The Construction of a Predictive Collision 2D Game Engine

Gaius Mulley
Department of Computer Science
University of South Wales
CF37 1DL
e-mail: gaius.mulley@southwales.ac.uk

***Abstract*—This paper discusses the construction of a game engine which is based around the principle of discrete event simulation. This work is interesting as it uses a predictive time of collision rather than a frame based approach. The key design decisions made and the tools used during the construction of the predictive game engine (PGE) are described. The objects modelled in the game engine are rigid circles and polyhedra which may have an orbit rotational velocity, a positional velocity and acceleration. Equations calculating the time of next collision between two circles, a line and a circle and two lines are given. If orbit rotation is excluded, the two circles expansion is simple as the output from maxima is only twelve lines, conversely when orbit rotation is included the output exceeds 835 lines. The input formula to maxima used for detecting the time of collisions between moving polyhedra are presented alongside a tool to automatically import the expanded formula into program code.**

Collision prediction; game engine; gm2; shared library; Python; SWIG.

I. Introduction

While frame based game engines have been well documented in [1] [2] and [3] there is less published about game engines using a predictive collision approach [4].

An event based collision prediction game engine has many advantages: it perfectly matches the well known technique of discrete event simulation and potentially it can be implemented more efficiently than the frame based approach. It also offers greater levels of accuracy over the more popular frame based equivalent.

An event based technique can serve certain classes of games well. For example a multi-user real-time game might be split into server and client. The server, running on the cloud, could employ the event based collision prediction game engine whilst clients running on mobile devices with limited power would run little more than the GUI and network protocol stack. Adopting the technique of discrete event simulation allows the client to reduce its power requirement considerably given it can become idle between events.

Published in Eurosim 2013, 8th Modelling and Simulation Conference, Cardiff, Wales 10-13 September 2013, Editors Khalid Al-Begain, David Al-Debass, Allessandra Orsoni, Richard Cant, Richard Zobel.

II. The problem

The system of interest is a game engine which models a 2D world of rigid bodies: polyhedra based objects and circles. An object may be moving with a positional velocity, positional acceleration and rotational velocity. The game engine needs to calculate the next time of collision between any moving object and another object.

The polyhedra objects are either composed of circles or line segments. Thus the collision prediction module needs to know the time of next collision between either a circle and another circle, a line segment and a circle or a line segment and another line segment.

In the simplest of these cases, namely the time of collision between two non orbiting circles, the solution can be found from basic equations of motion where the collision value is the smallest positive root after solving a quartic polynomial. The polynomial equation is twelve lines of output from maxima [5] and thus such a solution can be hand programmed with care.

Conversely when independant rotational velocity is introduced to this two circle case, maxima generates an octic polynomial equation of 835 lines. This is just one of the variants of expressions required to model the collision of polyhedra and circles. Clearly coding and debugging a game engine using collision prediction is non trivial.

III. Structure of pge

PGE is an event based collision prediction game engine and its structure is shown in figure 1.

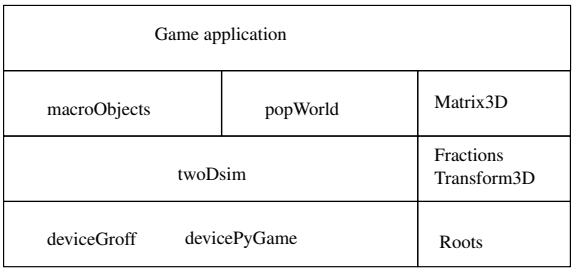

Figure 1. The structure of PGE

Aside from the game engine related data types PGE also introduces a symbolic fractional data type. While this data type is not used directly in the calculation of the

object collision time module `twoDsim`, it is used by the `Game application`, `macroObjects`, `popWorld`, `matrix3D`, `transform3D`, `deviceGroff` and `devicePyGame` modules. The symbolic fractional data type module `fractions` will also resolve simple symbolic fractions when required. The advantage of this approach is threefold: first the groff device can use fractional measurement for glyph placement and size; second it allows the game application to specify input measurements as fractions but third and most importantly it allows for much more intuitive debugging.

