Programming Proverbs

- 20. “Provide good documentation.”

- Henry F. Ledgard, “Programming Proverbs: Principles of Good Programming with Numerous Examples to Improve Programming Style and Proficiency”, (Hayden Computer Programming Series), Hayden Book Company, 1st edition, ISBN-13: 978-0810455221, December 1975.

Collision response references

- Ian Millington, "Game Physics Engine Development", 2nd Edition, Morgan Kaufmann, 2010

- David M Bourg, "Physics for Game Developers", O’Reilly Media, November 2001

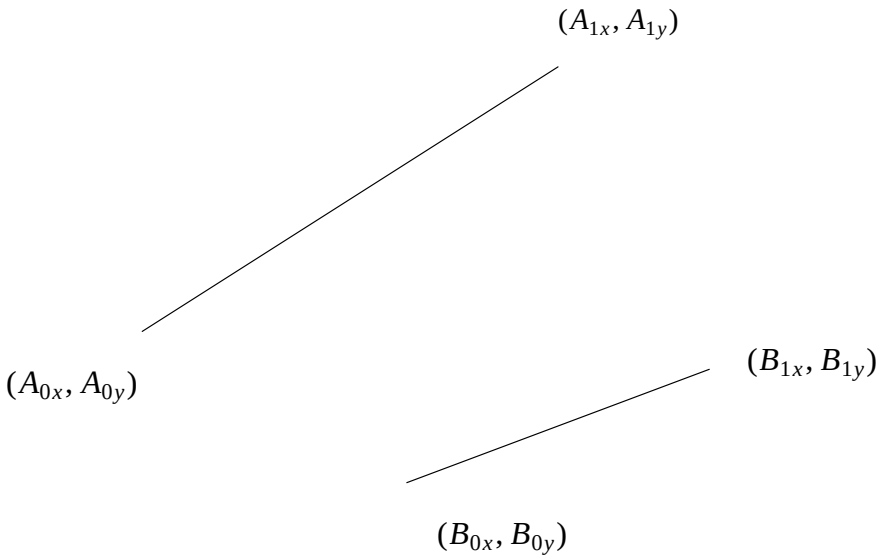
- André LaMothe, "Tricks of the Windows Game Programming Gurus: Fundamentals of 2d and 3d Game Programming", Sams; 2 edition, June 2002, ISBN-10: 0672323699, ISBN-13: 978-0672323690

Line on Line collision detection

- is actually very easy, if we already have implemented:
- circle circle collision detection
- circle line collision detection

Line on Line collision detection

- consider the following diagram:



Line on Line collision detection

- each line has a velocity and acceleration vector

Line on Line collision detection

- to find the time of next collision, we ask the following questions:

- what is the smallest value of time $t \geq 0$ for the next collision of a circle of radius 0 at
  - (A_{0x}, A_{0y}) crossing line B
  - (A_{1x}, A_{1y}) crossing line B
  - (B_{0x}, B_{0y}) crossing line A
  - (B_{1x}, B_{1y}) crossing line A

- thankfully we use the circle line algorithm described before
  - which in turn uses the circle circle solution

Game engine structure

- there are many components to a game engine: (non exclusive taxonomy)
  - collision detection
  - motion of objects
  - contact resolution
  - handling forces: gravity, friction
  - handling momentum: impacts, collision response
  - managing different objects: springs, rigid objects

Game engine structure

- there are also different high level techniques
  - frame based physics
  - event based physics
- most games use frame based physics
  - we have concentrated on event based (collision prediction)
- both have advantages and disadvantages

Event based

- as long as we can compute the time of the next event
  - then we only need to alter the state (game) when an event occurs
  - principle of discrete event simulation
  - can be highly efficient, and accurate
  - the correct solution for modelling a game of snooker for example

- not good for implementing Rage!
  - as the Mathematics would become highly complex

Event based

- event loop is very simple, here is the loop found in `c/twoDsim.c`

```
addEvent(0.0, drawFrameEvent);
addNextCollisionEvent;
while (s<t)
   {
      dt = doNextEvent();
      s = s + dt;
   }
updatePhysics(currentTime-lastCollisionTime);
lastCollisionTime = currentTime;
```

Collision response

- PGE implements six collision categories
  - moving circle hitting a fixed circle
  - moving circle hitting a moving circle
  - moving circle hitting a fixed line
  - moving circle hitting moving line (polygon)
  - moving polygon hitting fixed polygon collision
  - moving polygon hitting moving polygon collision

