Physics - Raycasting

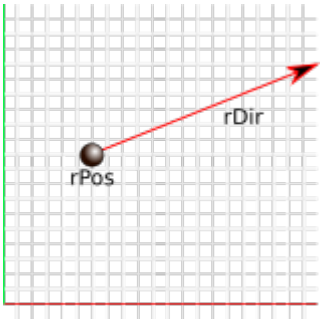
Introduction

A common operation when programming a video game is to determine whether an object can be seen or touched from a particular point - clicking on a button in a menu screen, or clicking on a tank in an RTS game are examples of this. Sometimes we may wish to determine whether an AI unit can 'see' another player unit from its current position. While these AI visibility checks and menu mouse clicks might not seem very similar on the surface, they are both often accomplished using the same process - *raycasting*. Raycasting allows us to fire an infinitely thin lines from a particular point in the world (whether a tank's turret or the player's mouse pointer), and see which objects it collides with along the way. Depending on what is collided with, we can then call whatever custon code we need to add interactivity to our game scenes.

In this tutorial we'll see how to define a ray mathematically, how to build rays that go in the direction we want, and look at the intersection tests that allow us to determine whether a ray passes through a simple shape or not.

Rays

A standard ray is formed out of a position in space, and a direction to travel through space in. We can think of this as being like a laser pointer - we position it, and then from it emit a laser that travels in a straight line to eventually hit something. In code terms, a ray is just two **Vector3** objects, one of which will be normalised to represent the direction of travel. We can imagine that our direction is an infinite line heading outwards from the ray's origin in space - the purpose of raycasting is to then see which objects in the world are intersected by this infinite line.



Rays from a transformation matrix

We can of course just define a ray anywhere in our simulation's 'world' and see what it hits, but usually we want to start our ray from some known point in the world, such as an existing object. Therefore, it's quite useful to be able to define a ray using an object's model matrix. Let's remind ourselves of what a model matrix contains:

$$\begin{bmatrix} 1 & 0 & 0 & p_x \\ 0 & 1 & 0 & p_y \\ 0 & 0 & 1 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Down the right we have the object's world space position, while the upper 3x3 matrix contains the world space rotation of the object. From this upper 3x3 region, we can determine which direction

the object is facing in. If we assume that -z is the 'forward' direction, (as in OpenGL coordinates), then we can form the 'forward' vector [0,0,-1] by negating either the third row or column of the upper 3x3 region - but which one is correct? We can determine this via examining another example model matrix, one which rotates the object so that it is instead looking 45° to the left:

$$\begin{bmatrix} x_x & y_x & y_z & p_x \\ x_y & y_y & z_y & p_y \\ x_z & y_z & z_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0.7071 & 0 & 0.7071 & p_x \\ 0 & 1 & 0 & p_y \\ -0.7071 & 0 & 0.7071 & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

From this, we can see that, when negated, that the matrix values $[x_z, y_z, z_z]$ give us the correct direction vector of $[0.7071, 0, -0.7071]$ to represent this new direction. From this, we can infer that from any object's model matrix, $[x_x, y_x, z_x]$ will point along that object's 'right axis' in world space, $[x_y, y_y, z_y]$ will face upwards, and $[x_z, y_z, z_z]$ will face forwards.

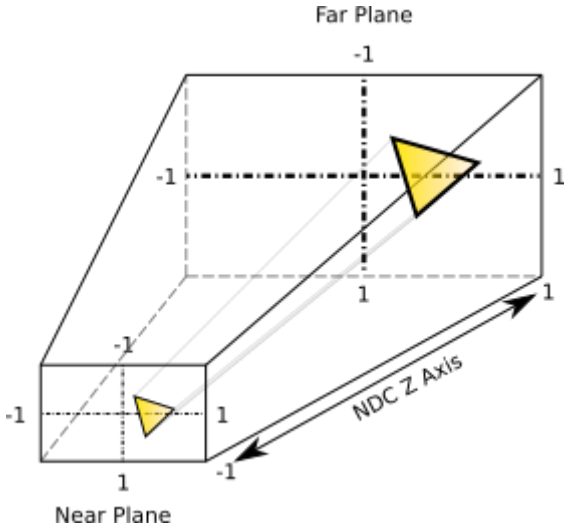
Rays from the mouse pointer

Sometimes, we might want to define a ray from the camera's point of view. One way to do this is using the *view matrix* - this isn't quite as straightforward as with the model matrix, though. Recall that the view matrix can be thought of as the 'inverse' of a model matrix, so to get the position and orientation of a view matrix, it must first be inverted to turn it into a model matrix. If we are sure that the view matrix has a uniform scale, then the *transpose* of the matrix will do just as well, so the 'forward' axis of a view matrix can be extracted using $[x_x, x_y, x_z]$ (remember, the *transpose* flips rows and columns, so if we already know what numbers to extract, we don't have to do the 'full' transpose).

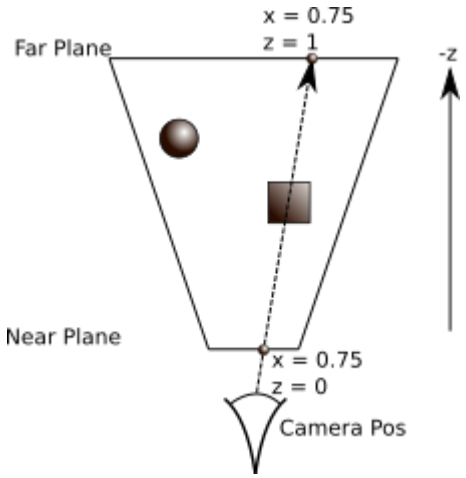
Usually we don't want to just fire out a ray from straight ahead, but click on something on screen - whether its a menu box in an RPG, or a tank factory in an RTS game, we in some way often need to be able to detect what's under the mouse pointer. If we're dealing with a 3D view, that means that we need to deal with the *projection* matrix, too - remember that a perspective matrix has a field of view which defines how far 'out to the side' we can see, and so how 'sideways' from the centre of the screen a ray coming from the camera can point.

