

# Mouse Picking with Ray Casting

Anton Gerdelan. Last Updated 2 October 2016


<https://antongerdelan.net/opengl/raycasting.html>

## Overview

It can be useful to click on, or "pick" a 3d object in our scene using the mouse cursor. **One way of doing this** is to project a 3d ray from the mouse, through the camera, into the scene, and then check if that ray intersects with any objects. This is usually called **ray casting**. This is an entirely mathematical exercise - we don't use any OpenGL code or draw any graphics - this means that it will apply to any 3d application the same way. The mathematical subject is usually called geometric **intersection testing**.

# 3d Transformation Pipeline

- create mesh
- has local origin



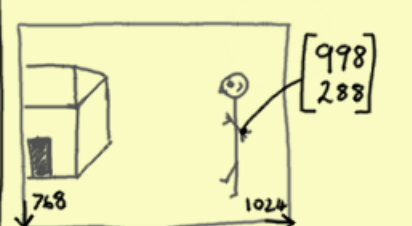
a vertex point  $\begin{bmatrix} 0.5 \\ 1 \\ 0 \end{bmatrix}$

\*origin

• e.g. character mesh

## LOCAL SPACE

- $xy -1:1 \rightarrow x,y$  pixels
- 2d (z used to sort)



$\begin{bmatrix} 998 \\ 288 \end{bmatrix}$

768 1024

## VIEWPORT SPACE

model matrix \*  
inverse model mat.

- meshes placed in world
- world origin

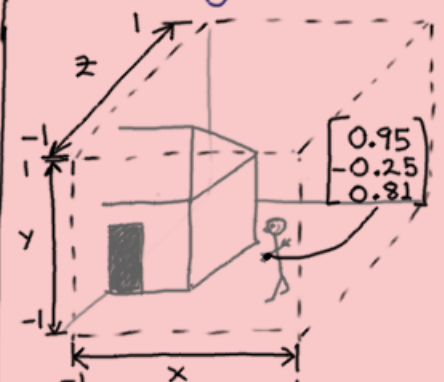


$\begin{bmatrix} 10.5 \\ -20 \\ 1 \end{bmatrix}$

\*origin

## WORLD SPACE

- frustum  $\rightarrow$  cube
- visible  $xyz$  is  $-1:1$




$\begin{bmatrix} 0.95 \\ -0.25 \\ 0.81 \end{bmatrix}$

## NORMALISED DEVICE SPACE

view matrix \*  
inverse view matrix

- camera is origin
- $xyz$  axes line up to cam. orientation



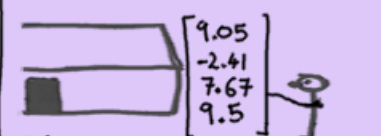
$\begin{bmatrix} 5 \\ -1 \\ -9.5 \end{bmatrix}$

+ origin at cam.

