last time: plane_hits() function.
(sphere_hits() to come)

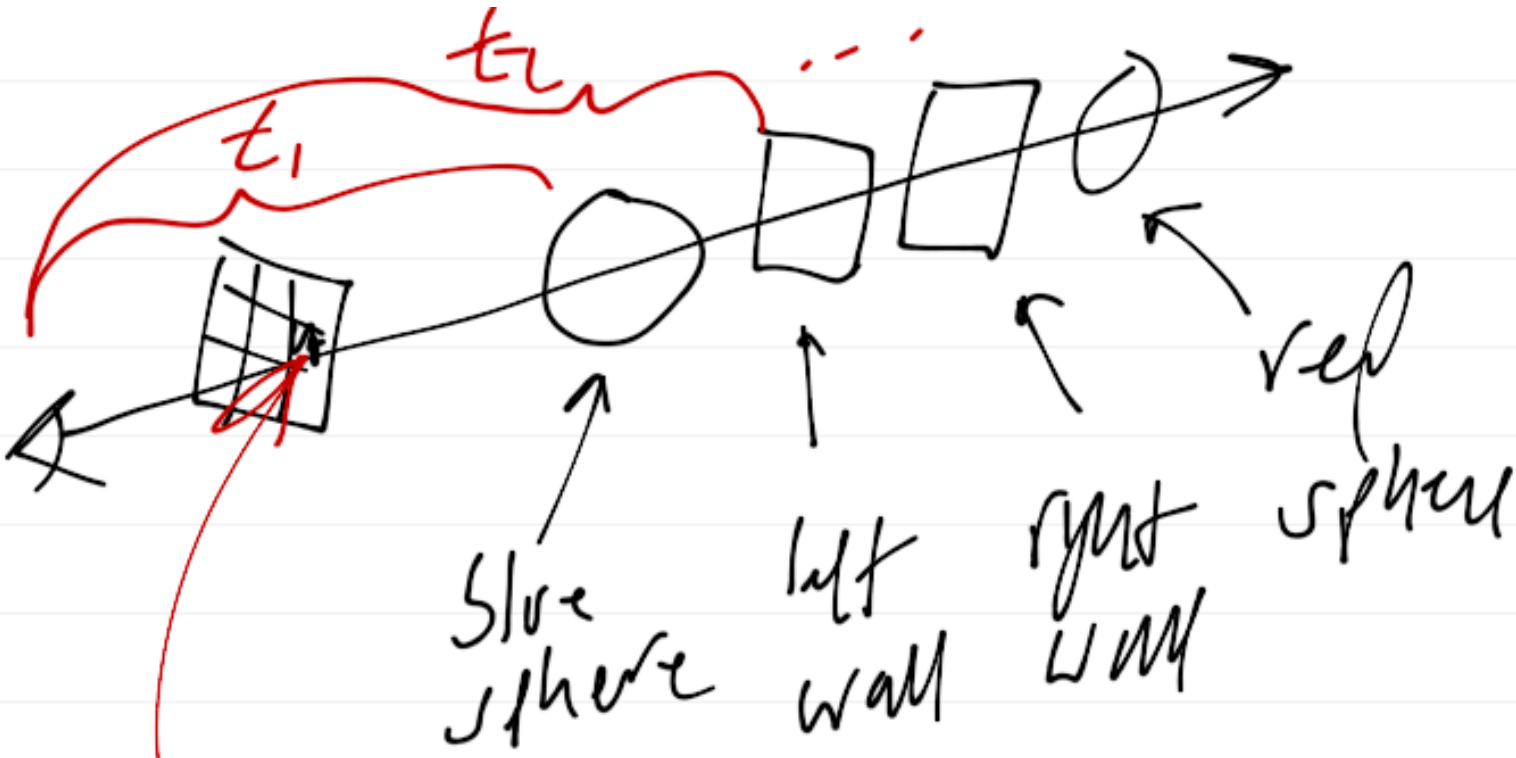
- called via function pointer stored by object_t:

obj → hits(...)

↑

same argument list for both plane, sphere, ...

- how is this used in ray tracer?



this pixel color will be set to the color of the closest hit object (min. dist.)

- how do we find the closest object?

- same problem as: "find smallest element in unsorted array"

```
int array[N], i;
int smallest = HUGE;

for(i=0; i<N; i++)
    if (array[i] < smallest)
        smallest = array[i];
```

- object_find_closest algorithm:

```
for each obj in obj list:
    if ( obj == last_hit ||
        dist = obj->hits() < 0 )
        continue; // skip this obj
    if ( closest == NULL ||
        dist < mindist ) {
        mindist = dist;
        closest = obj;
    }
return (closest);
```

- object_find_closest syntax:

return type: object_t *

(ptr to closest object,
can be NULL if ray
doesn't hit anything)

arguments:

list_t *objs; // obj list

vec_t base, vec_t dir; // ray

object_t *last_hit; // last hit obj (can be NULL)

double *ret_dist;

// distance to hit point,
pass-by-ref (ret-in value)

local variables:

```
double dist, mindist = -1;
```

set to -1 because smallest valid hit distance must be > 0 (so -1 means no hit)

```
Object_t *obj = NULL;
Object_t * closest = NULL;
```