Using GDB [6] and some debugging routines in the `fractions` and `Matrix3D` module the developer can inspect a value and display how a value was calculated. An example of its usefulness is in the debugging of the `Matrix3D` module where matrices are interactively dumped with their transform formulae. The inefficiency of this data type is not noticed as the module is only used to populate the game environment at the start and also during the output of each `groffDevice` frame. For normal interactive output the equivalent of the `float` data type is used. The collision calculations are achieved using `float` or `double` precision.

IV. Collision prediction between objects

PGE allows polygons and circles to be created dynamically with mass, velocity, acceleration and orbit rotational velocity attributes. The orbit rotational velocity is described by centre of rotation and velocity components. The centre of rotation may be outside the object.

The world modelled by PGE consists of circles and polygons. The collision prediction algorithm breaks the polyhedra into line segments and finds the smallest positive time of next collision. The algorithm is required to resolve the time of collision between the following pairs of objects: circle/circle, line/circle and line/line. This can again be subdivided by first solving the circle/circle case and then solving the point/line case.

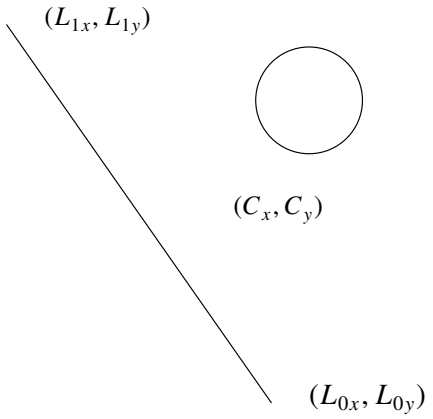

Figure 2. Circle and line

Thereafter the line/circle collision problem presented in figure 2 can be transformed into a point/circle and point/line problem in figure 3.

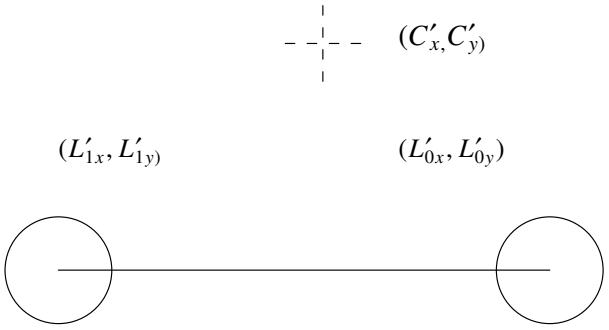

Figure 3. Point, circles and line

Likewise the line/line problem presented in figure 4 can be solved by testing four point/line collisions (whether the end point of each line crosses the other line).

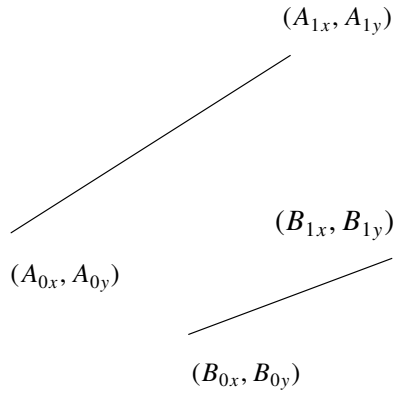

Figure 4. Line and line

The algorithm transforms the diagram so that the line (B_{0x}, B_{0y}) (B_{1x}, B_{1y}) is now resting on the $y = 0$ axis. It then uses the formula expressed in figure 9 to determine when the point (A_{0x}, A_{0y}) crosses $y = 0$. Finally the algorithm checks to see whether the x value of intersection is $\leq B'_{1x}$. If this condition holds then this is a possible event candidate and the time value is stored together with the appropriate line details. The advantage with this approach is that it can be solved with only 4 polynomial terms.

V. Collision equations described in max2code

This section describes the principle equations in `max2code` for the building blocks mentioned in the last section. Figure 5 shows the beginning of the source file for the tool `max2code`, the keywords are in the bold

font.

