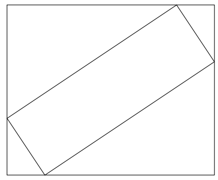
# Collision detection: bounding boxes, bounding spheres

- 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

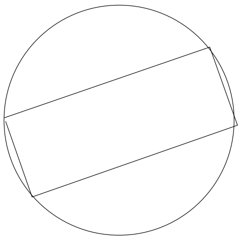


# Bounding rectangle (boxes)

- 



# Bounding circle

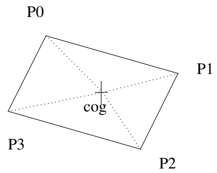
- 



# 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



# Collision detection pipeline

- the PGE uses both collision prediction and frame based collision detection
- both techniques are fed from the broadphase list
- study the function `initBroadphase` and also the `broadphase` structure (`_T5_r`)



# broadphase struct

- **pge/c/twoDsim.c**

```
   {
     unsigned int o0; /* first object potentially in colli
     unsigned int o1; /* second object potentially in coll
     broadphase next; /* next pair of objects.  */
   }
```

- the function `initBroadphase` generates a list of pairs of object which need to be examined
    - many of which will not collide

## broadphase struct

- it is expensive (time) to accurately determine whether an object will collide
  - but much less expensive to cull the list of object pairs which cannot collide
  - you can implement this optimisation and then observe the FPS of the game engine
  - study the function `optBroadphase`
  - notice that this is only called when the game engine is in frame based mode



## optPredictiveBroadphase

- examine the function `optPredictiveBroadphase`
  - this is only called when the game engine is in predictive mode
  - start with this function, as predictive mode is the game engine default

- try implementing `optPredictiveBroadphase` so that it culls pairs of objects which are moving away from each other
  - you should check both acceleration vectors and velocity vectors of both objects
    - hint examine and use `circle_moving_towards`



## optBroadphase

- is easier to implement than `optPredictiveBroadphase` but it is only used when `pge` runs in frame based interpentration mode

- `optBroadphase` can be implemented using bounding circles
  - also implement bounded rectangle culling
  - make your implementation count the culling categories

- it might be a good idea to have a Python API call to turn on/off these two optimisations



## optBroadphase

- observe the frames per second in your new optimised PGE
  - does it make a noticable difference?