An elegant time-management system for roguelikes

by Henri Hakl

Contents

Introduction

While browsing the roguelikedevelopment articles I've noticed a sore lack of information on good time management for roguelike games. This text tries to fill that void.

This is bound to be subject to some debate, but in my opinion the best time management system is used in ADOM: Creatures have speed and actions cost a certain amount of energy/time. This ensures that a creature with speed 102 is slower than a creature with speed 103, provided both compete in an activity that costs the same for both.

Unfortunately not all roguelikes adhere to such a system - which results in creatures that are always as fast as the player, significantly slower, or significantly faster - with no room for nuances. In Crawl, for example, creatures with speed 100 and 101 are indistinguishable; it is practically impossible to run away from creatures with the same speed-class as the player - nor can monsters catch-up to a player.

This article presents a compact and elegant solution to deliver ADOMesque time-management. The system not only caters for player and monster events - but any time-based or periodic events. For example, it is readily possible to use the time-manager for in-game day/year tracking, hunger escalation, enchantment duration, etc

Coding Background

Before we begin with the details it is necessary to describe the primary vehicle which powers the time-manager: A circular doubly-linked list with traveling sentinel.

Readers should be familiar with a linked list (next) and doubly-linked list (prev and next). A circular list simply means that the last object's "next" points to the first object; and in a doubly-linked circular list the first object's "prev" points to the last object. The advantage of a circular list over a regular list is that a circular list need not cater for null-references during insertion or deletion of objects.

A "sentinel" in a list is a fixed, known object that is usually placed into a circular linked list to indicate start-and-end position. When iterating through the list the sentinel is used as starting point and the break condition.

The idea of a "traveling sentinel" simply indicates that the position of the sentinel in a list may change:

```
.. -> x -> Sentinel -> a -> b -> ..
```

Travel:

```
.. -> x -> a -> Sentinel -> b -> ..
```

The time-manager described here uses the traveling sentinel to iterate through the list of timed entities.

Coding

Unfortunately the PC I'm working on currently only supports Turbo Pascal, so the code presentation will illustrate the system with old-school code.

First the Type definitions:

```
RateProc = function(p : pointer) : integer;
CostProc = function(p : pointer) : integer;

PTime = ^TTime;
TTime = record
  cur  : integer;    { current time track state }
  trg  : pointer;    { target for callback }
  rate : RateProc;   { callback function to return rate }
  cost : CostProc;   { callback function to return cost }
  prev : PTime;      { previous record in list }
```

```
    next : TTime;      { next record in list }
  end;
```

The publically available interface:

```
  procedure RegisterTimedEntity(c : integer; p : pointer; f : RateProc; g : CostProc);
  procedure ReleaseTimedEntity;
  procedure ProgressTime;
```

The private variables:

```
  TimeSentinel : TTime;
  releaseTime  : PTime;
```

The implementation:

```
  procedure RegisterTimedEntity(c : integer; p : pointer; f : RateProc; g : CostProc);
  { allocates and initializes a timed entity, and inserts it into circular linked list }
  var
    t : PTime;
  begin
    GetMem(t, sizeof(TTime));       { allocate space }
    t^.cur  := -c;                     { initialize current time track state}
    t^.trg  := p;                   { set return parameter }
    t^.rate := f;                   { set rate callback }
    t^.cost := g;                   { set cost callback }

    t^.prev := @TimeSentinel;       { insert t into circular linked list }
    t^.next := TimeSentinel.next;
    t^.next^.prev := t;
    TimeSentinel.next := t;
  end;

  procedure ReleaseTimedEntity;
  { remove TimeSentinel.next from circular list and store for later deallocation }
  begin
    releaseTime := TimeSentinel.next;          { store entity that will be removed }
    TimeSentinel.next := releaseTime^.next;    { remove entity  from circular list}
    TimeSentinel.next^.prev := @TimeSentinel;
  end;

  procedure ProgressTimeSentinel;
  { travels the sentinel through circular list, deallocates a released time entity if necessary }
  var
    x, a, b : PTime;
  begin
    if releaseTime <> nil then begin
      FreeMem(releaseTime, sizeof(TTime));   { deallocate released time entity }
      releaseTime := nil;
    end else if TimeSentinel.next^.next <> @TimeSentinel begin
      x := TimeSentinel.prev;                { travel TimeSentinel one step forward }
      a := TimeSentinel.next;
      b := a^.next;
      a^.prev := x;
      x^.next := a;
      TimeSentinel.prev := a;
      TimeSentinel.next := b;
      a^.next := @TimeSentinel;
      b^.prev := @TimeSentinel;
    end;
  end;
```

```
  procedure ProgressTime;
  { progresses through circular list and update time entity }
  begin
    if TimeSentinel.next <> @TimeSentinel then begin
      t := TimeSentinel.next;
      t^.cur := t^.cur + t^.rate(t^.trg);     { update the current time value based on entity speed/rate }
      while t^.cur >= 0 do begin
        t^.cur := t^.cur - t^.cost(t^.trg);   { while sufficient energy is present, perform actions }
      end;
      ProgressTimeSentinel;
    end;
  end;
```

The initialization:

```
  TimeSentinel.next := @TimeSentinel;
  TimeSentinel.prev := @TimeSentinel;
  releaseTime := nil;
```

Usage

Running the time-manager

```
  while not done do ProgressTime;
```

This encapsulates the main game loop - it continues until the "done" variable is set - for example by a "Q"uit command from the user-interface. The user-interface is part of the cost-function of the Player timed entity.

