DESIGN

OF A

A RELATIVISTIC ROBOT NAVIGATION SYSTEM

AND ITS

IMPLEMENTATION

HASSAN AÏT-KACI

School of Computing Science
Simon Fraser University
Burnaby, British Columbia
V5A 1S6, Canada

+1 (604) 291-5589
hak@cs.sfu.ca
http://www.isg.sfu.ca/~hak

November 12, 2006

Contents

1 Purpose of this work

This work means to illustrate a methodology that I propose for designing object-oriented software based on a *relativistic* paradigm (see Section A) to obtain, with a minimal set-up, a large collection of algorithms which can all be obtained as derived classes and instance objects of a single very abstract scheme. One then may thus explain and exploit the convenience of object-orientation—as supported by, *e.g.*, C++ (*viz.*, multiple inheritance, template classes and functions, and operator overloading)—for obtaining an easy generic setup of classes implementing a robot navigation system.

This document is a partial specification of an Application Program Interface (API) in the form of a few generic classes. This specification is sketched below, along with all the explanations needed to understand it.

If implemented correctly, this API can be used to navigate robots in a fixed area to avoid obstacles on their paths to a specified target. The relativistic approach enables easy extensibility (new obstacles are easy to introduce), and efficient (the "thinking" is distributed and done by the obstacles[1] each being aware of its own geometry and orientation in the set of reference attached to the room they are in).

2 Anecdotal Background

The best anecdote that I have read[2] giving a convincing conception of object-orientation is attributed (I think!) to Alan Kay, the inventor in the 70's of the *dynabook*, a precursor of the modern GUI desktop model...

Alledgedly, in the 70's when Alan (Kay, or whoever that real or imaginary person may have been, if at all), then a graduate student at the University of Utah's Computer Science department—one of the best in the world in Computer Graphics research—had the opportunity to spend a summer as a programmer at the Stanford Artificial Intelligence Lab (SAIL). His task was to program a robot to navigate in a room cluttered with obstacles of various geometric shapes. The robot starting at some position was to move to a specified target position, but avoid any obstacle on its way with minimum deflection in its trajectory toward the target.

Supposed, Alan tried very hard to develop a large program for the robot to perceive, recognize, and avoid, obstacles of all shapes and orientations... He failed miserably to do a good job as the task was rendered even more complex when new kinds shapes of obstacles were introduced. Indeed, the robot's navigation program (essentially a huge `switch` statement) had to be rewritten for new obstacles. However, upon drawing the conclusion of his failed attempt, Alan had the following epiphany... *It was silly and inefficient to program the robot! Programming the obstacles to avoid the robot was much easier, faster, and simpler to extend with new obstacles!...*

In fact, this realization corresponds exactly to the object-oriented conception of the navigation system. Clearly, distributing the processing to the objects (the obstacles) rather than centralizing it in the subject (the robot) is *orienting* the computation toward the objects, and away from the subject.

3 Basic Setup

We now proceed with describing a basic setup to specify such a system as above. In fact, Einstein's GRT [1] is more than a metaphor. In programming the obstacles to avoid the robot, the actual "moving" should of course not be done by the obstacles. The idea is that each obstacle is an object in the same context as the robot's (*i.e.*, the room) and is "aware" of its own geometry and orientation (*e.g.*, a cube knows that it has a convex square base, and that it faces a fixed direction in the set of reference attached to the room).

4 Basic Specification

The following is an algorithm that uses a simple deflection method for computing the robot navigation around the obstacles. This algorithm is guaranteed to enable the robot to reach its target after a finite number of iterations of the main deflection method computed by an obstacle on the way of the robot to the target, if the following assumptions hold:

1. the shapes of obstacles are *convex*;[3]
2. there is a minimum *clearance width* between the obstacles, and between them and the room's walls, which must be greater than the robot's own "body" width on *all* possible paths between the robot and the target.[4]

Although the deflection method shown next treats the case of one obstacle only, this is without loss of generality. Indeed, our assumptions entail that there is at most one obstacle *closest* to the robot following the direction to the target.

1. Let $\langle x_r, y_r \rangle$ denote the robot's current location.
2. Let $\langle x_t, y_t \rangle$ denote the target's current location.