The tool max2code parses the input file, and stores the initialisation, finalisation code and term designaters. The algebraic text between the **begin** and **end** is sent to maxima to be transformed into a polynomial form. Thereafter max2code parses the output from maxima's polynomial form and collects like polynomial terms. max2code now emits the initialisation code, followed by each polynomial designater and polynomial expression pairing (in this case a[0], a[1] etc). Lastly the finalisation code is emitted.

```
polynomial terms 8
initialise {
VAR
   t: REAL ;
   a: ARRAY [0..7] OF REAL ;
BEGIN
}
term 0 { a[0] := }
term 1 { a[1] := }
term 2 { a[2] := }
term 3 { a[3] := }
term 4 { a[4] := }
term 5 { a[5] := }
term 6 { a[6] := }
term 7 { a[7] := }
finalise {
  t := findSmallestPositiveRoot(a)
}
```

Figure 5. First section of colliding circles in max2code

There are a number of advantages of using max2code; the first is maintenance. The developer can express the complex equations in smaller components and let algebraic tools manipulate them into program code. Second the choice of programming language now becomes more fluid. Third, the code is much easier to debug, as max2code emits gcc style file and line directives. The debugger will seamlessly single step from C and Modula-2 source into this source file at appropriate points.

The input file fragment shown in figure 10 expresses the equations of motion for an arbitory point and also an end point on a line. Both points are moving independently with acceleration, velocity and orbital rotation. The first section of this file is shown in figure 9. The input variables expected for the point on line collision expression are given in figure 8.

VI. Output of max2code

While the equations of motion do not appear complex when expressed in this form, once they have been expanded into their polynomial form and then converted into program code the length of the expressions dramatically increase.

Comparison of expression complexity (lines of code)			
Expression type	max2code	maxima	Modula-2
Two circles	33	835	977
Point and line	34	18	24

Figure 6. Table showing complexity comparison of equations

A comparison table between the collision prediction equations used for circle/circle and point/line is given in figure 6. The equation for the time of orbiting circles collision uses 8 polynomial terms whereas the point on line only uses 4 polynomial terms.

```
begin
/* cos Taylor's expansion up to
   the 3rd term */

O(X) := ''1 - X^2/2 + X^4/4 ;

/* define sin using cos, as it
   uses smaller powers */

S(X) := ''O(%pi/2-X) ;

/* x pos of circle 1 */
C: (a + b*t + c*t^2/2 + d*S(e*t+f));
/* y pos of circle 1 */
D: (m + n*t + o*t^2/2 + d*S(p*t+q));
/* x pos of circle 2 */
E: (g + h*t + i*t^2/2 + j*S(k*t+l));
/* y pos of circle 2 */
F: (r + s*t + u*t^2/2 + j*S(v*t+w));
A: (C - E);
B: (D - F);

/* radius of circle 1 and 2 */
Z: (x+z);

load(format)$

/* Pythagorean distance */

A^2 + B^2 - Z^2;

collectterms(%,t);
ratsimp(%);
format(%, %poly(t));
end
```

Figure 7. Last section of colliding circles in max2code

Input variables relating to one end of the line	
a	is the initial y position
b	is the initial y velocity
c	is the y acceleration
d	is the orbit radius of the line
e	is the angular velocity
f	is the angular offset (relative to the c of g).
Input variables relating to the point	
g	is the initial y position of the point.
h	is the initial y velocity.
i	is the y acceleration of the point.
j	is the orbit radius of the point to its (c of g).
k	is the angular velocity.
l	is the angular offset (relative to the c of g).

Figure 8. The input variables for point on line collisions

```
polynomial terms 4
initialise {
VAR
   t: REAL ;
   a: ARRAY [0..3] OF REAL ;
BEGIN
}
term 0 { a[0] := }
term 1 { a[1] := }
term 2 { a[2] := }
term 3 { a[3] := }
finalise {
  t := findSmallestPositiveRoot(a)
}
```

Figure 9. First section of point on line in max2code

```
begin
/* cos Taylor's expansion up to the
   3rd term */

O(X) := ''1 - X^2/2 + X^4/4 ;

/* define sin using cos, as it uses
   smaller powers */

S(X) := ''O(%pi/2-X) ;

load(format)$

/* we multiply the equation by 64
   to remove a division */

((a + b*t + c*t^2/2 + d * S(e*t+f))
        -
 (g + h*t + i*t^2/2 + j * S(k*t+l)))
        * 64 = 0 ;

expand(%);
ratsimp(%);
format(%, %poly(t));
quit();
end
```

