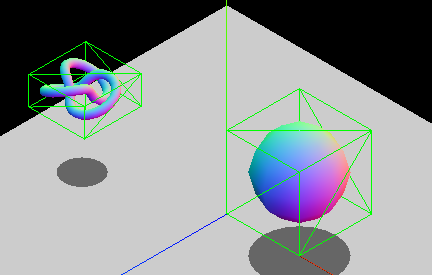
# Introduction

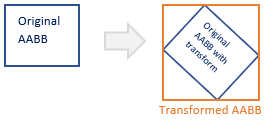


Some bounding boxes, from <https://developer.mozilla.org/en-US/docs/Games/Techniques/3D_collision_detection>

The exercises below are concerned with the 3D [axis aligned bounding box](https://www.gamasutra.com/view/feature/131833/when_two_hearts_collide_.php) (AABB) of an object, which is the smallest box that entirely contains a geometric object whilst keeping its edges aligned with the coordinate axes. An AABB is defined by its minimum and maximum vertices, and , whose coordinates correspond to the minimum points, or vertices, of an object. AABBs are commonly used to accelerate the collision detection process.

You may find the Symbolab [matrix multiplication calculator](https://www.symbolab.com/solver/matrix-multiply-calculator) useful for finding/checking your results.

# Tasks

1. Consider an object whose shape is defined by the set of five points (7, 11, -5), (2, 3, 8), (-3, 3, 1), (-5, -7, 0) and (6, 3, 4).
   1. What are the extents of the AABB, and , for these five points?  
      The minimum and maximum points are simply the ones made up of all the smallest/largest , and values out of all the points:
   2. Determine the centre point of the AABB.  
      For a general set of points, the centre point is just their average position. As an AABB is symmetrical, its centre is half-way between and :
   3. Multiply the five points by the following matrix (a 45° rotation about the -axis):
   4. What is the AABB of these transformed points?  
       = (-4.242, -8.484, -5), = (2.121, 12.726, 8)
   5. What is the AABB we get by transforming the original AABB, defined by the points in part (a), instead of the five object points? How does it compare to the AABB of the transformed points found in part (d)?  
        
      Note that if we simply apply the transformation to the bounding box extents and , the result would be a box of the same dimensions in the rotated coordinate space, which is **not** axis aligned – we want to get the new AABB of the rotated original AABB, as on the right.  
        
      As a result, we might think we need to find each of its corner points and transform them individually to see what the minimum and maximum values of each component are, but we can take a shortcut by expressing the transform as a parametric equation, which tells us what effect the transform will have on each of the input points. If us the result of applying the transformation to point , then from applying the matrix, we have:  
      (Note that the value is unchanged because that’s the axis we’re rotating around).  
        
      Since we will find the new min/max values by choosing the components from each transformed point, we just have to find which combination of the min/max values of , and will give the min/max values of , and , which (by inspection/brute force testing) in this case gives us:  
      Note that this is much larger than the AABB of the transformed points in part (e)… Depending on the context, the increased efficiency of not having to transform all the points may or may not compensate for the less-than-optimal AABB size. (For more details on this approach, please see section 9.4 of *3D Math Primer for Graphics and Game Development*, by Fletcher Dunn and Ian Parberry).
2. In the week 5 lecture, an algorithm was presented for finding the intersection of two AABBs in 2D; the extension to 3D is a simple matter, but there are more shapes we might want to deal with. Describe (in English and/or pseudocode) how one might test for the following intersections, giving an example of when each might be used:
   1. A plane and an AABB.
   2. A ray (line) and an AABB.

Plane-AABB intersection test

For the plane to pass through the AABB, at least one of the AABB vertices must be on the opposite side of the plane from the others, which we can test by comparing the dot products of the vertices with the plane normal.

If the plane is defined in the implicit manner as , where with being a known point on the plane, then points on one side of the plane will have a dot product greater than , while for those on the other side it is less than ; in other words, we compare the projection of the point we’re testing, , onto the plane normal direction with the projection of a point on the plane, , in the same direction, to see which is further from the origin (see also the calculation of the distance of a point from a plane in the second part of week 7’s lecture).

