Debugging large programs

Although *gdb* is a powerful tool that should always be used in unit testing and when trying to identify the location of a fatal fault, it is not always the best tool for identifying errors during integrated system tests for large, complex programs.

Here it is often the case that having the program print out crucial elements of its internal state for offline analysis by the programmer may be required. After an error is identified from the print file, then *gdb* often is the best tool to pinpoint the source of the problem.

Finally, it is generally true that very large programs are never truly bug free. Testing usually stops when the failure rate of the program becomes acceptably low in someone's point of view. There are good economic reasons for this.

Therefore it is common to build debugging code into large programs. This code is *never* removed because the chances are good that additional failures will occur as the program is used and certainly when it is modified.

The *make* facility in conjunction with the C preprocessor provide a good mechanism for selective activation and deactivation of debugging code. It is required that you make use of this facility

Recall the macro:

```
DEBUG = -DDBG_PIX -DDBG_HIT -DDBG_WORLD -DDBG_AMB -DDBG_FIND
```

that was present in the *makefile*. The -D flag allows one to specify to gcc that a particular symbol is *defined* in a sense that will be described later. In its current state the macro *arms* many debugging statements. If that leads to information overload the debugging data could be reduced by selective arming:

```
DEBUG = -DDBG_PIX -DDBG_HIT -DDBG_AMB
```

or when its thought that the program is ready for prime time debugging can be completely disabled by simply commenting out the macro definition and rebuilding the system.

```
# DEBUG = -DDBG_PIX -DDBG_HIT -DDBG_WORLD -DDBG_AMB -DDBG_FIND
```

Including debugging code in your program

When confronted with a problem in your program, since you don't know what the problem is, its hard to know where to put the debugging code. Hence the best technique is to put it *everywhere.* Just write it in as you go.

Here are some typical examples that I have used. Note: The only new line character \n you will see is the first fprintf(). This is done because you want all of the data associated with a single ray to appear on one line. You also want to carefully format so that the columns line up. The human brain has awesome pattern recognition skills but you need to show it a pattern!

Debug code should be enclosed in *#ifdef* / *#endif* blocks. In this way it easy to enable or disable debug prints by simply recompiling with the desired set of symbols defined.

in make_image(): (This one should generally always be on when debug is active since the pixel coords are key to all that we see)

```
   for (y = 0; ... ){
      for (x = 0; x < proj->win_size_pixel[0]; x++){

#ifdef DBG_PIX
         fprintf(stderr, "\nPIX %4d %4d - ", x, y);
#endif
```

in make_pixel(): (This one you might want to turn off once you are REAL SURE *map_pix_to_world()* is functional.)

```
      map_pix_to_world(x, y, world);
#ifdef DBG_WORLD
      fprintf(stderr, "WRL (%5.1lf, %5.1lf) - ",
                              world[0], world[1]);
#endif
```

in *ray_trace():* If *find_closest_obj()* believes it hit something its useful to see what it thought was the nearest hit, how far away it was and where the hit occurred.

```
   closest = find_closest_obj(model->scene,
                              base, dir, NULL, &mindist);

    if (closest == NULL)
       return;

#ifdef DBG_HIT
   fprintf(stderr, " HIT %4d: %5.1lf (%5.1lf, %5.1lf, %5.1lf) - ",
                    closest->objid, mindist,
                    closest->hitloc[0], closest->hitloc[1],
                    closest->hitloc[2]);
#endif
```

and after computing the ambient intensity of the pixel value

```
 #ifdef DBG_AMB
   fprintf(stderr, "AMB (%5.1lf, %5.1lf, %5.1lf) - ",
                    intensity[0], intensity[1], intensity[2]);

#endif
```

in *find_closest_obj():* After firing the ray at each object print the *id* of the object and the distance to the hit (which of course may be -1) if there was a miss. DBG_FIND should typically be used in initial testing with only one or two objects. If you have too many this will give you info overload.

```
      if (obj != last_hit)
      {
         dist = obj->hits(base, dir, obj);
#ifdef DBG_FIND
        fprintf(stderr, "FND %4d: %5.1lf - ",
                         obj->objid, dist);
#endif
```