Assuming we have the screen position of the mouse, we can work out where that position is in world space. If you remember back in the deferred rendering tutorial we did exactly the same thing - given a screen fragment position, we can get to a world space position by transforming the screen space position by the *inverse* of the view projection matrix, and then dividing by the 'inverse' w that we gain in the matrix multiplication. To put this another way, we are *unprojecting* the position back into world coordinates. That gives us a position, but how to get a 'direction'?

To work this out, it's worth taking a look at the view frustum formed by the view projection matrix:



Think back again to the deferred rendering tutorial - it wasn't just the screen x and y coordinates we used to perform our unprojection, we also made use of the depth sample, giving us a z coordinate, too. To form a ray from a screen position, we can perform this unprojection process on two positions, each with the same x and y coordinates, but different z axis coordinates; a simple subtraction and normalisation of these two positions should give us a ray. But which two z axis coordinates to use? As we should be familiar with by now, we have a 'near plane' and a 'far plane' bounding everything visible in a particular direction. In OpenGL, and with the matrices we've been using, this near plane maps to an NDC coordinate of -1 on the z axis, while the far plane maps to the NDC of +1. Therefore, if we unproject the NDC coordinates [x,y,-1] and [x,y,1] into world coordinates, we can form a direction vector in world space that travels straight through a point on screen:



In this example, we've clicked on the screen at approximately 75% of the way along the screen on the x axis; if we then unproject this coordinate at a depth of -1 and 1 using the inverse of the view projection matrix, we end up with world space coordinates, once they've been divided by the 'inverse' w they gain during the transformation. From these coordinates, a ray can be determined, and in this particular case, we can then see that the ray travels through the cube, but misses the sphere.

Calculating an inverse view matrix

The process of unprojection requires the inverse of the view projection matrix - even for a 4x4 matrix, the process of inversion is quite a costly process. For both the view and projection matrices, we know what values formed them (or can at least store them somewhere in our classes), which allows us to instead derive a matrix that will do the same as an inverted matrix, without requiring us to do the general purpose inversion process on our matrices.

In the **Camera** class that we've been using to manipulate the view matrix since the previous module, we've been separately storing a pitch, yaw, and position, to make it easier to change these values as keys are pressed and the mouse moved. We have then been generating the view matrix when required using the **BuildViewMatrix** method, which does the following:

```
1 Matrix4 Camera::BuildViewMatrix() const {
2    return   Matrix4::Rotation(-pitch, Vector3(1, 0, 0)) *
3             Matrix4::Rotation(-yaw, Vector3(0, 1, 0)) *
4             Matrix4::Translation(-position);
5 }
```

If we flip the order of these matrix multiplications, and use the *pitch*, *yaw*, and *position* member variables (note, the **BuildViewMatrix** method already negates the member variables, as they are already defined in world space and must be 'inverted' to get us a view matrix), then we can get a model matrix (and the position member variable gives us a ray origin point, too!):

```
Matrix4 cameraModel = Matrix4::Translation(position)  *
         Matrix4::Rotation(yaw   , Vector3(0, 1, 0))   *
         Matrix4::Rotation(pitch , Vector3(1, 0, 0));
```

Calculating an inverse projection matrix

Calculating the inverse of a projection matrix is a little more involved, but all we need are the same variables we define a projection matrix with - an aspect ratio, a field of vision, a near plane, and a far plane. Here's a reminder of what the perspective projection matrix looks like:

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zNear+zFar}{zNear-zFar} & \frac{2 \cdot zNear \cdot zFar}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Where f is the *cotangent* of the field of vision. If we more generically define the projection matrix as:

$$\begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & e & 0 \end{bmatrix}$$

We can see that parts a and b do not interact with any other axis, as they're down the diagonal, and there's nothing else on their row / column. Therefore, to 'undo' whatever affect they have on a vector during transformation, we can use their reciprocal. Parts d and e are slightly more tricky - part d gives us an interaction between the z and w axes, while part e maps the z axis of the incoming vector onto the result vector. To 'undo' these, we not only have to take their reciprocal, but also transpose them to map the values back onto their original axis, so part e goes from row 4 column 3, to row 3 column 4, while part d goes the other way. Part c is the trickiest to undo - in a projection matrix it's used to scale the z value written to the depth buffer which defines how far away a fragment is, but as this is the value we're now providing, we need some way in which to scale all the other coordinates by it to 'invert' its use. To do this, part c needs to instead go into the w axis of the resulting vector, so we can further divide the by 'inverse w', stretching out all of the coordinates from NDC space back into world space. We also need to undo the original translation that would be added to the z axis, so this goes in our w result, too - altogether we get the following matrix to invert the projection:

$$\begin{bmatrix} \frac{1}{a} & 0 & 0 & 0 \\ 0 & \frac{1}{b} & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{e} \\ 0 & 0 & \frac{1}{d} & -\frac{c}{de} \end{bmatrix}$$

We'll see how this relates back to the specific values we calculate for a perspective projection matrix in the tutorial code. A more complete derivation of the inverse of a projection matrix can be found on Brian Hook's website at:

http://bookofhook.com/mousepick.pdf.

Collision Volumes

Forming a ray is one thing, but we also need to be able to work out what that ray is intersecting. We could if we want test every triangle of an object's mesh against a ray, as there's a common calculation for determining whether a ray intersects a triangle, but that's generally overkill for many purposes - we don't care if we've clicked on a tank's turret or tracks, we just want to work out whether we've clicked on a particular unit in an RTS game. It's usually the case that we instead try to work out if we've intersected against a rough shape that encapsulates the object we want to raycast against, such as a sphere or a box shape. These rough approximations will be used throughout the tutorial series, to allow us to calculate the physical interactions between objects in an efficient manner, so performing raycasts against these shapes serves as a neat introduction to them.

Planes

We've already used one collision shape, although it's not really a *volume* as such. Throughout the graphics module we've seen how the view frustum can be represented as a set of 6 planes, each of which divides space into two sub-spaces, somewhat like an infinitely large, infinitely thin wall. So we can form volumes out of them (such as our view frustum), and planes often form part of the collision tests we'll be looking at in these tutorials.