## EYE SPACE

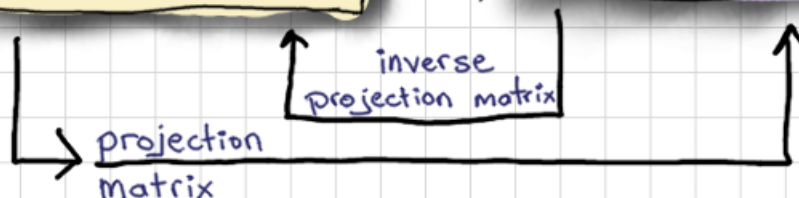
perspective  
division  $\frac{xyz}{w}$

- $w = -z$
- $xyz$  scaled into frustum shape
- $gl\_Position =$



$\begin{bmatrix} 9.05 \\ -2.41 \\ 7.67 \\ 9.5 \end{bmatrix}$

## HOMOGENEOUS CLIP SPACE



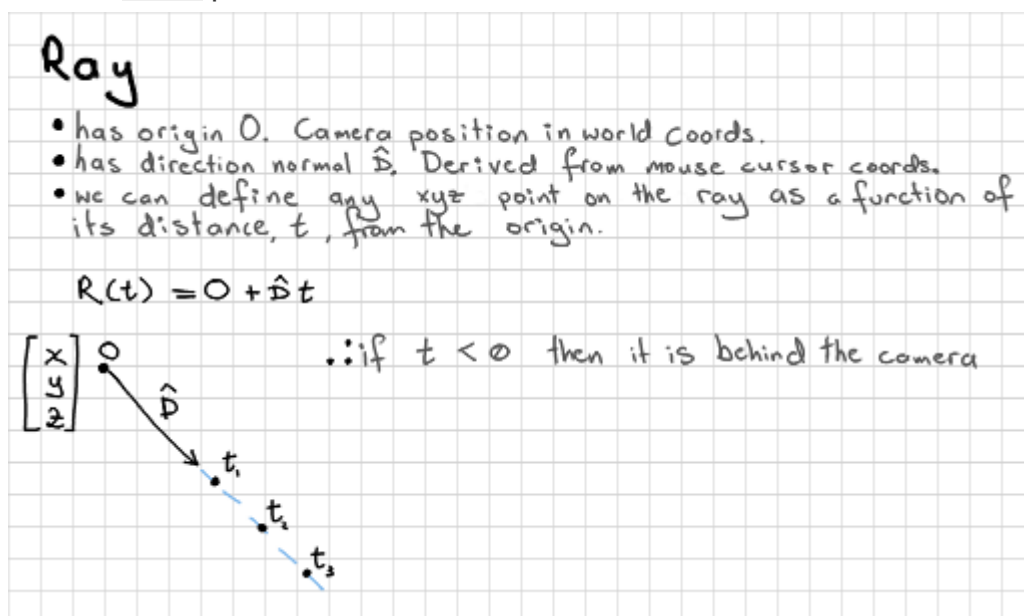
*Instead of starting with a mesh in local space, we are starting with a 2d mouse cursor position in viewport space. We work backwards through the transformation by using inverse matrices, and arrive with a ray in world space.*

With ray picking we usually simplify a scene into bounding spheres or boxes. This makes the calculation a bit easier than testing all of the individual triangles. We don't need to create a 3d sphere out of triangles that can be rendered; we just represent the sphere as a simple function. The premise is that we have a mathematical formula for the points along a ray, and a mathematical formula for the points on a sphere. If we substitute the points on the sphere with the equation for points on a ray, then we get the intersection of points that are common to both. It's interesting to do this technique now because it shows us how we can use the transformation pipeline in reverse; from 2d screen to a 3d world space by using the inverse of our matrices

e.g. `...inverse(view_matrix) * inverse(projection_matrix)...`. In a later tutorial we will look at an alternative technique using unique colours to determine where the mouse is hovering or clicking.

## Calculating a Ray from the Mouse

All ray casting starts with a ray. In this case it has an origin  $O$  at the position of the camera. We can do ray intersections in any space (world, eye, etc.), but everything needs to be in the same space - let's assume that we are doing our calculations in world space. This means that our ray origin is going to be the world  $x, y, z$  position of the camera.



*This function expresses all the points,  $t$ , along an infinite ray projected from our camera location,  $O$ , in the direction  $D$ , which we will work out by modifying and normalising the mouse cursor position.*

### Step 0: 2d Viewport Coordinates

range  $[0:\text{width}, \text{height}:0]$

We are starting with mouse cursor coordinates. These are 2d, and in the **viewport coordinate** system. First we need to get the mouse  $x, y$  pixel coordinates. You might have set up a call-back function (with e.g. GLFW or GLUT) something like this:

```
void mouse_click_callback(int b, int s, int mouse_x, int mouse_y);
```

This gives us an x in the range of `0:width` and y from `height:0`. Remember that 0 is at the top of the screen here, so the y-axis direction is opposed to that in other coordinate systems.

## Step 1: 3d Normalised Device Coordinates

*range [-1:1, -1:1, -1:1]*

The next step is to transform it into 3d **normalised device coordinates**. This should be in the ranges of x [-1:1] y [-1:1] and z [-1:1]. We have an x and y already, so we scale their range, and reverse the direction of y.

```
float x = (2.0f * mouse_x) / width - 1.0f;  
float y = 1.0f - (2.0f * mouse_y) / height;  
float z = 1.0f;  
vec3 ray_nds = vec3(x, y, z);
```

We don't actually need to specify a z yet, but I put one in (for the craic).

## Step 2: 4d Homogeneous Clip Coordinates

*range [-1:1, -1:1, -1:1, -1:1]*

We want our ray's z to point forwards - this is usually the negative z direction in OpenGL style. We can add a w, just so that we have a 4d vector.

```
vec4 ray_clip = vec4(ray_nds.xy, -1.0, 1.0);
```

**Note:** we do not need to reverse perspective division here because this is a **ray** with no intrinsic depth. Other tutorials on ray-casting will, incorrectly, tell you to do this. Ignore the false prophets!