Running the program:

Its a good idea to start with a *small* pixmap. Here is one that is 5 x 3 pixels. This is way to small to produce a useful image but if you have a broken program, it is not going to produce one anyhow so this is where to start:

```
 ray 5 3 < a2t1.txt > a2t1.ppm
Window size in pixels
    5     3

Window size in world coordinates
    8     6

Location of Viewpoint
  0.0    0.0    3.0

Dumping object of type Plane
Material data -
Ambient -    5.000   5.000   0.000
Diffuse -    0.000   0.000   0.000
Specular -   0.000   0.000   0.000

Plane data
normal -     0.000   0.000  -1.000
point  -     0.000   0.000  -5.000

PIX   0    0 - WRL ( -4.0,  -3.0) - FND  100:  15.5 -  HIT  100:  15.5 (-10.7,  -8.0,  -5.0) - AMB (  0.3,   0.3,   0.0) -
PIX   1    0 - WRL ( -2.0,  -3.0) - FND  100:  12.5 -  HIT  100:  12.5 ( -5.3,  -8.0,  -5.0) - AMB (  0.4,   0.4,   0.0) -
PIX   2    0 - WRL (  0.0,  -3.0) - FND  100:  11.3 -  HIT  100:  11.3 (  0.0,  -8.0,  -5.0) - AMB (  0.4,   0.4,   0.0) -
PIX   3    0 - WRL (  2.0,  -3.0) - FND  100:  12.5 -  HIT  100:  12.5 (  5.3,  -8.0,  -5.0) - AMB (  0.4,   0.4,   0.0) -
PIX   4    0 - WRL (  4.0,  -3.0) - FND  100:  15.5 -  HIT  100:  15.5 ( 10.7,  -8.0,  -5.0) - AMB (  0.3,   0.3,   0.0) -
PIX   0    1 - WRL ( -4.0,   0.0) - FND  100:  13.3 -  HIT  100:  13.3 (-10.7,   0.0,  -5.0) - AMB (  0.4,   0.4,   0.0) -
PIX   1    1 - WRL ( -2.0,   0.0) - FND  100:   9.6 -  HIT  100:   9.6 ( -5.3,   0.0,  -5.0) - AMB (  0.5,   0.5,   0.0) -
PIX   2    1 - WRL (  0.0,   0.0) - FND  100:   8.0 -  HIT  100:   8.0 (  0.0,   0.0,  -5.0) - AMB (  0.6,   0.6,   0.0) -
PIX   3    1 - WRL (  2.0,   0.0) - FND  100:   9.6 -  HIT  100:   9.6 (  5.3,   0.0,  -5.0) - AMB (  0.5,   0.5,   0.0) -
PIX   4    1 - WRL (  4.0,   0.0) - FND  100:  13.3 -  HIT  100:  13.3 ( 10.7,   0.0,  -5.0) - AMB (  0.4,   0.4,   0.0) -
PIX   0    2 - WRL ( -4.0,   3.0) - FND  100:  15.5 -  HIT  100:  15.5 (-10.7,   8.0,  -5.0) - AMB (  0.3,   0.3,   0.0) -
PIX   1    2 - WRL ( -2.0,   3.0) - FND  100:  12.5 -  HIT  100:  12.5 ( -5.3,   8.0,  -5.0) - AMB (  0.4,   0.4,   0.0) -
PIX   2    2 - WRL (  0.0,   3.0) - FND  100:  11.3 -  HIT  100:  11.3 (  0.0,   8.0,  -5.0) - AMB (  0.4,   0.4,   0.0) -
PIX   3    2 - WRL (  2.0,   3.0) - FND  100:  12.5 -  HIT  100:  12.5 (  5.3,   8.0,  -5.0) - AMB (  0.4,   0.4,   0.0) -
PIX   4    2 - WRL (  4.0,   3.0) - FND  100:  15.5 -  HIT  100:  15.5 ( 10.7,   8.0,  -5.0) - AMB (  0.3,   0.3,   0.0
```

