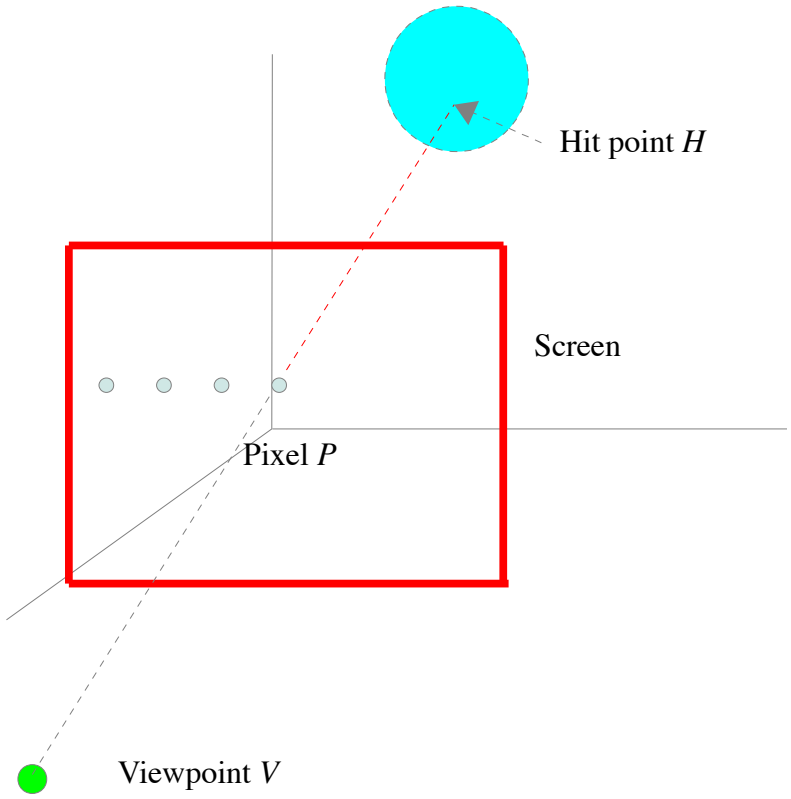
Hit functions

Given the viewpoint, ray direction and a pointer to an *obj_t* the mission of a hit function is to determine if the ray hits the object. If it does, the hit point and the normal vector at the hitpoint should be returned.



Given *V*, *D* and an object structure *O* the mission of a hit function is to determine if a ray based at *V* traveling in direction *D* hits *O*.

All points on the ray may be expressed as $V + t\,D$ for $-\mathbb{N} < t < \mathbb{N}$

Ray direction:

Distance to hit point:

Location of hit point:

Prototype for the hits functions

To determine if a ray hits an object you must add a *hits_objtype* function to your *sphere.c, plane.c etc* modules. These modules should also contain the loading and dumping code for the specific object type. A sample prototype is shown below:

```
 /*
  * hits_plane -         ????
  *
  * Parameters: base – (x, y, z) origins of the ray
  *             dir – (x, y, Z) direction of the ray
  *             obj – The object we want to hit :)
  */
double  hits_plane(double *base, double *dir, obj_t *obj);
```

Pointers to the hits' function should be stored in the object structure at the time it is created

```
   obj->hits = hits_plane;
```

Determining if a ray hits a plane

This basic strategy will be used in *all* hits functions:

0 - Assume that V represents the start of the ray and D is a *unit* vector in its direction

1 - Derive an equation for an arbitrary point P on the surface of the object.

2 - Recall that all points on the ray are expressed as $V + tD$

3 - Substitute $V + tD$ for P in the equation derived in (1).

4 - Attempt to solve the equation for t.

5 - If a solution t_h can be found, then $H = V + t_h D$.

A plane in three dimensional space is defined by two parameters

A normal vector $N = (n_x, n_y, n_z)$

A point $Q = (q_x, q_y, q_z)$ through which the plane passes.

A point $P = (p_x, p_y, p_z)$ is on the plane if and only if:

$N \text{ dot } (P - Q) = 0$ because, if the two points P, Q lie in the plane, then the vector from one to the other $(P - Q)$ also lies in the plane and is necessarily perpendicular to the plane's normal.

We can rearrange this expression to get:

$$N \text{ dot } P - N \text{ dot } Q = 0$$

$$N \text{ dot } P = N \text{ dot } Q \qquad (1)$$

Recall that the the location of any points on a ray based at V with direction D is given by:

$$V + t D$$

Therefore we may replace the P in equation (1) by $V + tD$ and get:

$$N \text{ dot } (V + tD) = N \text{ dot } Q \qquad (2)$$

Some algebraic simplification yields allow us to solve this for *t*

$$N \text{ dot } (V + tD) = N \text{ dot } Q \qquad (2)$$
$$N \text{ dot } V + N \text{ dot } tD = N \text{ dot } Q$$
$$N \text{ dot } tD = N \text{ dot } Q - N \text{ dot } V$$
$$t\,(N \text{ dot } D) = (N \text{ dot } Q - N \text{ dot } V)$$
$$t_h = (N \text{ dot } Q - N \text{ dot } V) / (N \text{ dot } D)$$

The location of the hitpoint that should be stored in the *obj_t* is thus:

$$H = V + t_h D$$

The normal at the hitpoint which must also be saved in the *obj_t* is just N

Unlike other quadric surfaces, there is only a single point at which a ray intercepts a plane. Therefore unlike equations we will see later, this one is not quadratic. There *are* some special cases we must consider:

(1) $(N \text{ dot } D) = 0$ In this case the direction of the ray is perpendicular to the normal to the plane. This means the *ray is parallel to the plane.* Either the ray lies in the plane or misses the plane entirely. We will always consider this case a *miss* and return -1. Attempting to divide by 0 will cause your program to either fault and die or return a meaningless value.