We would do that only in the special case of points, for certain special effects.

## Step 3: 4d Eye (Camera) Coordinates

*range [-x:x, -y:y, -z:z, -w:w]*

Normally, to get into clip space from eye space we multiply the vector by a projection matrix. We can go backwards by multiplying by the inverse of this matrix.

```
vec4 ray_eye = inverse(projection_matrix) * ray_clip;
```

Now, we only needed to un-project the x,y part, so let's manually set the z,w part to mean "forwards, and not a point".

```
ray_eye = vec4(ray_eye.xy, -1.0, 0.0);
```

## Step 4: 4d World Coordinates

*range [-x:x, -y:y, -z:z, -w:w]*

Same again, to go back another step in the transformation pipeline. Remember that we manually specified a -1 for the z component, which means that our ray isn't normalised. We should do this before we use it.

```
vec3 ray_wor = (inverse(view_matrix) * ray_eye).xyz;  
// don't forget to normalise the vector at some point  
ray_wor = normalise(ray_wor);
```

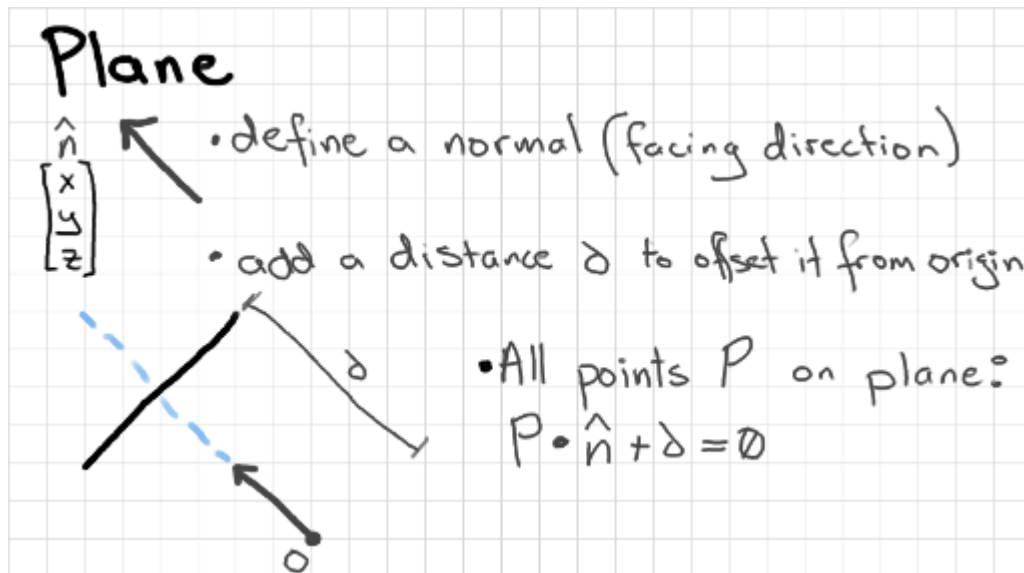
This should balance the up-and-down, left-and-right, and forwards components for us. So, assuming our camera is looking directly along the -Z world axis, we should get `[0, 0, -1]` when the mouse is in the centre of the screen, and less significant z values when the mouse moves around the screen. This will depend on the aspect ratio, and field-of-view defined in the view and projection matrices. We now have a ray that we can compare with surfaces in world space.

## Shortcut

If you're happy with this theory then you can condense all of these instructions into one line.

## Ray vs. Plane

We can describe any surface as a plane. It's often useful to do this before doing more detailed (and more computationally expensive) intersection tests. We can test, for example, if a point is inside a region bounded by planes - if not, then no need to check versus individual triangles inside the region. In my case, I wanted to create an imaginary ground plane that I could use to help me click-and-drag boxes around scenery. I do 2 ray-plane intersections to get the top-left and bottom-right corners of the box in xyz world coordinates.



