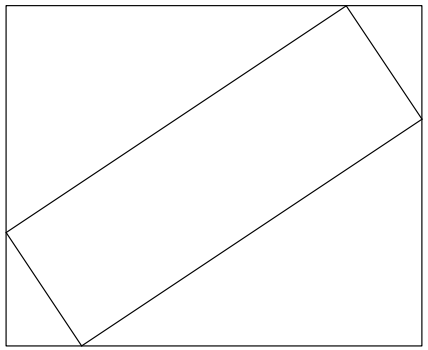
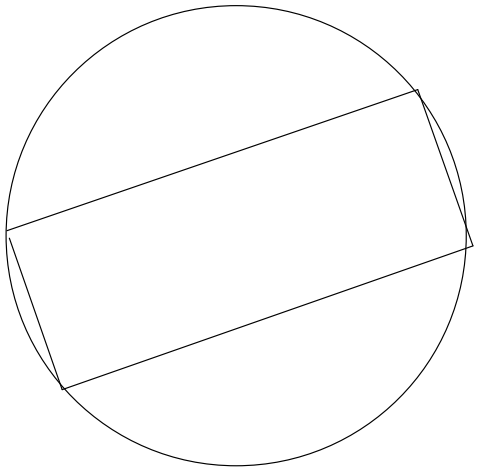
Collision detection: bounding boxes, bounding spheres

- accurate collision detection can be expensive
  - this is particularly true in PGE which will calculate the time of next collision
- sometimes an accurate time of next collision is not necessary
  - for example if the objects are sufficiently far apart and are travelling slowly
- an inexpensive way to determine whether objects are not going to collide is to use the bounded shape technique

Bounding rectangle (boxes)



Bounding circle



Bounding boxes, bounding spheres

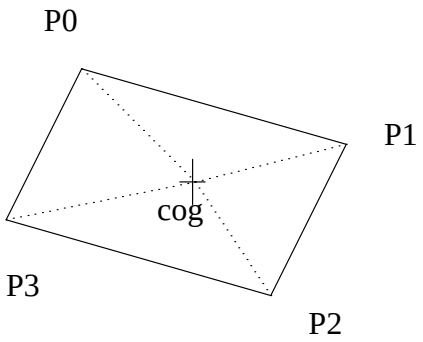
- these approaches can be very useful as they allow us to treat polygons as circles
  - and circles as polygons
  - for the purpose of collision detection

- we can also combine shapes into an aggregate circle or rectangle

- finally creating bounding circles will help detect whether a rotating object will not collide (within a time period)
  - should provide a significant optimisation for rotating objects which are spinning but not moving
  - a bounding circle is a single object, compared to a polygon - which must have at least 3 vertices

Implementing bounding circle in PGE

- recall that polygons are represented by an array of vertices
  - each vertice has a polor coordinate from the center of gravity



- we need to find the longest point away from the centre of gravity and this will become our radius

Implementing bounding circle in PGE

- the polar coordinates are defined by a radius and angle
- **Sandpit/git-pge/c/polar.c**

```
struct polar_Polar_r {
                        double r;
                        double w;
                      };
```

- we can ignore the angle and choose the largest radius
  - at this point we have a bounded circle which can be used to test against other circles

