PGE Predictive Game Engine

- purpose
- to provide a simple reference model for predictive collision detection between simple 2D objects
- as an educational experiment

Overview

- a game engine will simulate a 2D environment which understands and polygons, circles
- each circle and polygon can be fixed or unfixed
- each object may be given a mass, velocity and acceleration

Overview

- PGE predicts the time of the next collision
- and draws the world for each frame
- the game engine is a discrete event simulator
    - so an event is either a collision event or a draw frame event

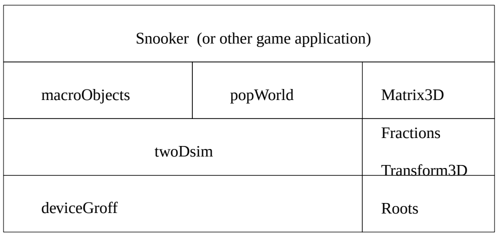
Limitations

- the game engine does not model rotation of objects
- collision response has a fixed inelastic property
    - no provision to have a per object inelastic or elastic property
    - easy to do, just not done yet
- contact resolution code could be improved

Points of note

- designed to be easy to debug
- version 1 used a `macroObject` module which allows more complex objects to be created
- the 2D world is populated via `macroObjects`
- uses a fractional data type for the render and macroObjects
    - allows for much easier debugging

Structure

- 

Snooker (or other game application)

macroObjects

popWorld

Matrix3D

twoDsim

Fractions

Transform3D

deviceGroff

Roots

Fractions

- `pge` is currently 20244 lines of code
    - `Fractions` accounts for 2973 lines of code
- nevertheless it provides good visual clues when debugging
    - much easier to spot 1/80 than 0.0125
- also knows about certain symbolic values: π, $\sqrt{2}$, $\sqrt{3}$, $\sqrt{6}$
    - symbolic numbers are only resolved once required, thus they might disappear if used together

Example of a debugging session with GDB and PGE

- let us assume there is a bug somewhere in the `macroObject_rotate` function
- an obvious way to solve this is to use `gdb` and single step the function, printing out the variable contents as they are created

---

Example of a debugging session with GDB and PGE

```
$ make npn
$ gdb a.out
(gdb) break macroObjects_rotate
(gdb) run
Breakpoint 24, macroObjects_rotate (m=0x6797f0, p=..., r=
at macroObjects.mod:562
(gdb) next
(gdb) print dmat(a)
+-
| 1  0  0
| 0  1  0
| -.1/4  -.1/4  1
+- 1 = void
(gdb) next
```

Example of a debugging session with GDB and PGE

```
(gdb) print dmat(b)
+-
| cos((pi/2))  -1  0
| sin((pi/2))  cos((pi/2))  0
| 0  0  1
+- 2 = void
(gdb) next
(gdb) print dmat(c)
+-
| 1  0  0
| 0  1  0
| .1/4  .1/4  1
+- 3 = void
```

Example of a debugging session with GDB and PGE

```
(gdb) next
(gdb) print dmat(d)
+-
| ((1*((cos((pi/2))*1)+0))+((0*((sin((pi/2))*1)+((cos((pi/2))*0)+0)))+0)) \
  ((1*((cos((pi/2))*0)+-1))+((0*((sin((pi/2))*0)+((cos((pi/2))*1)+0)))+0)) \
  ((1*((cos((pi/2))*0)+0))+((0*((sin((pi/2))*0)+((cos((pi/2))*0)+0)))+0)) \
| ((0*((cos((pi/2))*1)+0))+((1*((sin((pi/2))*1)+((cos((pi/2))*0)+0)))+0)) \
 ((0*((cos((pi/2))*0)+-1))+((1*((sin((pi/2))*0)+((cos((pi/2))*1)+0)))+0)) \
 ((0*((cos((pi/2))*0)+0))+((1*((sin((pi/2))*0)+((cos((pi/2))*0)+0)))+0)) \
| ((-.1/4*((cos((pi/2))*1)+0))+((-.1/4*((sin((pi/2))*1)+((cos((pi/2))*0)+0)))+.1
 ((-.1/4*((cos((pi/2))*0)+-1))+((-.1/4*((sin((pi/2))*0)+((cos((pi/2))*1)+0)))+.1
 ((-.1/4*((cos((pi/2))*0)+0))+((-.1/4*((sin((pi/2))*0)+((cos((pi/2))*0)+0)))+1))
+- 4 = void
```