## Ray vs. Plane

- Substitute points  $P$  on plane with points  $O + \hat{D}t$  on ray:

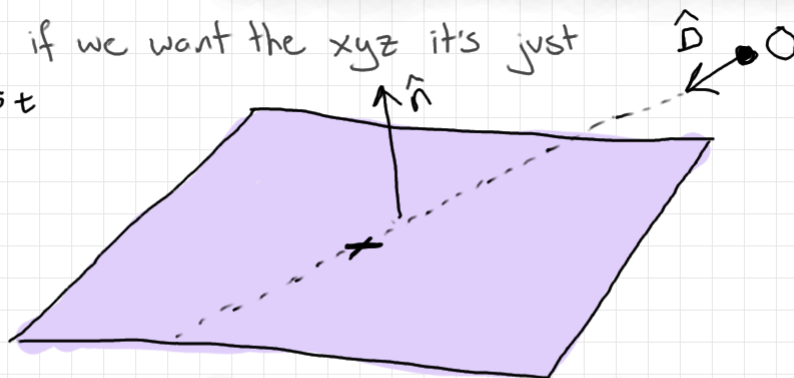
$$(O + \hat{D}t) \cdot \hat{n} + \delta = 0$$

(all points on ray **and** plane)

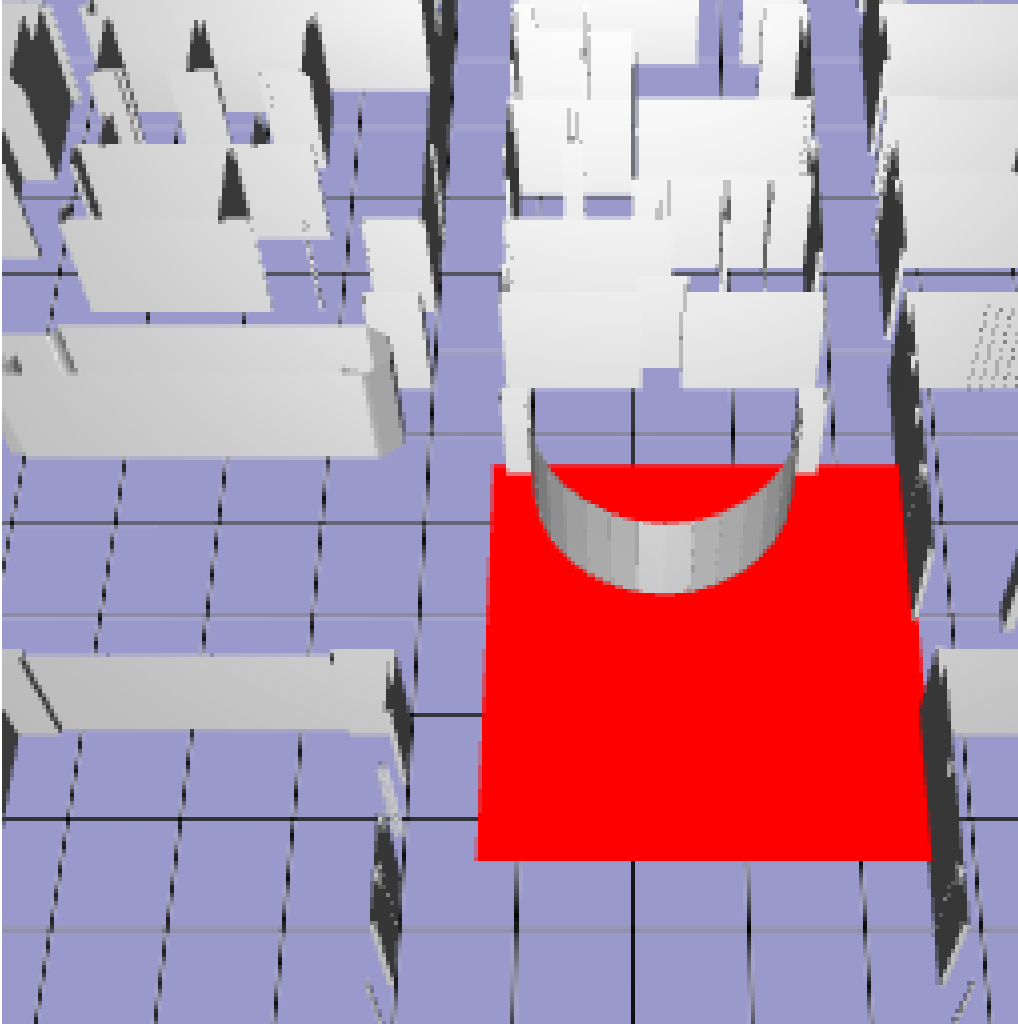
- We just want the distance  $t$  from ray origin:

$$t = - \frac{O \cdot \hat{n} + \delta}{\hat{D} \cdot \hat{n}} \quad \text{if zero then miss! (perpendicular)}$$

- then if we want the xyz it's just  $O + \hat{D}t$



\*if  $t < 0$  = miss (intersected behind  $O$ ).