We can represent a plane using 4 values, with the classic plane equation being as follows:

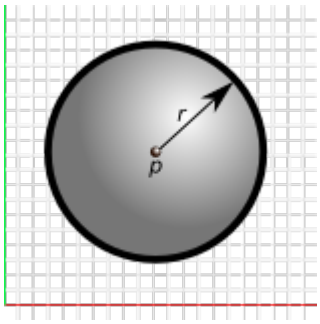
$$\mathrm{Ax} + \mathrm{Bx} + \mathrm{Cx} + \mathrm{D} = 0$$

Any point in space at the positions x,y,z is considered 'on' the plane if the dot product between plane values $ABCD$ and $[x,y,z,1]$ equals 0. Additionally, we can consider a point inside (or in front of) the plane if the result of the dot product is greater than 0, and outside (or behind) the plane if the dot product is less than 0.

Planes are often used in games as absolute bounds on a game world - sometimes bugs in the game level mean that a player can escape the game world and fall through the floor, so game developers often place 'kill planes' below the floor, which are hooked up to code that will automatically kill the player if their world position is on the 'wrong' side.

Spheres

We can represent a sphere using four values, too - a vector position p representing the middle of the sphere, and a radius r. Any point in space that is a distance of less than r away from x,y,z is therefore 'inside' the sphere, and if it's equal it is on the surface of the sphere.

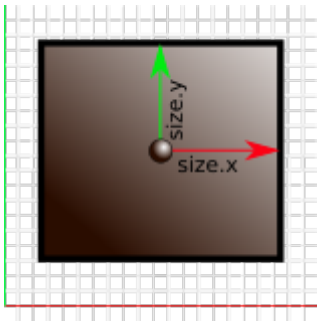


Boxes

Boxes are a little bit more complex than spheres. There's two types of box used for collision detection (sometimes called *bounding boxes*) - *Axis Aligned* bounding boxes, and *Object Oriented* bounding boxes.

AABBs

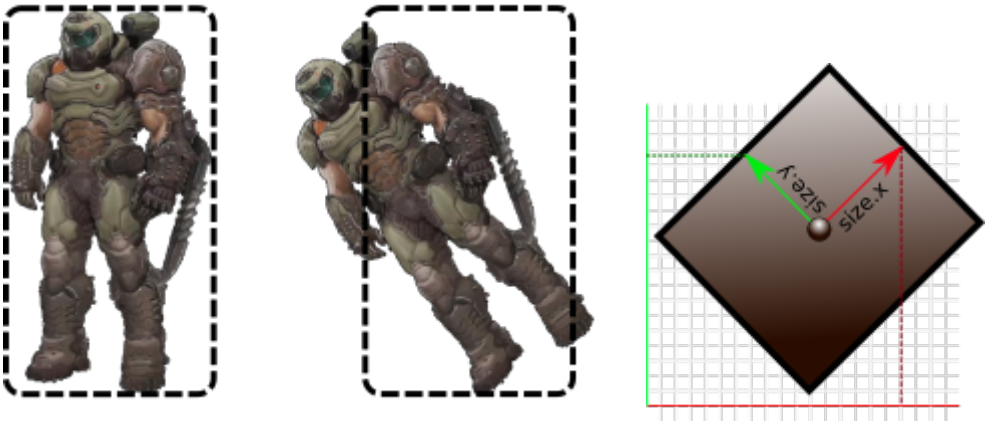
An *axis aligned* bounding box has a position in 3D space, and also a size, defining the length, width, and height of the box. Usually these are stored as 'half sizes', meaning the distance from the middle of the box to the edge:



No matter what orientation the object being represented with an AABB is, the AABB never rotates - so its x axis always points along world space x, and the same for y and z. This makes AABBs somewhat simpler to perform calculations on, as the box and whatever other construct we're testing it with (either a ray as in this case, or another collision volume to detect collisions between pairs of objects) will always be in the same 'space'.

OBBs

The problem with representing the bounding volume of an object with an AABB is that as the orientation of a shape changes over time (due to player movement or physics forces being applied to it) the box size will not reflect the new extents of the object in the world axes:



As the shape above rotates, the AABB of its collision volume matches up progressively worse, so any collision detection or raycasting performed on it will be inaccurate. We can extend the idea of a box with an origin and half size to also have an orientation, by storing the half size as a vector, and then transforming it by the upper 3x3 region of an object's model matrix - making an *oriented bounding box*. So why do we bother with AABBs if OBBs match the shape of the objects better? As we'll see over the course of this tutorial series, some of the properties of AABBs make them much easier to detect collisions between, so unless the extra accuracy is required, AABBs are often preferred over OBBs.

Ray Intersections

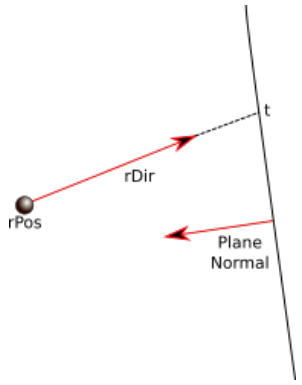
While the above collision shapes aren't the only common ones, they do make a good starting point in understanding how collisions between shapes and rays work. Let's now have a look at some of the algorithms for determining whether a ray has intersected any of these shapes, and what information we can get from them.

Ray / Plane Intersections

The easiest intersection test to calculate is between a ray and a plane. Unless the normal of the plane and the direction of the ray are facing in the same direction, there will *always* eventually be an intersection between a ray and a given plane, which can be calculated like so:

$$t = \frac{-(Ray_{pos} \cdot Plane_{abc} + plane_d)}{Ray_{dir} \cdot Plane_{abc}}$$
$$p = Ray_{pos} + t(Ray_{dir})$$

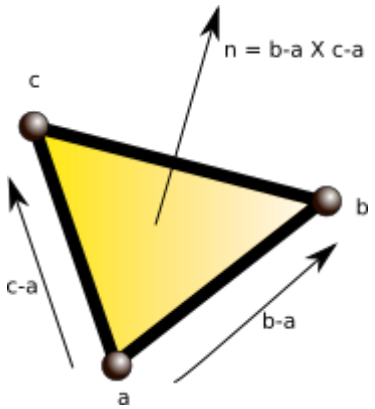
This allows us to get the length along the ray to the intersection t, and from there, the actual intersection point p. When coding a ray/plane intersection, you may wish to calculate $Ray_{dir} \cdot Plane_{abc}$ first, as this will check for orthogonality between the plane normal and the ray direction, so you can avoid a division by 0 in these cases.