Implementing bounding circle in PGE

- we can see the collision detection commences in the function `findCollision`

Implementing bounding circle in PGE

$HOME/Sandpit/pge/c/twoDsim.c

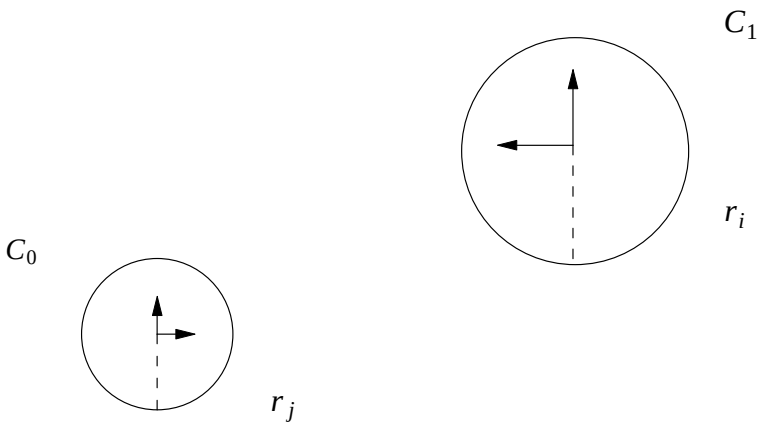
```
static void findCollision (Object iptr, Object jptr,
                           eventDesc *edesc, double *tc)
{
  if (! (iptr->fixed && jptr->fixed))
    {
      /* avoid gcc warning by using compound statement even
         if not strictly necessary.  */
      if ((iptr->object == circleOb) && (jptr->object == circleOb))
        findCollisionCircles (iptr, jptr, edesc, tc);
      else if ((iptr->object == circleOb) && (jptr->object == polygonOb))
        findCollisionCirclePolygon (iptr, jptr, edesc, tc);
      else if ((iptr->object == polygonOb) && (jptr->object == circleOb))
        findCollisionCirclePolygon (jptr, iptr, edesc, tc);
      else if ((iptr->object == polygonOb) && (jptr->object == polygonOb))
        findCollisionPolygonPolygon (jptr, iptr, edesc, tc);
    }
}
```

Implementing bounding circle in PGE

- we can see the engine categorise the various objects under collision
- potentially we can optimise these calls one at a time

Optimising earlierCircleCollision

- notice that the parameters to `earlierCircleCollision` define both circles, position, velocity, acceleration and radius



Optimising earlierCircleCollision

- r_i is the radius for circle, i
  - $a = x_i$ is the x position for circle, i
  - $g = y_i$ is the y position for circle, i
  - $c = vx_i$ is the velocity for circle, i, along the x axis
  - $k = vy_i$ is the velocity for circle, i, along the y axis
  - $e = ax_i$ is the acceleration for circle, i, along the x axis
  - $m = ay_i$ is the acceleration for circle, i, along the y axis

Optimising earlierCircleCollision

- r_j is the radius for circle, j
  - $b = x_j$ is the x position for circle, j
  - $h = y_j$ is the y position for circle, j
  - $d = vx_j$ is the velocity for circle, j, along the x axis
  - $l = vy_j$ is the velocity for circle, j, along the y axis
  - $f = ax_j$ is the acceleration for circle, j, along the x axis
  - $n = ay_j$ is the acceleration for circle, j, along the y axis

Implementing a short circuit test to determine whether circles might collide in near future

- we know that they touch when the circles are $r_i + r_j$ units apart
- we know the current distance apart is
- $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$
- so when $\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} - (r_i + r_j) > 0$
- can simply this to $(x_i - x_j)^2 + (y_i - y_j)^2 - (r_i + r_j)^2 > 0$

Implementing a short circuit test to determine whether circles might collide in near future

- the circles have a gap between them
- if the total relative distance travelled between the circles over the next frame is less than this gap we know they cannot collide in the next frame
  - this **might** be less expensive than completing the function `earlierCircleCollision`
- recall that the distance an object travels over time if we know its acceleration, velocity is
- $ut + \frac{1}{2} at^2$

Implementing a short circuit test to determine whether circles might collide in near future

- we can separate out the x and y velocity and acceleration
  - and use Pythagoras to determine the length of vector distance travelled

- let $t = \frac{1}{framesPerSecond}$

- $S = \sqrt{((d-c)t + \frac{1}{2}(e-f)t^2)^2 + ((k-l)t + \frac{1}{2}(m-n)t^2)^2}$

Implementing a short circuit test to determine whether circles might collide in near future

- $W = (x_i - x_j)^2 + (y_i - y_j)^2 - (r_i + r_j)^2 > 0$

- if the value $S^2 < W$ then there will be no collision in the next frame by these objects

Tutorial

- implement a short circuit function for two moving circles in earlierCircleCollision
- observe the frames per second in your new optimised PGE
  - does it make a noticable difference?
- a careful implementation will allow this short circuit function to be called from within polygon/polygon and polygon/circle tests (using bounded circles)