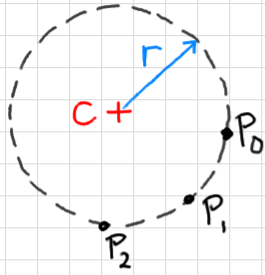


*I used ray-plane intersection to allow a user to drag out a "zone" on the floor. This might be useful in strategy games where you want to select a group of units. Unless you're on a globe. In that case, read on. Note that the square is projected into depth - it isn't a true square on the 2d screen, but will be if looking from directly above it.*

## Ray vs. Sphere

Probably the easiest method for selecting objects in a 3d scene. If you know a bounding radius, and centre point, of each object then we have the definition of its sphere. This might not be the best method to use for large, or irregularly shaped objects, which will have very large spheres that might overlap with smaller objects nearby.

# Sphere



All points 'p' on sphere:

$$\|P - C\| - r = 0$$

All points on sphere and ray:

(replace 'p' with ray equation)

$$\|O + \hat{D}t - C\| - r = 0$$

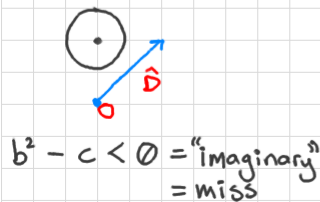
This re-arranges into a quadratic:

$$t^2 + 2tb + c = 0 \quad \left. \begin{array}{l} \\ \end{array} \right\} \text{quadratics have 2 solutions}$$

$$t = -b \pm \sqrt{b^2 - c}$$

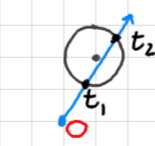
$$\text{where } b = \hat{D} \cdot (O - C) \\ c = (O - C) \cdot (O - C) - r^2$$

