (a) with dir from the spotlight_t
   c. if this value is > costheta, the hit location is illuminated
      • return a 1 (or return a 0 depending on how your logic is set up in illuminate( ) where it is called)
   d. otherwise
      • return a 0 (or return a 1 depending on how your logic is set up in illuminate( ) where it is called)

4. If you haven’t implemented the vischeck( ) method in light_t, all it does is returns 0 - sort of like the default handlers in the original object.c file. Because the spotlight_t is derived from the light_t, the spotlight_t vischeck( ) method will override the light_t vischeck( ), if the hitobj is illuminated by a spotlight. The vischeck( ) method is called in the illuminate function right before the find_closest_object( ) call, sending it the hitloc of the hitobj. If vischeck returns a 0, meaning the hitloc is not inside a spotlight, then return.
**ADDING TEXTURE**

Unit 19 notes, pages 1-9

1. A textured plane is a finite plane, so you'll need the `fplane_t` module complete as well (make sure the `fplane` has those two protected data members - on page 3 of notes). If you choose to do a textured plane, you'll have to add a new list for the textures. So, in the model constructor in `model.cpp`, you'll also create a new “texas” list. In the load_entity method, make sure you add the “txplane”. In the print method of `model.cpp`, you'll want to add the line for printing the texplane list along with the `material_`, `object_`, and `light_list_print` function calls.

```cpp
class texplane_t: public fplane_t
{
public:
    texplane_t(FILE *, model_t *, int);
    texplane_t();
    virtual void printer(FILE *);
    virtual void getdiffuse(drgb_t *pix);
    virtual void getambient(drgb_t *pix);
protected:
    int mode;
    texture_t *texture;
private:
    char texname[64];
};
```

2. The `txplane_t` relies on the hits function of the `fplane_t`, but overrides the default `getambient` and `getdiffuse` functions, which will return the values based upon the reflectivities specified by the `material` and the colors of the corresponding pixel in the texture.

3. `txplane_t` constructor:
   a. parse `texname` and `mode` attributes
   b. ask `texture_getbyname` function to return the address of the `texname` texture; this function should work just like the `material_getbyname()` function does except on the texture list instead of the material list

   in material.cpp, we have:

   ```cpp
   material_t *material_getbyname(model_t *model, char *name)
   // which will be texture_t *texture_getbyname(model_t *model, char *name)
   {
       material_t *mat; // you’ll have: texture_t *tex;
       list_t *list = model->mats; // list_t *list = model->texas;
       list->reset(); // same
       while (list->not_end()) // same
       {
           mat = (material_t *)list->get_entity();
           // tex = (texture_t *)list->get_entity();
           if (strcmp(name, mat->material_getname()) == 0) // tex->material_getname
               return(mat); // return(tex);
           list->next_link();
       }
       return NULL;
   }
   ```
   c. set the `object_type` to “txplane”

4. `txplane_t::getdiffuse`
   a. acquire the diffuse reflectivity `mat` of the underlying `object_t` and store into local variable `matdiff`
   b. ask `texture_fit(texel)` or `texture_tile(texel)` to return the value of the texel hit depending on the `mode`
   c. store component-wise product (pix_prod) of `matdiff` & `texel` in `value` (argument sent in)
5. `texture_t::getambient`  
   a. works similarly to the `getdiffuse` from step 4 except using the ambient value of the underlying `object_t`

6. The pointer to the `texture` is of type `texture_t`.

    ```
    class texture_t
    {
        friend texture_t *texture_getbyname(model_t* , char*);

        public:
            texture_t( );
            texture_t(FILE *in, model_t *model, int attrmax); // responsible for processing
            // the ppm header and the
            // irgb_t data
        void texture_fit(double relx, double rely, drgb_t *);
        void texture_tile(double worldx, double worldy, drgb_t *);
        void load_texture(void);

        private:
            void gettexel(int, int, drgb_t *);
            char name[NAME_LEN]; // descriptive name
            char filename[NAME_LEN]; // file name
            double pix_x_size; // pixel size in world coords
            double pix_y_size;
            int xdim; // of the texture
            int ydim;
            irgb_t *imagebuf;
    };
    ```

7. `texture_t` constructor (and `load_texture()` in step d):
   a. read the texture name (e.g. oak)
   b. consume the '{'
   c. parse filename attribute reading the name into the `filename` element
   d. call `load_texture()` to load the texture (can use functions from lab 1)
      - `fopen()` the filename
      - read in the header and extract the xdim and ydim
      - `malloc` the buffer for the `irgb_t` data and save address in `imagebuf`
      - `fread` the `irgb_t` data and verify correct amount read
   e. obtain the dimensions of a pixel in world coordinates from the camera - can add a method to camera, such as:
      ```
      void camera::getpixsize(double *x, double *y)
      {
          compute for the x the dimension in world coordinates
          compute for the y the dimension in world coordinates
      }
      ```
   f. add the texture to the `model->texs` list. (You should have added this list to the model up in step #1.)

8. `texture_fit()` to map hit location to texture location when `mode == 0`
   a. convert relative x (the x component of `newloc / dims[0]`) and y (the y component of `newloc / dims[1]`) coordinates to absolute pixel coordinates by multiplying the x and y pixel dimensions of the texture (xdim, ydim)
      - example: if the texplane is size 4 X 3 in world coordinates (dim[0] is 4, dim[1] is 3) and newloc is (1.0, 1.5, 0.0), the the relative location `newloc` with respect to texplane is (1.0/4.0, 1.5/3.0) = (0.25, 0.50) - these are the numbers set in for `relx` and `rely` - *those are the numbers multiplied by xdim and ydim in the texture_fit() method*
   b. ask `gettexel()` to retrieve the texel
9. `texture_tile()` to map hit location to texture location when `mode == 1`
   a. pass the x and y components of newloc to this method (1.0, 1.5 from the example above in #8)
   b. convert world hit coordinates (the worldx and worldy sent in) to plane pixel coordinates by dividing
      pixel size (`pix_x_size, pix_y_size`)
   c. convert plane pixel coordinates to texel coordinates by mod-ing with the appropriate texture dimension
      (`xdim and ydim`) - this will produce values between 0 and texture dimension -1
   d. ask `gettexel()` to retrieve the texel

10. `gettexel()` method converts relative offsets to actual and converts the texel from `irgb_t` to `drgb_t`. `gettexel()` is sort of like `store_pixel()` from camera, but in reverse. So, instead of storing a pixel, converting it from `drgb_t` items to `irgb_t` items, we will be getting a pixel and the `irgb_t` item will be converted to a `drgb_t` item. Also, since the origin of the image we are reading in is in the upper left corner, and we are assuming our origin is in the lower left corner, we need to make that adjustment - that's what it is referring to by the "upside down" problem. See `camera::store_pixel()` to see what was done there.
    a. convert pixel x and y offset to `imagebuf` offset dealing with the upside problem
    b. convert texel from `irgb_t` to `drgb_t` and save in `texel`