Polymorphism - II:

We have already discussed how the hits and printer functions provide polymorphic behavior in the object "class". Each specialization (plane, sphere) must provide its own characteristic functions because the default functions perform no useful function. Their only use is to prevent an instant segfault when the object is to be tested for a ray intersection.

In other cases, most specializations may want to use the default function. For example, the default ambient function simply calls material_getambient() (Unit 5 Notes). Recall that in the ambient only raytracer, the last steps of the operation are (from ray_trace() function in image.c, Unit 7 Notes):

- add mindist to total_dist
- set intensity to the ambient reflectivity of closest object
- divide intensity by total_dist

The first inclination is to implement the small amount of code in step 2 in the obvious way:

- this_pix.r = closest->mat->ambient.r;
- this_pix.g = closest->mat->ambient.g;
- this_pix.b = closest->mat->ambient.b;

or

- pix_copy(closest->mat->ambient, this_pix);

However that approach would make it not easy to override the default behavior. Thus a better approach is to replace the three lines above by:

- closest->ambient(closest->mat, this_pix);

During object_init(), the object constructor sets the ambient() function pointer to the “material_getambient()” function which contains the three lines of code we just replaced.

While this adds a slight bit of run time overhead, it also provides us with an easy hook with which we may override the material_getambient() with a custom routine. We will see an example of this with the tiled plane object.
Tiled planes

The tiled plane object is commonly found in ray tracing models.

We will use it to demonstrate how the inheritance hierarchy of specialization can be extended. In Object Oriented terminology the object_t structure is called a base class. The base class is at the top or root of a class hierarchy. The plane_t and sphere_t are specializations of the object_t and are called derived classes. The tiled plane tplane_t and the procedural plane pplane_t are further specializations of the plane_t.
The tiled_plane model object description

The tiled plane requires two attributes beyond those of the regular plane.

The dimensions specify the size of the tiling in world coordinates. The altmaterial specifies the color of the alternate tiles.

tiled_plane floor
{ material white
normal   0 1 0
point    0 0 0
dimension 2.0 3.0
altmaterial green
}

A specialization always requires that some aspect of the behavior of the parent class be modified. If no modifications at all were required, we could just use the base class.

A specialization never requires that all aspects of the parent's behavior be overridden. In that case it would be appropriate to create a new class.

When a new tiled_plane is created three structures must be allocated and linked together:

1. The tplane_init() function is called by the model loader. It mallocs the tplane_t and sets the priv pointer in the plane_t.
2. The plane_init() function is called by tplane_init(). It mallocs the plane_t and sets the priv pointer in the object_t.
3. The object_init() function is called by plane_init(). It mallocs the object_t.

The \texttt{tplane\_t} structure and the \texttt{tplane\_init()} function.

The structure is shown below and it must be filled in by \texttt{tplane\_init()} from the input data.

\begin{verbatim}
/* Tiled plane... descendant of plane */
typedef struct tplane_type
{
    char   matname[NAME_LEN]; /* for the background color */
    material_t *background;   /* background color         */
    double     dimension[2];  /* dimension of tiles       */
}  tplane_t;
\end{verbatim}

The dimension parameter specifies the (x, z) dimensions of a single tile in world coordinates. The \textit{altmaterial} is the name of the material used in alternate tiles. The \texttt{tplane\_init()} function must ask \texttt{material\_getbyname()} to lookup the corresponding material_t.

\begin{verbatim}
tiled\_plane floor
{
    material white
    normal 0 1 0
    point 0 0 0
    dimension 2.0 3.0
    altmaterial green
}
\end{verbatim}

The \texttt{tplane\_init()} function

\begin{verbatim}
void tplane\_init(FILE *in, model\_t *model)
\end{verbatim}

The function is called from the model loaders. Its missions are to:

- invoke \texttt{plane\_init(in, model, 2)} to create the \texttt{plane\_t} and \texttt{object\_t} structures
- The 2 is passed to \texttt{plane\_init()} to tell it not to process more than two attributes. You will need to fix \texttt{plane\_load\_attributes()} to make it honor this limit; instead of it going through the while loop until it reaches '}', it will use that attrmax argument to see if the number of attributes read in has reached that max number.
- recover a pointer from to the object structure from the model->objs list
- recover a pointer to the plane structure from the priv pointer of the object
- malloc a \texttt{tplane\_t} and set the priv pointer in the \texttt{plane\_t} structure
- parse the attributed data (call \texttt{tplane\_load\_attributes()} function)
- set the ambient pointer in the \texttt{object\_t} to point to the \texttt{tplane\_ambient()} function
- get the background color using \texttt{material\_getbyname()} and set it to \texttt{tpln->background}
- set the printer pointer in the \texttt{object\_t} to point to \texttt{tplane\_print}
- set \texttt{obj\_type} field in the \texttt{object\_t} to "tiled plane".

The \texttt{tplane\_init()} function need not override the hits function provided in \texttt{plane\_init()}. 
**The `tplane_load_attributes()` function**

/* similar to the other load_attributes functions */

**The `tplane_print()` function**

```c
void tplane_print(object_t *obj, FILE *out)

  ● invoke the `plane_print()` function
  ● print the `tplane` attributes

**The `tplane_ambient()` function**

This is the function that actually gives the `tplane` its characteristic behavior.