Ray / Triangles

There are cases we might want to perform ray intersection tests against a triangle - some games have realistic 'bullet hole' decals applied to models when they have been fired on, the point of which can be determined by raycasting along the bullet trajectory.

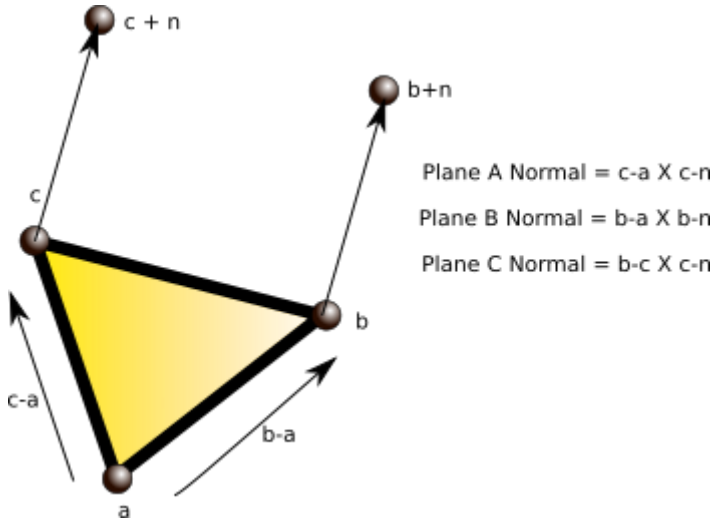
The first step in performing ray / triangle intersection is to form a plane out of the triangle - triangles are always planar surfaces, so we can use this to make the calculation simpler. To calculate the plane of a triangle, we need the normal of the triangle, which as you may remember from the previous module, can be computed from two vectors and a cross product:



The triangle normal gives us 3 plane coefficients $Plane_{abc}$, while performing the dot product on any point of the triangle and the normal gives us $plane_d$. From there, we can perform the ray / plane intersection described earlier to find the point on the 'triangle plane' the ray intersects at.

From there, we can form another 3 planes - instead of going along the surface of the triangle, each plane skims along one of the edges of the triangle. Only if the point on the triangle plane is inside each of these planes is a point considered inside the triangle. If the point *is* inside the plane, then we

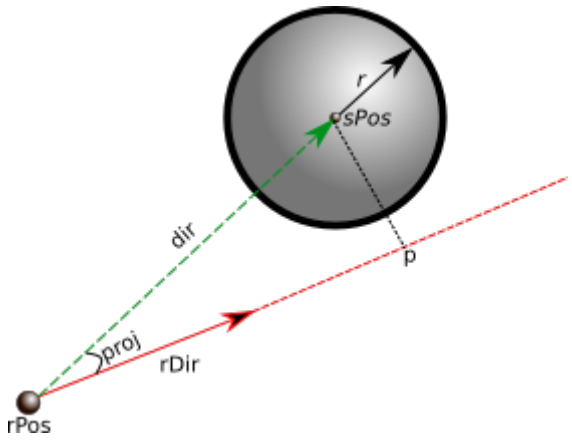
have the intersection point, and distance from the ray origin, just as before. To form the extra planes, we can determine extra points on them by moving along the triangle normal from any corner - that will give us 3 points to use just as we did to get the triangle normal above.



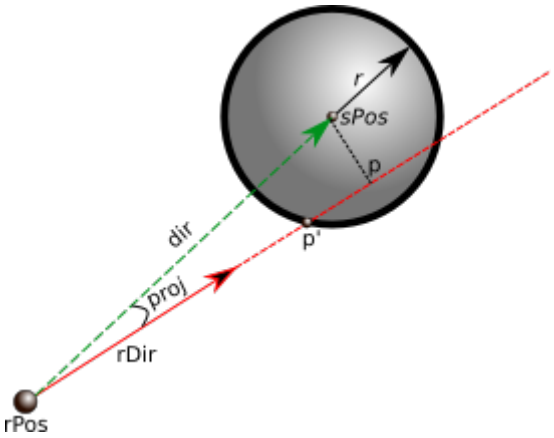
Ray / Sphere Intersections

Ray / sphere intersection is a little different from against triangles, but not too difficult. In this case, we're going to calculate the closest point along the ray to the sphere s by projecting the sphere's origin onto the ray direction vector - this sounds hard, but it's just a dot product and some vector multiplication:

$$dir = s_{pos} - r_{pos}$$
$$proj = r_{dir} \cdot dir$$
$$p = r_{pos} + proj(r_{dir})$$



Once we have this point, we can quickly determine whether the ray intersects the sphere by calculating the distance from the sphere origin to point p - if it's less than the sphere's radius, the ray intersects the sphere. In this case, if we move along *rDir* by *proj* units to get the closest point p, we can see that it's greater than the sphere's radius away, and thus our ray misses the sphere.
This can tell us whether the ray intersects or not, but we often also need to know how far along the ray the intersection point is. You might think the distance between rPos and p would give us the correct answer, but this isn't quite true:



When the ray *does* intersect the sphere, the above calculation to return p still gives us the closest point between the sphere's *origin* and the ray direction - so in this case p ends up inside the sphere. To calculate the actual ray intersection distance d, we need a further calculation:

$$\mathrm{d} = ||p - r_{pos}|| - \sqrt{r^2 - (||p - s_{pos}||^2)}$$

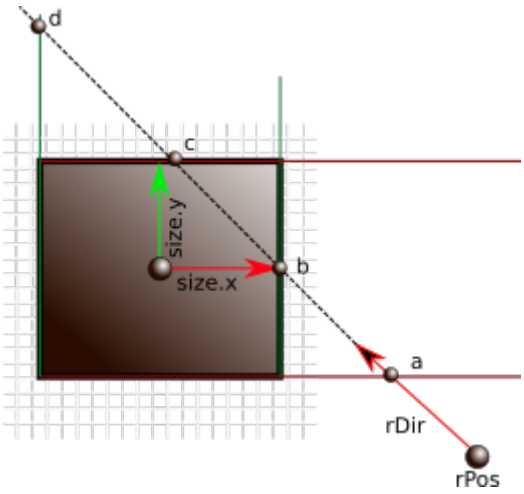
That is, we get the distance between the intersection point and the ray, and subtract the radius squared - the distance between point p and the sphere's origin. We can then find the true intersection point p' by travelling along vector $rDir$ by a distance of d.

