

PGE Springs, Python and Internals

- examine the example code in the pge source tree:

PGE Springs, Python and Internals

- `$HOME/Sandpit/git-pge/examples/springs/bridge.py`

```

left = placeBall (wood_light, 0.25, 0.45, 0.03).fix ()
right = placeBall (wood_light, 0.75, 0.45, 0.03).fix ()

prev = left
springs = []
for x in range (35, 75, 10):
    step = placeBall (wood_dark, float (x) / 100.0, 0.33,
                    .mass (0.9)
    s = pge.spring (prev, step, spring_power, damping, 0.1)
    .draw (yellow, 0.002)
    s.when (snap_length, snap_it)
    springs += [s]
    prev = step

s = pge.spring (right, prev, spring_power, damping, 0.1)
.s.draw (yellow, 0.002)
.s.when (snap_length, snap_it)

```

PGE Springs, Python and Internals

- notice that two circles are fixed in position left and right
- now free moving circles are declared at positions: 35, 45, 55 and 65.
- all of these circles are joined by a spring and each spring will snap if it exceeds `snap_length`
- each spring has a `k` value and also a damping value

PGE Springs, Python and Internals

- `s = pge.spring (prev, step, spring_power, damping, 0.1) \`
`.draw (yellow, 0.002)`
- here the `k` value is `spring_power` and uses damping and has an at rest length of 0.1 unit
- pge allows debugging (or visual showing of a spring yellow and 0.002 (width of the rectangle representing the spring)
 - this yellow visual cue has no effect in pge, it is simply drawn between the end points of a spring object

PGE Springs, Python and Internals

- a spring can be requested to call a callback function when it reaches a specific length
 - for example when it reaches `snap_length` it calls `snap_it`
- `snap_it` is a simple function

```
def snap_it (event, object):
    object.rm ()
```

- and the spring is deleted, the event parameter (representing the function call) is ignored
 - as the only time this function is called is when a spring is to be deleted

PGE Spring Internals

- the main module of the physics engine is `$HOME/Sandpit/git-pge/c/twoDim.c`
- the `Spring` entity is declared as a struct called `Spring_r` and is defined as:

PGE Spring Internals

- `$HOME/Sandpit/git-pge/c/twoDim.c`

```
struct Spring_r {
    unsigned int id1; /* spring connects to object id1. */
    unsigned int id2; /* and id2. */
    coord_Coord f1; /* force of spring acting on id1. */
    coord_Coord f2; /* force of spring acting on id2. */
    coord_Coord a1; /* acceleration vector of spring op
    coord_Coord a2; /* acceleration vector of spring op
    double k; /* Hookes constant for the spring. */
    double d; /* Damping constant for the spring. */
    double l0; /* at rest length. */
    double cbl; /* the call back length of the spring. */
    double l1; /* l1 is the current length of the spring.
    double width; /* width of the rectangle used for draw
```

PGE Spring Internals

- `$HOME/Sandpit/git-pge/c/twoDim.c`

```
    unsigned int drawColour; /* drawing colour. */
    unsigned int endColour; /* what colour to draw spring
    unsigned int midColour; /* what colour to draw spring
    unsigned int draw; /* should the spring be draw
    unsigned int drawEnd; /* should it be redrawn at t
    unsigned int drawMid; /* should it be redrawn in t
    unsigned int hasCallBackLength; /* is the call back l
    unsigned int func; /* which function should we call
};
```

PGE Internals

- one of the design decisions in building PGE was to assume that acceleration remains constant in between events
- velocity and position components of objects however will vary depending upon time
 - acceleration remains constant over time
 - but might change at the next event (collision or user input)
- this works well until springs are introduced!
- Hookes Law $F = -k(l_1 - l_0)$
- and Newtons Law: $F = ma$
- can be combined to show that:

- $a = \frac{F}{m}$
- $a = \frac{-k(l_1 - l_0)}{m}$

slide 10

slide 11
gaius

PGE Internals

- however PGE can be adapted so that it adjusts the acceleration of each sprung object every time frame
 - this is an approximation - similar to numerical integration
 - a tradeoff, but it allows springs to coexist inside PGE

PGE Internals

- mass is constant, but l_1 changes with time
 - thus acceleration will also vary over time
- in effect adding a spring into PGE will potentially violate one of the core design parameters of PGE

slide 12
gaius

The data structures inside c/twoDsim.c

- `c/twoDsim.c`

```
typedef enum {polygonOb, circleOb, springOb} ObjectType;
typedef enum {frameKind, functionKind, collisionKind} eventKind;
typedef enum {frameEvent, circlesEvent, circlePolygonEvent, polygonPolygonEvent, functionEvent, springEvent}
```
- `ObjectType` defines the different kinds of object (ignore spring object)
- `eventKind` defines the three major classification of events