↳ what we return

- remember useful function calls:

```
list_reset(objs); // reset internal iterator
list_not_end(objs);
obj = (object_t *) list_get_data(objs);
list_next_link(objs);
```

≠

```
dist = obj->hits(obj, base, dir)
```

A5G 2: using object_find_closest
& cleaning up where
list_t *objs, *mats;
lists are stored:
- in a new encapsulating
object called model_t
- model_t also stores
new camera_t
object

(p 90 of PDF notes describes model_t object)

```
typedef struct model_type
{
    camera_t *cam;
    list_t   *mats;
    list_t   *objs;
} model_t;
```

(Note: Asst 2 doesn't use camera_t yet, can comment out)

- only two member methods:

```
model_t *model_init(FILE*);
void
    model_print(
        model_t*,
        FILE*);
```

model.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <math.h>
#include "vector.h"
    "pixel.h"
    "list.h"
    "material.h"
    "object.h"
    "model.h"
```

ORDER MATTERS

"camera.h"

```
model_t *model_init(FILE *in)
{
    material_t  *mat;
    object_t    *obj;
    char        token[16];

// get memory, check for NULL
    model = (model_t *)
        malloc(sizeof(model_t));
    memset(model, 0, sizeof(model_t));
```

```
// create lists
model->mats = list_init();
model->objs = list_init();

// read in model file
while (fscanf(in, "%s", token)) {
    // as in lab 4
    if (!strcmp(token, "camera"))
        camera_init(in,
            &(model->cam), 0);
```

pointer to pointer

```
if (!strcmp(token, "material"))
    material_init(in,
        model->mats, 0);
if (!strcmp(token, "plane"))
    plane_init(in,
        model->objs,
        model->mats, 0);
if (!strcmp(token, "sphere"))
    sphere_init(in,
        model->objs,
        model->mats, 0);
} // end while
```

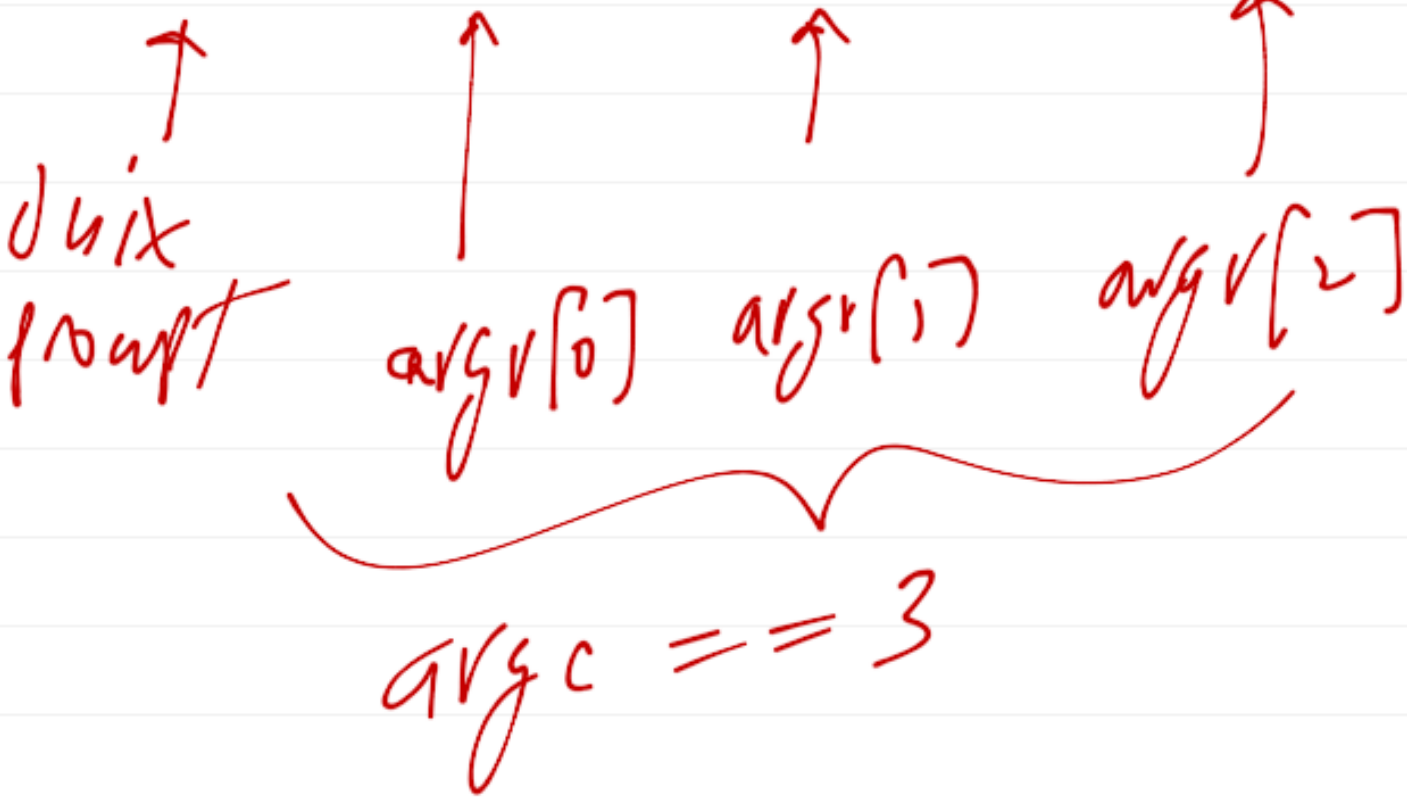
```
    return (model);
}