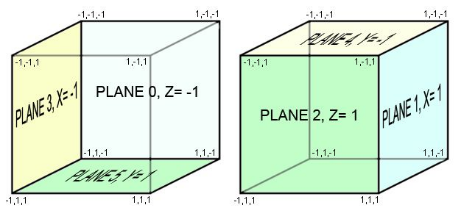
The calculation can be simplified by finding only the minimum and maximum values of the dot product, instead of computing it for all eight vertices. Since all of the AABB vertices are just different combinations of the min and max , and values from the box extents and , and the dot product is the sum of products of the normal coefficients with the point coefficients, we can find the **maximum** value by choosing e.g. if and if , and vice versa for the minimum.

bool planeIntersectsAABB(AABB a, Vector3 n, float d)  
  
 float minD = 0, maxD = 0;  
  
 for each component c in x, y, z  
 if (nc > 0)  
 minD += nc \* minc;  
 maxD += nc \* maxc;  
 else  
 minD += nc \* maxc;  
 maxD += nc \* minc;

return (minD < d and maxD > d)

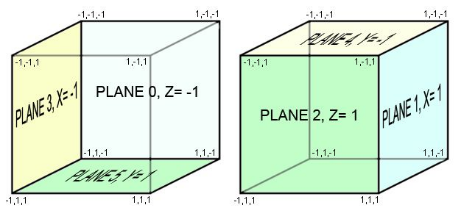
Ray-AABB intersection test

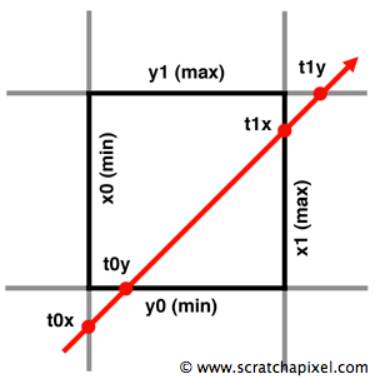
This test is commonly used to speed up ray-casting or ray-tracing of complex objects, by first performing a rejection test on the objects’ AABBs.

One way to perform this test is to note that a cuboid is formed of six intersecting planes, each of which is aligned with two of the axes: i.e. there are two planes parallel to each of the , and -planes. We can define these planes using the values of the AABB’s min/max points – for example, , – which constrains the values along one of the axes while allowing the plane to extend infinitely along the other dimensions.

For the AABB to be intersected means that the ray must pass through at least one of these planes, within the bounds of the vertices. With the line expressed in the usual manner, as  
 (with being any point on the line, the direction vector, a known point on the line and the scalar parameter), we can find the point of intersection of the ray with each of the AABB planes via the standard method of substituting the plane “equation” for the relevant component, in this case a single scalar, into the ray/line equation.  
  
For example, to find the intersection with the plane defined by the minimum value of the AABB:  
(In other words, we already know the value of the coordinate at the point of intersection, since it’s the same everywhere on the plane, so we can substitute this directly to find using only the components).

Image from <https://www.innerdrivestudios.com/blog/part-iv-plane-detection-a-first-step-to-implementing-panorama-hotspots/>



Once we’ve found the intersection points for all the planes (testing first to make sure that the ray is not parallel to any of them – note the division in the equation above!), we need to check that they are actually within the box. We could compare all the points against all the AABB vertices, however a faster way is to notice that if an intersection occurs within the box boundaries, it must include the one that is the furthest from the ray origin, i.e. with the greater value of , for the planes defined by the box’s minimum boundaries (assuming the ray source is on that side of the box).  
  
Comparisons of the values can also be used to discover which rays miss the box; further details on this computation (along with code examples) can be found here: <https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-box-intersection>  
and in Appendix A.18 of “3D Math Primer for Graphics and Game Development” by Fletcher Dunn and Ian Parberry.