The data structures inside c/twoDsim.c

- eventType further subclassifies the event kind with the collision event info
 - we distinguish between a circle/polygon collision and a circle/circle collision and a polygon/polygon collision

object (typedef struct _T2_r)

c/twoDsim.c

```

unsigned int id;          /* the id of the object. */
unsigned int deleted;    /* has it been deleted? */
unsigned int fixed;      /* is it fixed to be world? */
unsigned int stationary; /* is it stationary? */
double gravity;          /* the gravity for this object. */
coord_Coord saccel;     /* the acceleration due to a sp
double vx;               /* velocity along x-axis. */
double vy;               /* velocity along y-axis. */
double ax;               /* acceleration along x-axis.
double ay;               /* acceleration along y-axis.
double inertia;         /* a constant for the life of t
double angleOrientation; /* the current rotation angle o
double angularVelocity; /* the rate of rotation. (Rota
double angularMomentum; /* used to hold the current mom
unsigned int interpen;   /* a count of the times the obj
ObjectType object;      /* case tag */
union {
    Polygon p; /* object is either a polygon, circle
    Circle c;
    Spring s;
};

```

object (typedef struct _T2_r)

- c/twoDsim.c


```

typedef struct _T2_r _T2;
typedef _T2 *Object;

```

- notice you can ignore the inertia, angleOrientation, angularVelocity and angularMomentum as these are used to implement rotation

Circle

- c/twoDsim.c


```

typedef struct Circle_r Circle;

struct Circle_r {
    coord_Coord pos; /* center of the circle in the w
    double r;        /* radius of circle. */
    double mass;     /* mass of the circle. */
    deviceIf_Colour col; /* colour of circle. */
};

```

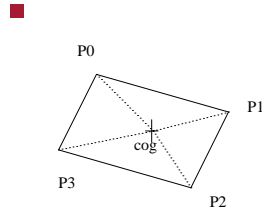
Polygon

```

c/twoDsim.c
typedef struct Polygon_r Polygon;
struct _T3_a { polar_Polar array[MaxPolygonPoints+1]; };
struct Polygon_r {
    unsigned int nPoints;
    _T3 points;
    double mass;
    deviceIf_Colour col;
    coord_Coord cOfG;
};
typedef struct _T3_a _T3;

```

- the polygon has an array which is used to contain each corner
 - a corner is a polar coordinate from the centre of gravity



Polar coordinates

- remember that a polar coordinate has a magnitude and an angle
 - an angle of 0 radians is along the x-axis
 - magnitude of, r and an angle of ω
- so we can convert a polar to cartesian coordinate by:
 - $x = \cos(\omega) \times r$
 - $y = \sin(\omega) \times r$

Polygon

- in our diagram
 - $P0 = (p0, 135/360 \times 2\pi)$
 - $P1 = (p1, 45/360 \times 2\pi)$
 - $P2 = (p2, 315/360 \times 2\pi)$
 - $P3 = (p3, 225/360 \times 2\pi)$
- where $p1, p2, p3, p4$ are the lengths of the line from the CofG to the corner
 - dotted lines in our diagram

Polar coordinates

Polar coordinates

- the angle values in the polar coordinates for our polygon are the offset of the angle for the particular corner
 - the angularVelocity is used to determine the rotation of the polygon, this is added to each corner to find out the corner position at any time
- this allows rotation of the polygon to be modelled at a later date

Polar coordinates

- at any time in the future, t we can determine the polygons corner, i by:
 - $\Omega = \text{angleOrientation} + \text{angularVelocity} \times t$
 - $x_i = \text{cofg}_x + r_i \times \cos(\omega_i + \Omega)$
 - $y_i = \text{cofg}_y + r_i \times \sin(\omega_i + \Omega)$

Polar coordinates

- we can see how this data structure represents a polygon by following the `dumpPolygon` function

Polar coordinates

- see how each corner is defined by following through the function `box`
 - into `poly4`
- how it calculates the box CofG
- how it defines each corner relative to the CofG and as a polar coordinate
 - each corner is orbiting the CofG

dumpPolygon

c/twoDsim.c

```

static void dumpPolygon (Object o)
{
    unsigned int i;
    coord_Coord c0;

    libc_printf ((char *) "polygon mass %g colour %d\n", 20,
                 o->p.mass, o->p.col);
    libc_printf ((char *) " c of g (%g,%g)\n", 19,
                 o->p.cOfG.x, o->p.cOfG.y);
    for (i=0; i<=o->p.nPoints-1; i++)
    {
        c0 = coord_addCoord (o->p.cOfG,
                             polar_polarToCoord (polar_rotatePolar
                                                    ((polar_Polar) o->p.points.array[i], o->p.angle0),
                                                    o->p.points.array[i]));
        libc_printf ((char *) " point at (%g,%g)\n", 20, c0.x, c0.y);
    }
}

```

dumpPolygon

- follow through the function doDrawFrame and see how the corners of a polygon are updated dependant upon the angularVelocity, angleOrientation and the acceleration and velocity components
- examine newPositionRotationCoord, newPositionRotationSinScalar and newPositionRotationCosScalar