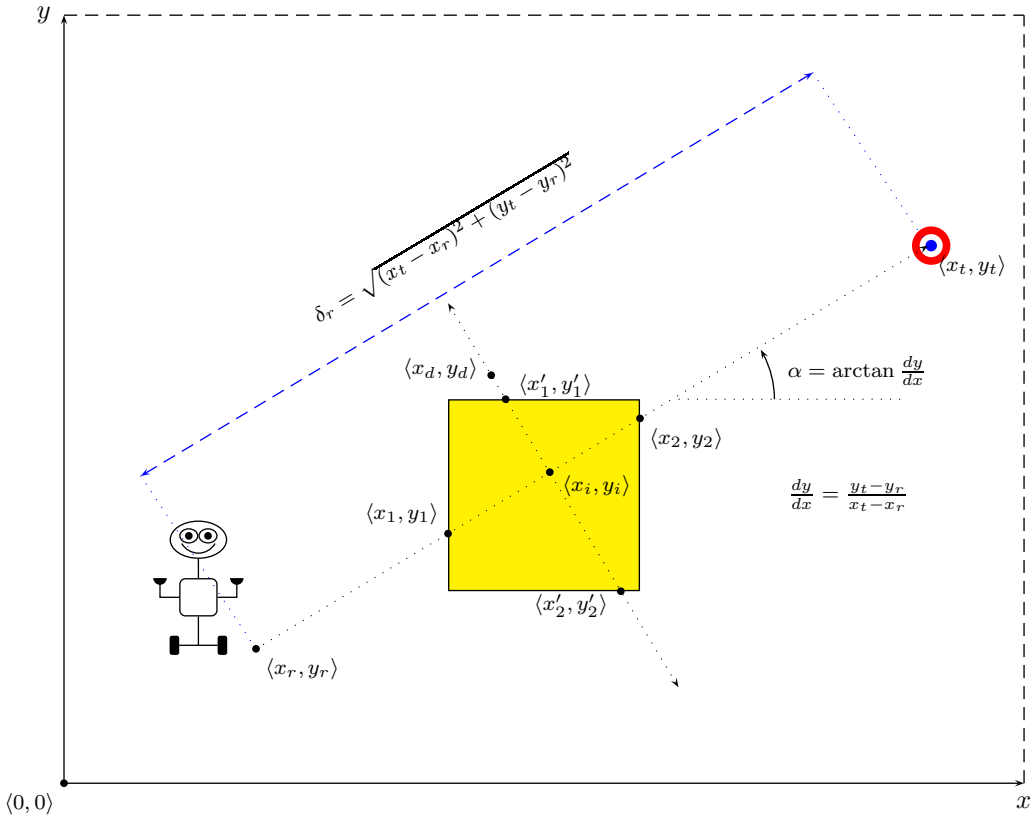
[1] . . . Not the robot, who is in fact doing *no thinking at all!*

[2] . . . Or heard—I do not recall where or from whom

[3] But this is without loss of generality, since one can use the *convex hull* of any non-convex shape. However, it is computationally more expensive, and may yield non-optimal paths.

[4] In other words, there must be sufficient room between obstacles (and the walls) for the robot to move freely between. We must set, or relate, this clearance to the ε clearance constant used in the algorithm to follow.

Description: Robot's room
Snapshot Number: 0



3. $\delta_r = \sqrt{(x_t - x_r)^2 + (y_t - y_r)^2}$ is the distance between the robot and the target.

4. $\alpha = \arctan \frac{y_t - y_r}{x_t - x_r}$ is the angle from the x-axis to the straight line from the robot to the target.

5. The parameters x_r, y_r, and α are used by the robot to compute its move towards the target disregarding the obstacle; namely, the robot moves to position $\langle x_r + \delta_r \cos\alpha, y_r + \delta_r \sin\alpha \rangle$. Therefore, the robot *always* moves using the method (in C++ syntax):

```
Robot::move()
{
  xR += deltaR * cos(alpha);
  yR += deltaR * sin(alpha);
}
```

6. If no obstacle is intersected by the straight line between the robot and the target,

   (a) then the robot moves straight to the target unhindered, and the algorithm terminates;

   (b) else, the obstacle (assumed convex) is intersected by the straight line between the robot and the target,

      i. then the obstacle's method `intersect` returns `true` and computes the two points on its border called the *crossing points* $\langle x_1, y_1 \rangle$ and $\langle x_2, y_2 \rangle$.

      ii. Let $\langle x_i, y_i \rangle = \langle \frac{x_1+x_2}{2}, \frac{y_1+y_2}{2} \rangle$ be the midpoint between the crossing points; and let the line through $\langle x_i, y_i \rangle$, perpendicular to the robot-target line, cross the obstacle's border at $\langle x'_1, y'_1 \rangle$ and $\langle x'_2, y'_2 \rangle$.

      iii. Choose one of these two points; say,$\langle x'_1, y'_1 \rangle$.[5] Let

$$\langle x_d, y_d \rangle = \langle x'_1 - \varepsilon \sin\alpha, y'_1 + \varepsilon \cos\alpha \rangle$$

      be the new *deflection point*, where $\varepsilon > 0$ is a small parametric *clearance* constant.

      iv. Set $\langle x_t, y_t \rangle = \langle x_d, y_d \rangle$, and go to Step 2.