Ray / Box Intersections

Detecting collisions between boxes and rays again reduces down to plane intersection tests. Much as with our view frustum in the previous module, we can represent a box using 6 planes to form an encapsulated volume.

AABB intersection

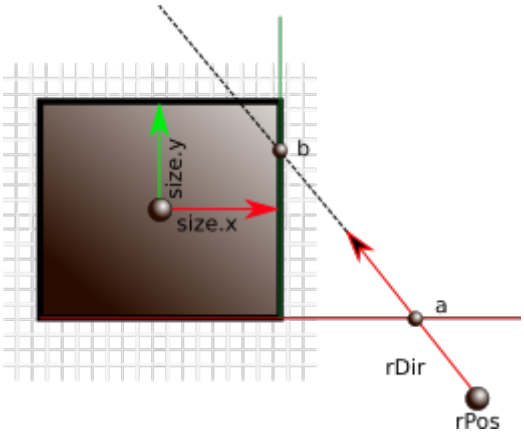
To perform ray intersection against an axis aligned bounding box, we could calculate the intersection point against all 6 planes that make up the box, and work out if any of them are on the surface of the box. From our definition of a box earlier of having a position and a half size, if we subtract the position of the point in space we wish to test from the box point, the test point must be inside the box if the resulting vector has an absolute position of less than the box's half size on each axis.



In this case, points a and d (representing the point at which the ray intersects the 'left' and 'bottom' plane of the AABB) are outside of the surface of the AABB, but point b and c (representing the intersections along the 'right' plane and 'top' plane) *do* touch the surface (point b is exactly $size.x$ units away from the cube's origin on the x axis, and point c is equal to $size.y$ units on the y axis), and thus we know there's an intersection - if we really need to know the exact intersection point, we need

to determine whether point b or point c is closer to $rPos$, by calculating the length of the vectors b-$rPos$ and c-$rPos$, and picking the shorter length - in this case, point b should be the intersection point.

While we *could* test all 6 sides, it's actually unnecessary. If a ray is coming in from the right of a box, then the ray will *always* hit the right side plane before the left side plane:

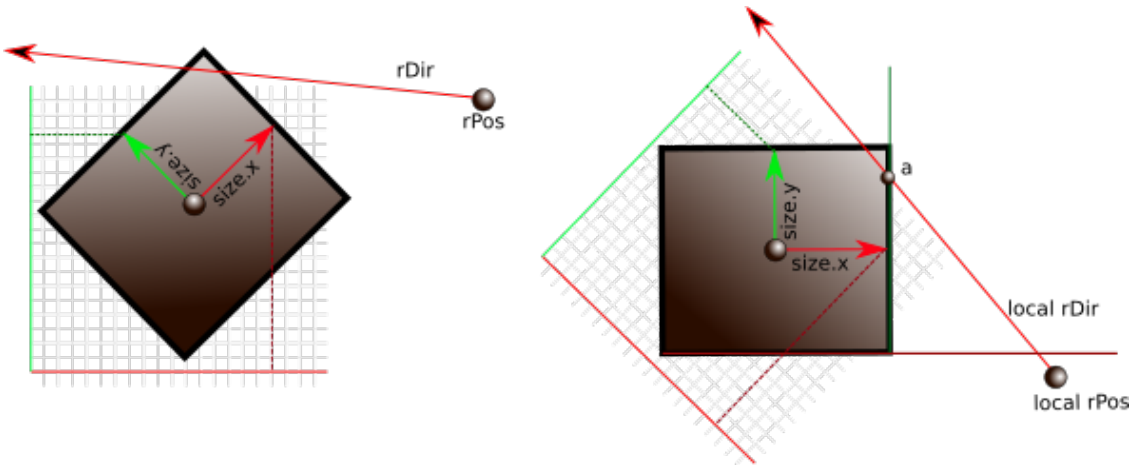


We can therefore use this to just test against 3 planes instead of 6, by using the values of the ray's normalised direction vector - for each axis, if the ray direction is positive we check the 'negative' plane (we subtract the size from the origin), and if it is negative we test the 'positive' plane (we add the size to the origin). With the number of tests reduced to three, we can now see that the *furthest* intersection point will now be the 'best' choice, as we're skipping any planes that would be hit once the ray had left the box. In the example above, even though point 'a' is closer, we can skip checking it, as point b is further along the ray. As long as the furthest intersection is on the surface of the AABB (it's no further away on the relevant axis than the box's size), then we have our collision point.

OBB intersection

Oriented bounding boxes are trickier. While we can get the plane offsets of an AABB via simple addition and subtraction of single values because we know the planes will be axis aligned - the 'right hand side' of a box will *always* have an offset of [some distance,0,0] from its position. If we're dealing with a *rotated* box, though, this is no longer true, and we have to determine the correct direction vectors that point forwards, upwards, and to the right of the object (or in other terms, the world space mapping of the object's local space axes). We can extract the x,y, and z axes out of the transformation matrix of the object as described earlier on, but we'll now have to test 6 planes after all as it's now much harder to determine the 'best' 3 planes. It's also harder to then test whether the 'best' point is inside the box after projection for the same reasons - we can't just check one number per axis any more!

