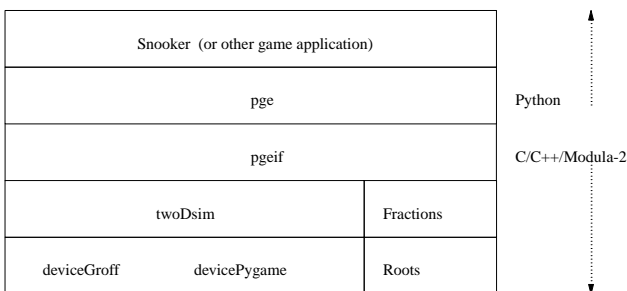


Data structures used in PGE

- in this lecture we will examine the key data structures used in PGE
- at the end of the lecture you should understand how these data structures are used to represent the world of polygons, circles and colours in the game engine
- before we examine the data structures we will examine the API layering in a little more detail

slide 3
gaius

API layering



slide 4
gaius

API layering

- recall
 - python/pge.py is written in Python
 - c/pgeif.c is written in C and its external Python functions are defined in i/pgeif.i
 - swig generates the wrapping code
- the file c/pgeif.c contains the implementation of all the publically accessible Python methods
- it also ensures that all publically created objects in the Physics game engine are remembered and stored in this file

API layering

- this allows colours, polygons, circles to be mapped onto their high level Python counterparts in `python/pge.py`
- it also allows the implementation of `python/pge.py` to be cleaner as it will always obtain any object from `c/pgeif.c`
- examine the implementation for `box` inside `c/pgeif.c`
- we see that much of `c/pgeif.c` just calls upon the services of the lower layer `c/twoDsim.c`
 - after performing extensive checking of parameter types

Implementation of box

c/pgeif.c

```

/*
 * box - place a box in the world at (x0,y0),(x0+i,y0+j)
 */
unsigned int box (double x0, double y0,
                 double i, double j, unsigned int c)
{
    double k;

    x0 = check_range (x0, (char *) "box", 3, (char *) "x0",
                     y0 = check_range (y0, (char *) "box", 3, (char *) "y0",
                     k = check_range (x0+i, (char *) "box", 3, (char *) "x0+",
                     k = check_range (y0+j, (char *) "box", 3, (char *) "y0+");
    return trace (addDef ((TypeOfDef) object,
                          twoDsim_box (x0, y0, i, j,
                                       (deviceIf_Colour) lookupDef ((TypeOfDef) object,
                                                                      (char *) "box", 3);
    }

```

Implementation of box

- we see that it creates a box (using `twoDsim_box`)
 - it saves this box in its local definitions `addDef`
 - it is saved as an object and not a colour
- also note that the 5th parameter to `twoDsim_box` is a colour id, `c`, which is looked up using `lookupDef`

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} eventType;

```

- `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

c/twoDsim.c

```
typedef struct _T2_r {
    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 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 time.
    double angleOrientation;  /* the current rotation angle.
    double angularVelocity;   /* the rate of rotation.
    double angularMomentum;  /* used to hold the current angular momentum.
    unsigned int interpen;    /* a count of the times the object has interpenetrated.
    ObjectType object;       /* case tag */
    union {
        Polygon p; /* object is either a polygon, circle or spring.
        Circle c;
        Spring s;
    };
};
```

object

- 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 world.
    double r;        /* radius of circle.
    double mass;     /* mass of the circle.
    deviceIf_Colour col; /* colour of circle.
};
```

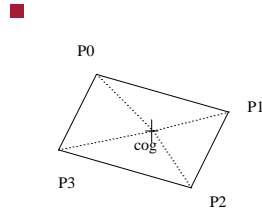
Polygon

```

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", 2,
                 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->angle0,
                                                    libc_printf ((char *) " point at (%g,%g)\n", 20,
    }
}

```

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

Acceleration and Conclusion

- examine the function getAccelCoord and see if you can think how you might modify PGE to allow per object gravity