(2) $t_h < 0$ In this case the hit lies behind the viewpoint rather than in the direction of the screen. This should also be considered a miss and *-1* should be returned.

(3) The hit lies on the view point side of the screen.

$$H = (h_x, h_y, h_z) \text{ if } h_z > 0 \text{ the hit is on the wrong side}$$

and -1 should be returned.

Determining if a ray hits a sphere.

Assume the following:

V = *viewpoint or start of the ray*

D = *a unit vector in the direction the ray is traveling*

C = *center of the sphere*

r = *radius of the sphere.*

The arithmetic is much simpler if the center of the sphere is at the origin. So we start by moving it there! To do so we must make a compensating adjustment to the base of the ray.

$C' = C - C = (0, 0, 0)$ = *new center of sphere*

$V' = V - C$ = *new base of ray*

D *does not change*

A point P on the sphere whose center is (0, 0, 0) necessarily satisfies the following equation:

$$p_x^2 + p_y^2 + p_z^2 = r^2 \qquad (1)$$

All points on the ray may be expressed in the form

$$P = V' + t\,D = (v'_x + td_x,\ v'_y + td_y,\ v'_z + td_z) \qquad (2)$$

where t is the Euclidean distance from V' to P

Thus we need to find a value of t which yields a point that satisfies the two equations. To do that we take the *(x, y, z)* coordinates from equation (2) and plug them into equation (1). We will show that this leads to a quadratic equation in t which can be solved via the quadratic formula.

$$(v'_x + td_x)^2 + (v'_y + td_y)^2 + (v'_z + td_z)^2 = r^2$$

Expanding this expression by squaring the three binomials yields:

$$(v'_x{}^2 + 2tv'_x d_x + t^2 d_x^2) + (v'_y{}^2 + 2tv'_y d_y + t^2 d_y^2) + (v'_z{}^2 + 2tv'_z d_z + t^2 d_z^2) = r^2$$

Next we collect the terms associated with common powers of t

$$(v'_x{}^2 + v'_y{}^2 + v'_z{}^2) + 2t (v'_x d_x + v'_y d_y + v'_z d_z) + t^2(d_x{}^2 + d_y{}^2 + d_z{}^2) = r^2$$

Now we reorder terms as decreasing powers of t and note that all three of the parenthesized trinomials represent dot products.

$$(D \text{ dot } D)t^2 + 2 (V' \text{ dot } D)\, t + V' \text{ dot } V' - r^2 = 0$$

We now make the notational changes:

$$a = D \text{ dot } D$$
$$b = 2 (V' \text{ dot } D)$$
$$c = V' \text{ dot } V' - r^2$$

to obtain the following equation

$$at^2 + bt + c = 0$$

whose solution is the standard form of the quadratic formula:

$$t_h = \frac{-b +/- \text{sqrt}(b^2 - 4ac)}{2a}$$

Recall that quadratic equations may have 0, 1, or 2 real roots depending upon whether the discriminant:

$$(b^2 - 4ac)$$

is negative, zero, or positive. These three cases have the following physical implications:

negative	=>	ray doesn't hit the sphere
zero	=>	ray is tangent to the sphere hitting it at one point (we will consider this a miss).
positive	=>	ray does hit the sphere and would pass through its interior (this is the *only* case we consider a *hit).*

Furthermore, the two values of *t* are the distances from the base of the ray to the points(s) of contact with the sphere. We always seek the *smaller* of the two values since we seek to find the "entry wound" not the "exit wound".

Therefore, the *hits_sphere()* function should return

$$t_h = \frac{-b - sqrt(b^2 - 4ac)}{2a}$$

if the discriminant is positive and

$$t_h = -1$$

otherwise.

Determining the coordinates of the hit point on a sphere.

The *hits_sphere()* function should also fill in the coordinates of the *hit* in the *obj_t* structure.

The *(x, y, z)* coordinates are computed as follows.

$$H = V + t_h D$$

Important items to note are:

The actual base of the ray V and not the translated base V' must be used

The vector D must be a *unit vector* in the direction of the ray.

Determining the surface normal at the hit point.

The normal at any point on the surface of a sphere is a vector from the *center* to the *point*. Thus

$$N = P - C$$ *(note that N will be a unit vector <==> r = 1)*

Therefore a unit normal may be constructed as follows:

$$N_u = (H - C) / || (H - C) ||$$

Functions and parameters

Functions provide a useful way to partition a large program in to a collection of small manageable entities

Each function should be:

1 - small (ideally less than 40 lines of code)
2 - perform a single, well defined task (*e.g.*, print the attributes of a sphere)
3 - designed as to minimize information flow and shared data

Proper use of functions

1 - speeds development
2 - reduces the difficulty of the debugging process
3 - facilitates the process of adding new functionality to a program.

A C language function is defined as follows

```
return-type function-name(parameterType parameterName, … ,
                          parameterType parameterName){
     local variables;
     executable code;
}
```

A specific example is:

```
int adder(int a, int b){
   int sum;

   sum = a + b;
   return(sum);
}
```

Parameter passing

There are many possible mechanisms by which parameters may be passed.

The standard C language uses *call-by-value*.

In this approach a *copy* of the parameter is placed on the stack. The called function is free to modify the copy as it wishes, but this will have *no effect* on the value held by the caller.

```
/* p23.c */

/* Parameter passing 1 */

int try_to_mod(int a){
   printf("The address of try's a is %p\n", &a);
   a = 15;
}

int main(){
   int a = 20;

   printf("The address of main's a is %p\n", &a);
   try_to_mod(a);
   printf("a = %d \n", a);
}
```