Pointers to functions

Pointer variables may also hold the address of a function and be used to invoke the function indirectly:

```
gcc p32.c
cat p32.c

/* p32.c */

int adder(int a,int b) {
   return(a + b);
}

main(){
   int (*ptrf)(int, int);  // declare pointer to function
   int sum;

   ptrf = adder;           // point it to adder (note no '&')

   sum = (*ptrf)(3, 4);    // invoke it
   printf("sum = %d \n", sum);
}
a.out
sum = 7
```

Function pointers as do-it-yourself polymorphism

Recall the the *obj_t* structure contained a function pointer:

```
typedef struct obj_type {
   struct  obj_type *next; /* objects are linked        */
   int     objtype;      /* light, sphere, floor, etc   */
   double  (*hits)(double *base, double *dir,
                   struct obj_type obj);
     :
```

The *hits* pointers must be set in the initialization module. In the *sphere_init* function this pointer should be set as follows:

```
   obj->hits = sphere_hits;
```

where the *sphere_hits()* function is declared as follows.

```
/* sphere_hits – Does the ray hit our sphere?
 *
 * Parameters: base – (x, y, z) coords of the base of the ray
 *             dir  - (x, y, z) direction of the ray
 *             obj – object to be tested for the hit
 */
double sphere_hits(double *base, double *dir, obj_t *obj){

  // awesome code

}
```

The *find_closest_object()* function.

When a ray is fired from the viewpoint through a pixel, depending upon the nature of the objects in the scene, the ray may pass through several objects or it may hit none at all. The color of the pixel will be derived from the *material* properties of the *first* object the ray hits.

The ray trace function should rely upon *find_closest_object* to return a point to the *closest* object that is hit by the ray. If *none* of the objects in the scene is hit, NULL must be returned. The distance from the base of the ray to the nearest hitpoint is returned in *mindist.*

```
closest = find_closest_object(model->scene,
                              base, dir, NULL, &mindist);
```

The *find_closest_object()* function processes the complete object list and calls the appropriate hits function for each object found.

```
obj = lst->head;
while ((obj != NULL)) {
   if (obj != last_hit)
   {
      dist = obj->hits(base, dir, obj);
   :
   :
```

This approach provides a much cleaner and more efficient mechanism than the functionally equivalent way:

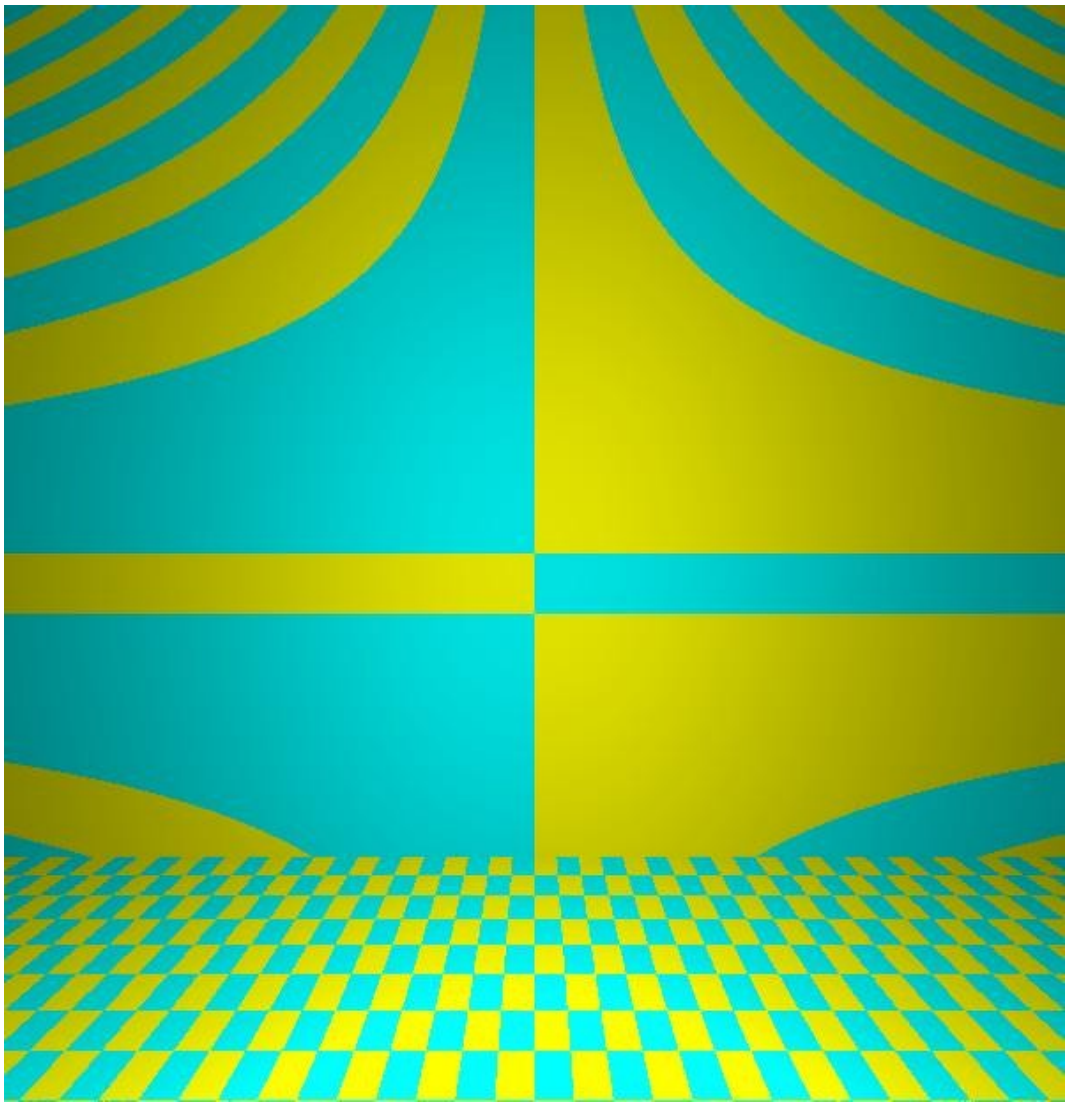
```
obj = lst->head;
while ((obj != NULL))
{
    switch (obj->objtype)
    case SPHERE:
        dist = sphere_hits(base, dir, obj);
        break;
    case PLANE:
        etc
```

From the software reliability perspective an important advantage of this approach is that when a new object type is added *it is not necessary to modify find_closest_object().*

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 an extremely lightweight refinement of the *plane_t*. In fact it is such a lightweight refinement that there is no need for a *pplane_t* data structure.

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 that provides the polymorphic behavior.

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

```
static void (*plane_shaders[])(obj_t *obj, double *intensity) =
{
   pplane0_amb,
   pplane1_amb,
   pplane2_amb
};
#define NUM_SHADERS sizeof(plane_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.

```
8 6            world x and y dims
0 0  3         viewpoint (x, y, z)
20             pplane
8  8  8        ambient
0 0 0          diffuse
0 0 0          specular
0 0  1         plane is a "back wall"
0 0 -6         located 6 units behind the screen
1              shader selector index
```

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.

```
obj_t *pplane_init(FILE *in, int objtype){
   obj_t *new;
   double dndx;
   int    ndx;

   new = plane_init(in, objtype);

   ndx = vec_get1(in, &dndx);
   if (ndx != 1)
      return(0);

   ndx = dndx;
   if (ndx >= NUM_SHADERS)
      return(0);

   new->getamb = plane_shaders[ndx];
   return(new);
}
```

The *getamb* element of the *obj_t()* structure is a pointer to a void function which is passed pointers to the object structure and the *intensity* vector.

```
typedef struct obj_type{
   struct  obj_type  *next;      /* Next object in list        */
   int     objid;                /* Numeric serial # for debug */
   int     objtype;              /* Type code (14 -> Plane )   */

/* Optional plugins for retrieval of reflectivity */
/* useful for the ever-popular tiled floor        */

   void    (*getamb)(struct obj_type *, double *intensity);
```