5 Enhancements and Variations

6 Graphical User Interface

[5] This is a non-deterministic choice; it can be either points. However, not all *strategies* of choice will guarantee convergence. The simple strategy of always choosing the point closer "as the crow flies" to the target works.

Appendix

A Relativity principle as a metaphor for OO programming

The essence of object-orientation coincides with that of Einstein's Special and General Relativity theories [1].

Einstein's Special Relativity Theory (SRT) is all based on the observation that there is a mathematical duality between being at rest on one hand, and being in motion on the other hand: all motion is relative to a set of reference. Hence, it is mathematically irrelevant whether I sit in a train moving along with it at some speed with respect to the scenery, or whether I sit in a motionless train while the scenery moves by in the opposite direction at the same speed.

Similarly, Einstein's General Relativity Theory (GRT) is all based on the observation that there is a mathematical duality between free-falling frictionless in a straight line on one hand, and the texture of space being warped by massive bodies on the other hand: the curvature of all trajectory of motion is relative to space's own curvature. Hence, it is mathematically irrelevant whether the Earth is orbiting the Sun elliptically in a closed curve, or whether it free-falls frictionless indefinitely in a straight line, while space in which it moves is itself curved by the same opposite factor into the (hyper) elliptical (hyper) "eddy" created by the Sun's gravity.[6] Thus is GRT the key to explaining the mystery of "action at a distance" of gravity.

Similarly as well, object-orientation (OO) is based on the observation that there is a mathematical duality between an object being acted upon by a function on one hand, and a function being acted upon by an object on the other hand: the *orientation* of $f(x)$ is relative to the structure of interpretation of the object or the function. Hence, it is mathematically irrelevant whether the function f is applied to the object x, or whether the object x is *sent* the *message* f. In the first case (the conventional view), the function f *knows* what to do with an object of the type of x and performs it on x; in the second case (the *object-oriented view*), the object x *knows* what to do when it is asked to respond to the message sent to it as f, and performs it. Thus is OO the key to a new *decentralizing* view of computation which allows *distributed* computation and code modularity: whereas the conventional view's *centralizing* computation in functions made them huge, inefficient, and quickly impractical to maintain, the (mathematically equivalent) OO view now delegates computation to objects by making them react to messages sent to them by using methods specified for them by their class definitions.

Thus, object-orientation may simply be construed as exploiting a mathematical relativity principle. This relativistic view can be used as a systematic object-oriented software design methodology.

[6] "Hyper" because space is at least 3-dimensional. . .

To be precise, the change of perspective, when orienting computation with *reference* to an object rather than a function, is expressed mathematically by the set isomorphism:

$$A \to (B \to C) \simeq B \to (A \to C). \tag{1}$$

This equation essentially captures the dual *relativity* of computation alluded to above.

This article is an example of the general case, which can be expressed as follows:

$$\begin{aligned} &\mathit{method} : \mathit{Context} \to (\mathit{Object} \to \mathit{Object}) \\ &\qquad\qquad \simeq \\ &\mathit{method} : \mathit{Object} \to (\mathit{Context} \to \mathit{Object}). \end{aligned} \tag{2}$$

Therefore, we can define two class structures, `Object` and `Context`, which *always* respectively declare a method (here called `method`) as shown in Figures 1 and 2.[7] Some examples are given in Figure 3.

References

[1] Albert Einstein. *Relativity—The Special and the General Theory*. Crown Publishers, Inc., New York, NY, 1961.

[7] Using C++ syntax:.

```
class Object
  {
    virtual Object *method (Context *context);
  }
```

Figure 1: Object class skeleton

```
class Context
  {
    Object *method (Object *object) { return object.method(this); }
  }
```

Figure 2: Context class skeleton

Context	**Object**	**method**
`Name_Value_Environment`	`Expression`	`evaluate`
`Name_Type_Environment`	`Expression`	`typecheck`
`Run_Time_Environment`	`Instruction`	`execute`
`Algebraic_Structure`	`Equation`	`solve`
`Logical_Theory`	`Theorem`	`prove`
`Constraint_Structure`	`Constraint`	`resolve`
`Windows95`	`Windows_Application`	`WinMain()`/window function

Figure 3: Some instances using the context/object relativity principle