Figure 10. Last section of colliding point on line in max2code

```
$ max2code --lang=m2 point-line.mxm \
 -o point-line.mxd
$ max2code --lang=m2 circle-circle.mxm \
 -o circle-circle.mxd
$ gm2 -g -c collisions.mod
```

Figure 11. Command line for building the collision detection module

VII. Construction of PGE

The game engine is written in Modula-2 [7] but the game application can be written in any SWIG [8] compatible scripting language [9], such as Python. Modula-2 was an ideal language for implementing the game engine as the gm2 compiler [10] provides command line switches to automatically convert a well behaving definition and implementation module into a Python module [11].

The tool `max2code` provides a method for expressing the collision prediction code in very a high level form. Therefore it is easy to maintain variants of the collision prediction equations presented in figure 7 and figure 10 which involve no orbit of rotation for one or more objects. This in turn optimises PGE considerably due to the reduction in polynomial terms which require solving. PGE solves quadratic, cubic and quartic roots internally, but uses the GNU scientific library to solve octic roots. PGE consists of 17560 lines of Modula-2 source code. The collision detection module was built by the command line sequence shown in figure 11.

At the core of PGE is a discrete event simulator which uses three classes of events: input event, draw frame event and collision event. The draw frame event either creates a new frame buffer under PyGame or groff. The groff frames were generally used for debugging or for demonstration purposes by a pipeline of gs, png conversions and mencoder to produce an anti-aliased animation of the 2D model.

VIII. Conclusions and further work

In conclusion the technique of using collision prediction in a game engine is shown to be feasible. Indeed, in certain game applications it might be more efficient than the frame based approach. In the future with the advent of low cost, low power, highly parallel hardware where each core is not aggressively fast this approach becomes very advantageous since it scales well. Each movable object could be assigned a different processor as the time of collision for each object can be calculated independently.

The tool `max2code` was found to be very useful in

maintaining and implementing PGE. In particular it is now feasible to maintain variants of the collision equations for objects which have no orbital rotation a radius of zero or no acceleration. In the future it might be possible to request `max2code` to derive the variants and also issue the selection criteria in the desired progamming language.

It is particularly pleasing to see how well this game engine integrated with PyGame and it would be interesting to implement a pedagogical platform game using this approach. If a general solution to octic equations [12] is found then the approach taken by PGE becomes even more attractive.

REFERENCES

[1] Millington, Ian, "Game Physics Engine Development: how to build robust commercial-grade physics engine for your game" (ISBN 978-0-12-381976-5), pp. 113-143, Morgan Kaufman (2010).

[2] Ericson, Christer, "Real Time Collision Detection" (ISBN 978-1-558860-732-3), pp. 7-21, CRC Press, Taylor and Francis Group (2004).

[3] Eberly, David, "Game Physics" (ISBN 1-55860-740-4), pp. 240-348, Morgan Kaufman (2004).

[4] Schomer, Elmar and Thiel, Christian, "Efficient collision detection for moving polyhedra," *In Proc. 11th Annu. ACM Sympos. Comput. Geom,* pp. 51-60 (April 1995).

[5] Maxima, *Maxima Manual,* Ver. 5.30.0 (2013).

[6] Stallman, Richard, Roland Pesch, Stan Shebs, et al, *Debugging with gdb,* Free Software Foundation (2013).

[7] ISO/IEC, "Information technology - programming languages - part 1: Modula-2 Language," *ISO/IEC 10514-1* (1996).

[8] Beazley, D., "SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++," *The 4th Annual Tcl/Tk Workshop,* Monterey, CA. (July 1996).

[9] Beazley, D., "Using SWIG to Control, Prototype, and Debug C Programs with Python," *The 4th International Python Conference* (June 1996).

[10] Mulley, G., *The GNU Modula-2 front end to GCC* Edition 1.0, Free Software Foundation, 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA (2010).

[11] Mulley, G., *Exploiting front end knowledge to effortlessly create Python modules* (2010). `http://gcc.gnu.org/wiki/summit2010`.

[12] Kulkarni, Raghavendra, "On the Solution to Octic Equations," *The Montanna Mathematics Enthusuast* 4(2), pp. 193-209 (2007).