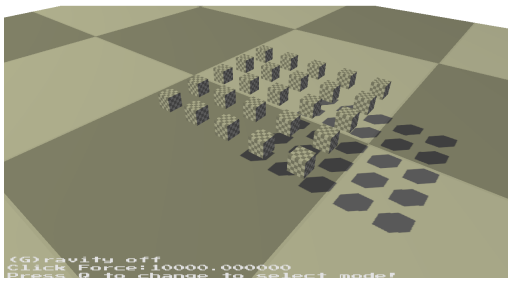
In this case, it's better to think about things in a different way - we can temporarily transform the position and direction of the ray by using the inverse of the object's transformation matrix, turning the OBB test in world space into an AABB test in the box's local space:



This gives us an intersection point a in the *local space* of the object, so it'll need to be transformed by the object's model matrix again to get it back in 'world space' - that's only if we need the intersection point, if all we need to know is whether the ray is intersecting or not, we can just leave the point in local space. This gives you some indication as to the difficulty in using oriented bounding boxes, which only gets more complex as we start looking at object collisions later. This is also an introduction to a common operation we do in physics calculations - bringing some position a into the 'local space' of some object b. If all we care about is a position, we can just subtract a from the position of b to get a's relative position. If, however, we need to know rotation as well, we must use the inverse of the object's model matrix, as a transform by this matrix will bring us back to the local space of the object.

Tutorial Code

Our example program to demonstrate raycasting is going to be fairly simple; just enough to get a feel of how raycasting can be used, and integrated into a game codebase. The **TutorialGame** class provides us with a default scene consisting of cubes and spheres - perfect for testing some raycasting code with! Here's how the 'game' looks when we start the program:



Everything has been already set up to build a list of objects to render on screen every frame. What the code *doesn't* really have, though, is a working physics engine, or any way of performing associated tasks like ray casting. It has a fairly empty class called **PhysicsEngine**, and a namespace called **CollisionDetection**, though, and the physics tutorials of this module will be filling those in, to give us an insight into how to build up a working physics simulation in our games.

Collision Volumes

Being able to represent a collision object in our code is clearly very important - the raycasting techniques above use collision shapes of varying types, and as the tutorial series progresses we'll be looking at how to detect collisions between them, too. In the provided codebase, each of the volumes we'll be testing against will derive from the **CollisionVolume** class, as shown here:

```
#pragma once
namespace NCL { //keep track of yournamespaces!
   enum class VolumeType {
      AABB     = 1,  OBB      = 2, Sphere = 4,  Mesh     = 8,
      Compound = 16, Invalid  = 256
   };
   class CollisionVolume   {
   public:
      CollisionVolume() {
         type = VolumeType::Invalid;
      }
      ~CollisionVolume() {}
      VolumeType type;
   };
} //end of namespace!
```

CollisionVolume Class

There's not much to it, other than storing an **enum**, which we can use to determine what type a derived class is, without doing any virtual methods, or dynamic casting. To see how this is used to represent one of the actual collision volume types we covered earlier, here's how the provided **SphereVolume** class looks:

```
#pragma once
#include "CollisionVolume.h"

namespace NCL {
    class SphereVolume : CollisionVolume   {
    public:
        SphereVolume(float sphereRadius = 1.0f) {
            type      = VolumeType::Sphere;
            radius    = sphereRadius;
        }
        ~SphereVolume() {}

        float GetRadius() const {
            return radius;
        }
    protected:
        float radius;
    };
}
```

SphereVolume Class

It has a radius, but no stored position. This is because we'll instead be using the world position of a transformation matrix, stored inside a '**Transform**' class, which will itself be part of a **GameObject** - these classes were covered more in the introduction tutorial, so we won't go over them again here. The other collision volumes in the codebase follow much the same pattern, so get familiar with it.

Ray Class

To represent the concept of a ray in C++, we have the appropriately named *Ray* class. There's not too much to it:

```
class Ray {
        public:
            Ray(Vector3 position, Vector3 direction);
            ~Ray(void);

            Vector3 GetPosition()  const { return position;  }
            Vector3 GetDirection() const { return direction; }

        protected:
            Vector3 position;     //World space position
            Vector3 direction;    //Normalised world space direction
        };
```

Ray Class header

We were already introduced to the **Vector3** class in the previous module, and we just need two of those to represent our ray; one for position, and one for its direction - this vector is always assumed to be normalised. Other than getters, that's all we need for a ray. Things get a bit more interesting to represent a ray *collision*, though. For this, you've also been provided with a templated *RayCollision* class, shown here:

```
13      template<typename T>
14      struct RayCollision {
15         T* node;              //Node that was hit
16         Vector3 collidedAt;   //WORLD SPACE pos of the collision!
17         RayCollision(T*node, Vector3 collidedAt) {
18            this->node        = node;
19            this->collidedAt  = collidedAt;
20         }
21         RayCollision() {
22            node = nullptr;
23         }
24      };
```

Ray Class header

The templating is just to abstract away the idea of a ray hitting something from the exact implementation of our code (as a ray is quite a useful construct, we can make it more reusable by keeping it separate from how exactly it is to be used). We'll be using it to store pointers to **GameObjects**, along with a world space position where the collision occurred.

Collision Detection Class

To actually use the **Ray** class, and perform intersection tests between rays and our collision volumes, we're going to add some functionality to the **CollisionDetection** class, which currently doesn't do much, but does provide us with some functions to calculate inverse projection and view matrices, and 'unproject' screen positions to world space. It also has a set of ray intersection functions which take in a a ray, transform, and bounding volume, and return either **true** or **false** depending on whether there's been a collision or not. To start off with though, each of these functions just returns **false**, so they don't do anything!