```c
void tplane_ambient(object_t *obj, material_t *mat, drgb_t *value) {
  int foreground = tplane_foreground(obj);

  if (foreground)
    material_getambient(obj, obj->mat, value);
  else
    copy ambient reflectivity from background material
}
```

**The `tplane_foreground()` function**

```c
int tplane_foreground(object_t *obj) {
  Compute \( x_{ndx} \) = tile index of the hitpoint in the x direction
  Compute \( z_{ndx} \) = tile index of the hitpoint in the z direction

  if ((\( x_{ndx} + z_{ndx} \)) is an even number)
    return(1);
  else
    return(0);
}
```

A tile index is an `int` that is computed by adding 10,000 to the x (or z) coordinate of the hitpoint and dividing by the x (or z) dimension of the tile.

Typically, foreground is associated with the object material and background with the `tplane` altmaterial. If you get that backwards, your colors will just alternate the opposite way.
Procedural surfaces

Procedural surfaces are those in which an object's reflectivity properties are *modulated* as a function of the location of the hit point on the surface of the object.

There are literally an infinite number of ways to do this. In the next few pages we propose a framework for incorporating procedurally shaded surfaces into raytraced images.
Implementation of procedural shaders

Construction of such shaders is facilitated by the use of both inheritance and polymorphism within a C language framework. The procedurally shaded plane is a lightweight refinement of the `plane_t`.

```c
typedef struct pplane_type
{
    int  shader;
}  pplane_t;
```

The distinction between a standard plane and a procedurally shaded plane is made at object *initialization time* by the `pplane_init()` function when it establishes a single function pointer (for ambient only images) that provides the polymorphic behavior.

That function pointer is taken from a table of pointers to programmer provided functions that are contained in the module `pplane.c` and perform the procedural shading. These procedural shading functions are passed pointers to the `object_t` structure and to the `drgb_t intensity vector` whose *(r,g,b)* components are filled in procedurally. Here is an example in which there are three possible shaders.

```c
void (*pplane_shaders[ ])(object_t *obj, material_t *mat, drgb_t *value) =
{
    pplane0_ambient,
    pplane1_ambient,
    pplane2_ambient,
};
```

```c
#define NUM_SHADERS sizeof(pplane_shaders)/sizeof(void *)
```

Note that:
1. The number of elements in the array is not explicitly specified.
2. The value `NUM_SHADERS` can be computed by dividing the size of the table by the size of a single pointer.

The index of the shader to be used is supplied in the model description as shown below.

```c
pplane floor
{
    material gray
    normal  0 1 0
    point    0 0 -8
    shader   0
}
```

```c
pplane backwall
{
    material gray
    normal  0 0 1
    point    4 3 -8
    shader   1
}
```
The `pplane_init()` function

As shown below, the `pplane_init()` function simply invokes the `plane_init()` function to construct the object and then overrides the default `getamb()` function, replacing it with the shader function whose index is provided in the model description files.

```c
object_t *pplane_init(FILE *in, model_t *model, int attrmax)
{
    plane_t  *pln;
    object_t *obj;
    int  mask;
    plane_init(in, model, 2);
    /* Recover object pointer from object list */
    /* Recover plane pointer from object pointer */
    /* malloc a pplane structure and link plane structure to it */

    count = pplane_load_attributes(in, ppln, 0);
    assert(count = 1);
    strcpy(obj->obj_type, "pplane");
    obj->ambient = pplane_shaders[ppln->shader];
    obj->printer = pplane_print;
}
```

The mysterious `attrmax` parameter finally gets to do something!
Tiled shading

To produce a tiled “floor” the modulation must be a function of the $x$ and $z$ coordinates because the $y$ coordinate does not vary on the floor. For a “backwall” it would be necessary to modulate $x$ and $y$, and for a “sidewall” it would be $y$ and $z$.

```c
void pplane0_ambient(object_t *obj, drgb_t *value)
{
    int    ix;
    int    iz;
    ix = 2 * obj->hitloc.x + 1000;
    iz = 2 * obj->hitloc.z + 1000;
    pix_copy(&obj->mat->ambient, value);

    // test for odd or even sum
    if ((iz + ix) & 1) {
        value->r = 0.0;   // make pixel cyan
    }
    else {
        value->b = 0.0;   // make pixel yellow
    }
}
```

The factor of 2 controls the width of the tiles. The larger the factor the smaller the tile. The value of 1000 is known as Westall’s hack for preventing an ugly double wide strip at the origin.
Continuously modulated shading

The image shown below is produced by a procedural shader that continuously modulates the ambient reflectivity.

The modulation function is shown below. A vector \( V \) in the direction from the point defining the plane location to the hitpoint is computed first. Then the angle that the vector makes with the positive \( X \) axis is computed. Finally the red, green and blue components are modulated using the function \( 1 + \cos(w t + f) \) where the angular frequency \( w \) is 2 for all three colors, and phase angles \( f \) are 0, \( 2\pi/3 \), and \( 4\pi/3 \) respectively. Different effects may be obtained by using different frequencies and phase angles for each color, and it is also possible to combine continuous modulation with striping or tiling.

```c
vec_diff(&p->point, &obj->hitloc, &vec);

v1 = (vec.x / sqrt(vec.x * vec.x + vec.y * vec.y));
t1 = acos(v1);

if (vec.y < 0)
    t1 = 2 * M_PI - t1;

value->r = 6 * (1 + cos(2 * t1));
value->g = 6 * (1 + cos(2 * t1 + 2 * M_PI / 3));
value->b = 6 * (1 + cos(2 * t1 + 4 * M_PI / 3));
```

10