Example of a debugging session with GDB and PGE

```
(gdb) print PolyMatrix3D_eval(d)
4 = (POINTER TO RECORD ... END ) 0x6876e0
(gdb) print dmat(d)
+-
| 0  -1  0
| 1  0  0
| 0  .1/2  1
+- 15 = void
```

Performance testing of a game engine

- let us build and run snooker

-
  ```
  $ make snooker
  gm2 -pg -g -fiso -fextended-opaque -fonlylink snooker.mod
  $ ./a.out
  ```
- notice the `-pg` flag to gm2 (the same applies to gcc)

Performance testing of a game engine

- this flag turns on runtime profiling

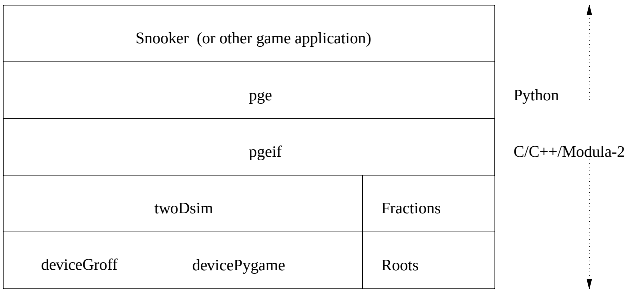
-
  ```
  $ gprof a.out
  Flat profile:

  Each sample counts as 0.01 seconds.
    %   cumulative   self              self     total
   time   seconds   seconds    calls   s/call   s/call  nam
   34.22      2.06     2.06    99486     0.00     0.00  ini
   30.15      3.88     1.82 132186249     0.00     0.00   In
   29.49      5.65     1.78 132183929     0.00     0.00   In
    1.41      5.74     0.09                             Ind
    1.08      5.80     0.07     2320     0.00     0.00  Ind
    0.50      5.83     0.03   236365     0.00     0.00  unM
  ```

Useful to profile version 1 of PGE

- version 1 was completely implemented in a 3rd generation language (Modula-2)
- we can profile all this code and optimize the hotspots
- as above the `InBounds` was optimized (removed) and this gave a 30% performance improvement
- version 1 did not link up to Pygame and the game had to be written in Modula-2 as well
- version 2 interacts with Pygame and has a Python interface

Structure of version 2 PGE

- 

Conclusion to the construction of version 2 of PGE

- implemented in Modula-2, C, C++ and Python
- the Modula-2 code is translated into C or C++ code
  - the translated code conforms to GNU coding standards and is very neatly formatted
- the Python interface documentation ⟨`http://floppsie.comp.glam.ac.uk/Southwales/gaius/pge/homepage.html`⟩ is available on line

---

Obtaining and building pge for the coursework

- you can either obtain pge from the debian package - or from the git repository
  - I'd advise the git repository as it will contain very minor incremental improvements
-
```
$ cd
$ mkdir -p Sandpit
$ cd Sandpit
$ git clone https://github.com/gaiusm/pge
```

Building PGE

- you can build a local copy by:
-
```
$ cd
$ mkdir -p Sandpit
$ cd Sandpit
$ rm -rf build-pge
$ mkdir build-pge
$ cd build-pge
$ ../pge/configure --prefix=$HOME/opt --enable-langc
$ make
```

Testing your local copy of PGE

```
$ cd
$ cd Sandpit/build-pge
$ ./localrun.sh ../pge/examples/breakout/breakout.py
```