To get raycasting working we'll have to fill these functions out with some proper functionality. We'll start with the function *RayIntersection*, the purpose of which is to get the type of a passed in volume, and call the appropriate ray intersection function to see if the passed in ray collides with it. Here's the code for the *RayIntersection* function:

```
1  bool CollisionDetection::RayIntersection(const Ray& r,
2     GameObject& object, RayCollision& collision) {
3     const Transform& transform       = object.GetConstTransform();
4     const CollisionVolume* volume    = object.GetBoundingVolume();
5     if (!volume) {
6        return false;
7     }
8     switch (volume->type) {
9        case VolumeType::AABB:     return RayAABBIntersection(r,
10          transform, (const AABBVolume&)*volume  , collision);
11       case VolumeType::OBB:      return  RayOBBIntersection(r,
12          transform, (const OBBVolume&)*volume   , collision);
13       case VolumeType::Sphere:   return  RaySphereIntersection(r,
14          transform, (const SphereVolume&)*volume   , collision);
15    }
16    return false;
17 }
```

CollisionDetection::RayIntersection

Not much to it, but it helps us see how the **GameObject** class holds a **Transform** and a **CollisionVolume** (lines 3 + 4). Every **GameObject** in this codebase is assumed to have a transform, so the accessor to this object will return a *reference*, while the object might not be collideable with, and so might not have a **CollisionVolume**, so the accessor instead returns a *pointer*, which we can

check against, and return false if there isn't one, as we can't possibly collide with it (lines 6 - 8). If the passed in **GameObject** *does* have a volume, we can **switch** against its *type* variable, and call the appropriate intersection function. This involves casting the **CollisionVolume** to the correct sub class - as long as we don't modify the *type* variable set in the subclass constructors this will always hit the 'correct' switch statement for the actual collision type.

Ray / Sphere Intersection Code

Great! Now to see how the theory behind ray casting can be implemented in C++. We'll start with the easiest, and look at Ray / Sphere collisions, which should be implemented in the *RaySphereIntersection* function, detailed here:

```
1  bool CollisionDetection::RaySphereIntersection(const Ray&r,
2     const Transform& worldTransform, const SphereVolume& volume,
3     RayCollision& collision) {
4     Vector3  spherePos    = worldTransform.GetWorldPosition();
5     float sphereRadius    = volume.GetRadius();
6
7     //Get the direction between the ray origin and the sphere origin
8     Vector3 dir = (spherePos - r.GetPosition());
9
10    //Then project the sphere's origin onto our ray direction vector
11    float sphereProj = Vector3::Dot(dir, r.GetDirection());
12
13    if(sphereProj < 0.0f) {
14       return false; //point is behind the ray!
15    }
16
17    //Get closest point on ray line to sphere
18    Vector3 point  = r.GetPosition() + (r.GetDirection() * sphereProj);
19
20    float sphereDist = (point - spherePos).Length();
21
22    if (sphereDist > sphereRadius) {
23       return false;
24    }
25
26    float offset =
27       sqrt((sphereRadius * sphereRadius) - (sphereDist * sphereDist));
28
29    collision.rayDistance   = sphereProj - (offset);
30    collision.collidedAt    = r.GetPosition() +
31             (r.GetDirection() * collision.rayDistance);
32    return true;
33 }
```

CollisionDetection::RaySphereIntersection

It's a pretty direct implementation of the theory - we work out the direction vector between the ray and the sphere's centre (line 8), and then use the dot product operator to project this vector against the ray's direction vector (line 11) - this let's us see how far along the direction vector we can travel before we 'join up' to the end of the other vector (line 18). If that projected point is greater than the sphere's radius, the ray can't be colliding (line 19), otherwise, we determine the collision point by moving the collision point back along the direction vector, so that it touches the surface of the sphere, rather than being inside it. Line 14 covers the case where the sphere is *behind* the ray - the dot product between the ray direction and the direction vector between the ray and object will in this case end up being negative, and we should not consider the object any further.

Ray / AABB Intersection Code

Next, we're going to take a look at ray collisions with boxes. First we're going to write the function that'll perform intersections between a box and a ray. Add in this code to the *RayBoxIntersection* function in the **CollisionDetection** class file:

```
1  bool RayBoxIntersection(const Ray&r, const Vector3& boxPos,
2              const Vector3& boxSize, RayCollision& collision) {
3     Vector3 boxMin = boxPos - boxSize;
4     Vector3 boxMax = boxPos + boxSize;
5
6     Vector3 rayPos = r.GetPosition();
7     Vector3 rayDir = r.GetDirection();
8
9     Vector3 tVals(-1, -1, -1);
10
11    for (int i = 0; i < 3; ++i) { //get best 3 intersections
12       if (rayDir[i] > 0) {
13          tVals[i] = (boxMin[i] - rayPos[i]) / rayDir[i];
14       }
15       else if(rayDir[i] < 0) {
16          tVals[i] = (boxMax[i] - rayPos[i]) / rayDir[i];
17       }
18    }
19    float bestT = tVals.GetMaxElement();
20    if(bestT < 0.0f) {
21       return false; //no backwards rays!
22    }
```

CollisionDetection::RayBoxIntersection

To perform the box test, we're going to use the 'reduced' case that only checks the 3 closest box planes, rather than all 6. To do this, we're going to check the direction of the ray - if it's going left, we check only the right side of the box, if its going up we only check the bottom side of the box, and if its going forward we only check the back side of the box. As the box is axis aligned, we only need to check each individual axis of the ray direction, along with either the minimum or maximum extent of that box along the axis - that's why on lines 13 and 16 we check against [i], which will give us either the x, y or z axis of a vector. The resulting lengths along the vector are then stored in another **Vector3**, *tVals*. This allows us to use the *GetMaxElement* member method of the vector, which as its name suggest will give us the float with the greatest magnitude. We initialise the *tVals* vectors with negative values - if the maximum element within it is still negative after the **for** loop has concluded, then the intersection is actually *behind* the ray, and should be ignored (via the **return** on line 22). From the maximum value, we can then determine the best intersection point along the ray vector, and store it in the *intersection* variable, and work out whether its actually on the surface of the box or not:

```
23    Vector3 intersection = rayPos + (rayDir * bestT);
24    const float epsilon = 0.0001f; //an amount of leeway in our calcs
25    for (int i = 0; i < 3; ++i) {
26       if (intersection[i] + epsilon < boxMin[i] ||
27           intersection[i] - epsilon > boxMax[i]) {
28          return false; //best intersection doesn't touch the box!
29       }
30    }
31    collision.collidedAt    = intersection;
32    collision.rayDistance   = bestT;
33    return true;
34 }
```

CollisionDetection::RayBoxIntersection

The for loop on line 25 just goes through each axis, and works out whether the intersection point is too far to one side or the other of the box, determine by the box's minimum and maximum extents along that axis. Note that we're using a slight error bound (called an *epsilon*) to accommodate for slight variations in floating point accuracy - we don't want a point that is 0.0001 units away from a cube to count as 'not intersecting' if this distance was just due to how floating points operate. If it *does* collide, we can fill in our collision details directly, and return **true**.