model_print (model_t *model,
             FILE *out)
{
    camera_print (model->cam, out);
    material_list_print (model->mats,
                         out);
    object_list_print (model->obj, out);
}
```

- use of the model object cleans up main.c !.

```
int main(int argc, char *argv[])
```

usage can now be

```
$ ./main plane.txt hitlist.txt
```



```
int main(...)
{
    model_t *model=NULL;

    model=model_init(stdin);
```

— or —

```
    FILE *model_file, *hits_file;
if(argc!=3) {
    fprintf(stderr, "Usage: %s
        <model file> <hits file>\n",
            argv[0]);
    exit(1);
}
```

```
if ((model_file = fopen(argv[1], "r"))
    == NULL) {
    fprintf(stderr, "Error opening
        file: %s\n", argv[1]);
    exit(1);
}
model = model_init(model_file);
fclose(model_file);
```

↑ DON'T FORGET THIS!

— same with hits file

Makefile revisited:

```
OBJS = \
    pixel.o \
    material.o \
    object.o \
    plane.o \
    camera.o \
    model.o
```

```
all: libvec liblist main

libvec: libvec.a
libvec.a: vector.o matrix.o
	ar rcs $@ $?
	ranlib $@

liblist: liblist.a
liblist.a: list.o
	ar rcs $@ $?
	ranlib $@
```

matrix.o ?

- recall typedef double vec_t[3];

vec_t v; // v[0], v[1], v[2]
x y z

typedef double mtx_t[3][3];

mtx_t m; // m[0][0] m[0][1] m[0][2]
m[1][0] m[1][1] m[1][2]
m[2][0] m[2][1] m[2][2]

m[0], m[1], m[2]
is itself like a vec_t

- two functions:

void vec_xform(mtx_t, //input
vec_t, //input
vec_t); //output

vector-matrix mult ✓

void mat_xpose(mtx_t, //in
mtx_t); //out

matrix transpose ✓

matrix.c:

```
#include <stdio.h>
        ⋮
#include "vector.h"
#include "matrix.h"

void mat_xpose (mtx_t m1,
                mtx_t m2)
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

m1 m2

```
int i, j;
for(i=0; i<3; i++)
    for(j=0; j<3; j++)
        m2[i][j] = m1[j][i];
}
```

void vec_xform(mtx_t m,
vec_t v1,
vec_t v2)

$$\begin{pmatrix} v2[0] \\ v2[1] \\ v2[2] \end{pmatrix} = \begin{pmatrix} m[0][0] & m[0][1] & m[0][2] \\ m[1][0] & m[1][1] & m[1][2] \\ m[2][0] & m[2][1] & m[2][2] \end{pmatrix} \begin{pmatrix} v1[0] \\ v1[1] \\ v1[2] \end{pmatrix}$$

```
int i;
for(i=0; i<3; i++)
    v2[i] = vec_dot(m[i], v1);
```

```
for(i=0; i<3; i++)
    v2[i] = vec_dot(m[i], v1);
// BUG!
```

- compiles & runs, syntactically ok.
- but call to

  vec_xform(m2, v4, v4)

produces incorrect result — a semantic bug — fix in lab6 (hint: aliasing problem)

The camera data type:

stores:

position	(rect_t)
screendim	(640, 480)
worlddim	(8, 6)
name	("camera1")
cookie	(494959123)

camera.h:

```
#define CAM_COOKIE 4949523
typedef struct camera_type
{
    int   cookie;
    char  name[NAME_LEN];
    int   pixel_dim[2];
    int   world_dim[2];
    vec_t view_point;
}
```

method prototypes:

```
void camera_init(FILE*,
        camera_t**, int);
```

pointer to pointer
(or, pointer passed-by-ref)

```
void camera_load_attributes
        (FILE*, camera_t**);
void camera_print(camera_t**,
        FILE*);
char* camera_getname(camera_t**)
```

camera.c:

```
void camera_init(FILE *fin,
        camera_t ** cam,
        int attr_mask)
{
  // malloc camera_t struct
  (*cam) = (camera_t *)
            malloc(sizeof(camera_t));
  memset((void *) *cam, 0,
            sizeof(camera_t));
  (*(cam))->cookie = CAM_COOKIE;
  camera_load_attributes(in, *(cam));
}
```

*can notation is
dereferencing
pointer to pointer
(so it's a pointer)

same idea as:

```
int swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

pass-by-ref

camera file entry is:

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

write camera_load_attributes & camera_print to read/write file