- still to do are:
  - rotating polygon collision prediction

Collision response

- worth noting that implementing an event based system makes it easier to categorise the above
  - we recall that line on line collision builds upon line on circle and circle on circle
  - likewise if we remember this information, we can sometimes call the simpler collision response routines

- for example if a circle hits a fixed polygon corner
  - then we call circle hitting fixed circle of radius zero

Response for a moving circle hitting a fixed circle

- movable is an circle Object
- center is a coordinate which has been hit
- following code uses linear kinetic energy equation

$$KE_{linear} = \frac{mv^2}{2}$$

- and energy is conserved:

$$m_1 v_1^2 + m_2 v_2^2 = m_1 v_3^2 + m_2 v_4^2$$

Response for a moving circle hitting a fixed circle

```
/* calculate normal collision value */
c.x = movable->c.pos.x - center.x ;
c.y = movable->c.pos.y - center.y ;
r = sqrt(c.x*c.x+c.y*c.y) ;
normalCollision.x = c.x/r ;
normalCollision.y = c.y/r ;
relativeVelocity.x = movable->vx ;
relativeVelocity.y = movable->vy ;
```

Response for a moving circle hitting a fixed circle

```
j = (-(1.0+1.0) *
     ((relativeVelocity.x * normalCollision.x) +
      (relativeVelocity.y * normalCollision.y)))/
     (((normalCollision.x*normalCollision.x) +
      (normalCollision.y*normalCollision.y)) *
      (1.0/movable->c.mass)) ;

movable->vx := movable->vx + (j * normalCollision.x) / movable->c.mass ;
movable->vy := movable->vy + (j * normalCollision.y) / movable->c.mass ;
```

Response for a moving circle hitting a moving circle

- `iptr` and `jptr` are both circles moving and have just collided
- very similar code
- David M Bourg, "Physics for Game Developers", O’Reilly Media, November 2001 see p90-97
- in both previous and next code `j` is the impulse of the collision

Response for a moving circle hitting a moving circle

```
/* calculate normal collision value */
c.x = iptr->c.pos.x - jptr->c.pos.x ;
c.y = iptr->c.pos.y - jptr->c.pos.y ;
r = sqrt(c.x*c.x+c.y*c.y) ;
normalCollision.x = c.x/r ;
normalCollision.y = c.y/r ;
relativeVelocity.x = iptr->vx - jptr->vx ;
relativeVelocity.y = iptr->vy - jptr->vy ;
```

Response for a moving circle hitting a moving circle

```
j = (-(1.0+1.0) *
     ((relativeVelocity.x * normalCollision.x) +
      (relativeVelocity.y * normalCollision.y)))/
    (((normalCollision.x*normalCollision.x) +
      (normalCollision.y*normalCollision.y)) *
     (1.0/iptr->c.mass + 1.0/jptr->c.mass)) ;

iptr->vx = iptr->vx + (j * normalCollision.x) / iptr->c.mass ;
iptr->vy = iptr->vy + (j * normalCollision.y) / iptr->c.mass ;

jptr->vx = jptr->vx - (j * normalCollision.x) / jptr->c.mass ;
jptr->vy = jptr->vy - (j * normalCollision.y) / jptr->c.mass ;
```

Circle colliding against fixed edge

- `cPtr` is the circle object `p1` and `p2` are the coordinate pairs of the edge

```
/* firstly we need to find the normal to the line */
sortLine(p1, p2) ;  /* p1 is left of p2, or lower than p2 */

/* create the vector p1 -> p2 */
v1 = subCoord(p2, p1) ;

perpendiculars(v1, n1, n2) ;

/* use n1 */
n1 = normaliseCoord(n1) ;
vel = initCoord(cPtr->vx, cPtr->vy) ;
vel = addCoord(scaleCoord(n1, -2.0 * dotProd(vel, n1)), vel) ;

cPtr->vx = vel.x ;
cPtr->vy = vel.y ;
```

Further reading

- chapter 13 in
- André LaMothe, ‘‘Tricks of the Windows Game Programming Gurus: Fundamentals of 2d and 3d Game Programming’’, Sams; 2 edition, June 2002, ISBN-10: 0672323699, ISBN-13: 978-0672323690
- pages 90-97 of
- David M Bourg, ‘‘Physics for Game Developers’’, O’Reilly Media, November 2001