Once we have the *RayBoxIntersection* function in place, we can use it for both AABB and OBB ray collisions. AABB collisions are calculated with the *RayAABBIntersection* function, which is pretty much just a 'pass through' to the function we've just written, as all it needs to do is get the box size and position from the AABB, and use them as parameters:

```
1  bool CollisionDetection::RayAABBIntersection(const Ray&r,
2           const Transform& worldTransform,
3           const AABBVolume& volume, RayCollision& collision) {
4     Vector3  boxPos    = worldTransform.GetWorldPosition();
5     Vector3 boxSize    = volume.GetHalfDimensions();
6     return RayBoxIntersection(r, boxPos, boxSize, collision);
7  }
```

CollisionDetection::RayAABBIntersection

Things are a little more tricky for an OBB, as we need to transform the ray so that it's in the local space of the box (line 12), and if its colliding, transform the collision point back into world space (line 18). To bring the ray into the local space of the box, we subtract the box's position (line 10), and transform the newly formed relative position by the inverse of the box's orientation (formed using the *conjugate* of its orientation quaternion - nicer than inverting matrices!), along with the ray's direction, giving us a new temporary ray, defined within the frame of reference of the OBB. We can then raycast versus an 'AABB' that is at the origin (the box sits at its own origin position), and, if its colliding, we bring the collision point back into world space by performing the opposite operations on the collision point (we add the position back on, and then we rotate it by the world transform, seen on line 18).

```
1  bool CollisionDetection::RayOBBIntersection(const Ray&r,
2        const Transform& transform,
3        const OBBVolume& volume, RayCollision& collision) {
4     Quaternion orientation  = worldTransform.GetWorldOrientation();
5     Vector3 position        = worldTransform.GetWorldPosition();
6
7     Matrix3 transform    = Matrix3(orientation);
8     Matrix3 invTransform = Matrix3(orientation.Conjugate());
9
10    Vector3 localRayPos  = r.GetPosition() - position;
11
12    Ray tempRay(invTransform*localRayPos,invTransform*r.GetDirection());
13
14    bool collided = RayBoxIntersection(tempRay, Vector3(),
15             volume.GetHalfDimensions(), collision);
16
17    if (collided) {
18       collision.collidedAt=transform*collision.collidedAt+position;
19    }
20    return collided;
21 }
```

CollisionDetection::RayOBBIntersection

Main File

The program has been wired up to already use the raycasting functions - that's why they were already there, but were just returning **false**. The actual raycasting is performed in the *SelectObject* function in the **TutorialGame** class - note that it's calling the *Raycast* method of the **GameWorld** class, which will iterate over every game object in the scene, and call the *RayIntersection* method we filled in earlier to see if its colliding or not. There's quite a lot of functions that we haven't covered the contents of here, so you might want to investigate how they work, and how everything is tied together within the **TutorialGame** class, which has an *UpdateGame* method called by the **main** function, in a **while** loop that should feel pretty familiar to what you saw in the previous module.

Conclusion

While the demo program looks somewhat simple, we've learned quite a lot by making it. Firstly, we've seen how to click on things in the world - there aren't many games where you don't click on something or at least determine what's under the crosshair or mouse pointer, so this in itself is important. Secondly, we've seem how this same mechanism can allow programmers to determine whether an object can see another. AI often uses such tests to simulate vision to work out where to go or what to attack, so this will come in handy as we get into AI algorithms later. Finally, we've also learned the fundamentals behind some basic collision shapes, and seen how calculating intersections with rays is not too computationally intensive. We'll be using these shapes more often as we get into collision detection and collision resolution algorithms.

In the next two tutorials, we're going to take a look at how to simulate physically accurate linear and angular motion in our objects, so that they can move in a realistic manner, and start to build up the actual physics part of our physics engine. We'll be using raycasting to push objects around, though, so its been very useful adding the to form rays from the camera and select objects.

Further Work

1) Sometimes we want to selectively ignore certain types of object when performing raycasting - if the ghosts in Pac-Man were to use raycasts to determine if they could see the player, it wouldn't be much use to nearly always return a collectable pellet as the closest object! In the Unity game engine, this problem is usually alleviated via the use of collision *layers*, in which every object is tagged with a 'layer' property, where only certain combinations of layers can collide or be raycasted against. Investigate Unity's 'Layer-based collision detection' and consider how you could implement a similar system as a small addition to the **GameObject** and **GameWorld** classes.

2) It's a common operation in game AI to determine whether object A can see object B. Try using the new raycasting code out to see which **GameObject** in the world can be seen from the forward direction of the selected **GameObject**.

3) Now is a good time to familiarise yourself with the debug line drawing capability - try adding some calls to **Debug::DrawLine** to visualise the rays being cast from the mouse / objects.

$$
\begin{aligned}
k_1 &= \Delta t \cdot a(t, v(t)) \\
k_2 &= \Delta t \cdot a(t + \tfrac{\Delta t}{2}, v(t) + \tfrac{k_1}{2}) \\
k_3 &= \Delta t \cdot a(t + \tfrac{\Delta t}{2}, v(t) + \tfrac{k_2}{2}) \\
k_4 &= \Delta t \cdot a(t + \Delta t, v(t) + k_3) \\
v(t + \Delta t) &= v(t) + \tfrac{k_1 + 2k_2 + 2k_3 + k_4}{6}
\end{aligned}
$$