Case 1



$$b^2 - c < 0 = \text{"imaginary"} \\ = \text{miss}$$

Case 2



$$b^2 - c > 0$$

Case 3

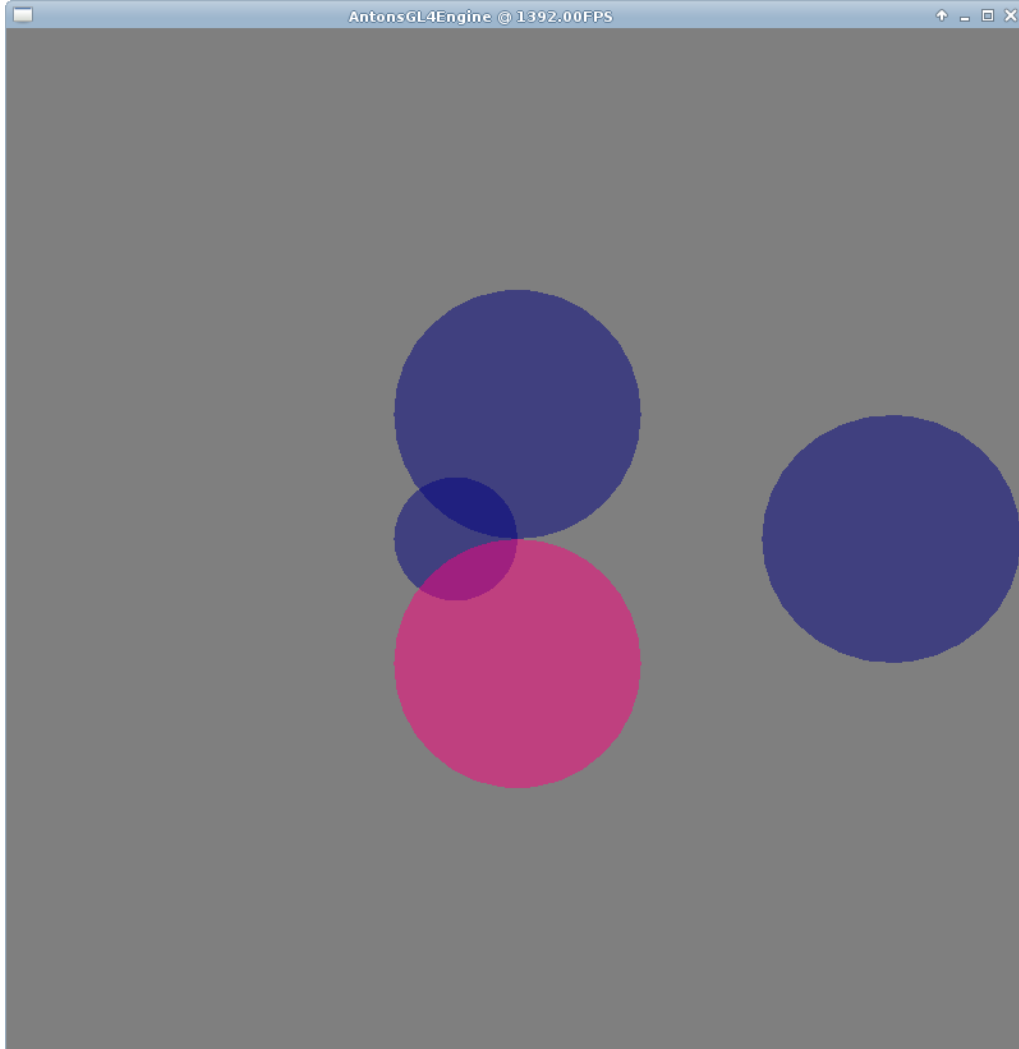


$$b^2 - c = 0$$

Here I have an equation for all the points,  $P$  on a sphere's surface, which says "anything that is radius  $r$  distance from centre point  $C$ ". The idea is to replace  $P$  with the points-along-a-ray equation, and solve for  $t$ . I skipped the re-arrangement steps, but if you're interested in a full discussion have a look at Akenine-Möller et al. [Real-Time Rendering](#).

We can actually get 2 output intersections from a ray-sphere intersection because the ray can hit both the back and the front of the sphere. In this case we need to check for the closest intersection (smallest value of  $t$ ). The ray can also miss the sphere, hit the very edge of the sphere (both  $t$  values are the same), or be cast from inside the sphere (one  $t$  value is negative).

Be sure to check the  $b^2 - c$  term before evaluating the expensive square root operator. If you miss the sphere entirely you will not get a sensible result for  $t$ . It is also possible that the ray will be cast from in front of or inside of the sphere, so you will need to check that your  $t$  values are positive.



*Here I have a collection of spheres in 3d space. I generated these in Blender and scaled them to match the radius of my bounding sphere. They are all the same size, but one is farther away and behind the others - this was to test that only the closest intersection was being selected. When a sphere is clicked on, I changed its colour to pink. This helped me determine that the ray was being calculated correctly, and the intersection was returning the correct result. You can imagine each sphere surrounding a mesh or object that is being selected.*

## Optimisations

You probably won't need to unless you're doing **a lot** of clicking on spheres, but you can reduce the cost of this algorithm by adding a few checks before getting to the square root. Compare squared distances between the ray origin and the sphere centre. You can project this as a 2d distance along the ray direction (dot product) and compare the end point's squared distance to the sphere origin (use Pythagoras' theorem) with the squared radius.

## Round-Up

Generally, if you want to draw on a shape (e.g. dragging boxes) then ray-casting is a good choice. If you want inaccuracy i.e. letting the user click in the general area of a fiddly little mesh to select it then ray-casting is a good choice. If you want accuracy then you're almost certainly better off using a colour-based picking method, which has the cost of one extra rendering pass, rather than 1-5 ray casts per selectable object in the scene.



# Further Things (That I Might or Might Not Add Later)

## Axis-Aligned Bounding Box (AABB)

You can do ray versus axis-aligned box intersection, which is quite commonly used. An object in the world is considered to have a bounding box, rather than a bounding sphere. This has the weakness that irregularly shaped, rotating objects will create gigantic boxes, but it is quite easy to work out the extents, and use these to validate a set of ray-plane intersections.

## Oriented Bounding Box (OBB)

If your shapes are actually boxes, or you want the box to rotate with the object inside, then you can do a ray versus oriented box intersection in much the same way. A few optimised methods exist (Haines "Essential Ray Tracing Algorithms", Chapter 2). I haven't had occasion to use this "in the wild".

## Barycentric

Ray versus triangle collisions will let you determine exactly where on a more complex mesh that you have clicked on. If you need this much accuracy, you are probably better off using the colour buffer method and getting a pixel-correct method. It might be useful for things like shooting games where you want put a bullet hole image in an exact location though. I haven't found a good reason to use this one in a while either. Before testing several different surfaces of a complex object, it's generally advised to do a ray versus bounding sphere test first - no point testing all of the triangles if the ray has missed the whole object!