Adding a timed entity

```
  RegisterTimedEvent(InitialTime, Monster, MonsterSpeedFunc, MonsterAIFunc);
```

Removing a timed entity

Removing a timed entity is done by calling "ReleaseTimedEntity", this should *only* be done within the Rate or the Cost function of the the timed entity.

Example:

The example shows how the time-manager can accurately keep track of in-game day/year time, it accurately keeps track of how time passes irrespective of the speed of the player and the amount of energy (cost) his/her actions require:

```
type
  PGameTime = ^TGameTime;
  TGameTime = record
    sec  : integer;
    min  : integer;
    hour : integer;
    day  : integer;
    year : integer;
  end;
```

```
function GameTimeRate(p : pointer) : integer;
{ called by time-manager to determine progress of time for entity }
begin
  if done_with_game then ReleaseTimedEntity;  { will deallocate time entity from manager }
  GameTimeRate := 100;                        { game time progresses at "standard speed" }
end;
```

```
function GameTimeUpdate(p : pointer) : integer;
{ called by time-manager to indicate a full action is complete and new action is requested }
begin
  inc(sec);                     { update game time variables, GameTime assumes one action = one second }
  if sec = 60 then inc(min);
  if min = 60 then inc(hour);
  if hour = 24 then inc(day);
  if day = 365 then inc(year);
  sec := sec mod 60;
  min := min mod 60;
  hour := hour mod 24;
  day := day mod 365;
  GameTimeUpdate := 1000;       { cost of one action (one second) is 1000 units }
end;
```

```
var
  GameTime : TGameTime
```

```
begin
  GameTime.sec  := 0;           { initilize GameTime }
  GameTime.min  := 0;
  GameTime.hour := 0;
  GameTime.day  := 0;
  GameTime.year := 0;
```

```
  RegisterTimedEntity(1000, @GameTime, GameTimeRate, GameTimeUpdate);
end;
```

Other:

Time entities can be of virtually any type and quality:

The Player:

would be registered as a timed entity where the rate-function returns the speed of the player and the cost-function would perform the user-interface update to determine what new action the player takes and then returns the corresponding cost of the action. Like in ADOM the action cost of attacking a monster could be decreased with increasing combat skill, the cost of movement could be decreased with seven-league boots, etc

Monsters:

all monsters (or rather, all NPCs) would be registered as seperate timed entities. The rate-functions would return the relevant speed of the NPCs, and the cost-function encapsulates the AI of the NPC and returns the cost of the action that the AI has determined.

Spells/effects:

Some spells might wish to register themselves as timed entities, for example, it would be prudent to handle "Poison" using the time-manager. Enchantments are another good example, the rate-function would indicate the decay of the enchantment and the cost-function would automatically be called when the duration expires:

```
function BlessRate(p : pointer) : integer;
begin
  BlessRate := 1;
end;
```

```
function BlessCost(p : pointer) : integer;
{ when action completes "Bless" is considered expired }
begin
  RemoveEffect(et_bless, PCreature(p));                    { remove "bless" effect from target creature }
  ReleaseTimedEntity;                                      { will deallocate time entity from manager }
  BlessCost := 1;                                          { ensure that time-manager proceeds }
end;
```

```
procedure CastBless(target : PCreature);
begin
  AddEffect(et_bless, target);                             { add "bless" effect to target creature }
  RegisterTimedEvent(1000, target, BlessRate, BlessCost); { duration: 1000 rate updates (about 100 GameTime seconds) }
end;
```

Conclusion

This concludes the presentation of the time managing system, I hope the ideas in this article will carry fruit for many readers. :)

PS

the validity conditions in ProgressTime and ProgressTimeSentinel:

```
if TimeSentinel.next <> @TimeSentinel
  and
if TimeSentinel.next^.next <> @TimeSentinel
```

are not necessary if at least two timed entities are in the list (not counting the TimeSentinel); I've included them in the article for safety reasons, but I don't use them in my own implementation as I ensure that the list is sufficiently filled.

Python Implementation

For anyone who might be scared off by the pascal, here is some bare-bones Python code. This really is an elegant system and for those of us who don't need to implement our own data structures, the whole traveling sentinal thing is equivalent to just rotating a queue. In this implementation, any object you register must have a speed property and a take_turn() method that returns the cost of the action taken.

Note: This is not exactly the same as the pascal code above because I've tweaked it a bit to suit my preferences (new objects go at the end of the queue, release any object at any time, etc.) but it is essentially the same system.

```
from collections import deque

time_travelers = deque()

def register(obj):
    time_travelers.append(obj)
    obj.action_points = 0

def release(obj):
    time_travelers.remove(obj)

def tick():
    if len(time_travelers) > 0:
        obj = time_travelers[0]
        time_travelers.rotate()
        obj.action_points += obj.speed
        while obj.action_points > 0:
            obj.action_points -= obj.take_turn()
```