Script Classes for Adventure Games

Author: Jeff Stephenson
Date: 5 April 1988

SIERRA CONFIDENTIAL

Table of Contents

Introduction

An interesting feature of object-oriented languages is that simply knowing the syntax and the primitive procedures (kernel calls) of the language does not help you all that much in the actual process of sitting down and embodying an idea using the language. The basis of object oriented programming is in the hierarchical class system -- the properties of the classes, the methods which operate on them, and the inter-relationships of the classes. Thus this document, which attempts to describe the class system for the Script development system.

Note that describing the class system for Script is like trying to shoot a moving target -- the classes are constantly changing, though the rate of change does seem to be slowing down somewhat. As we gain experience, the class hierarchy will stabilize, but until then keep your seatbelts fastened, since the ride is likely to be a bit bumpy.

The most important principal to keep in mind when programming in an OOPL is that classes should be embodiments of abstract concepts, not kludged code to do something that you don't feel like thinking through. Purity of programming and concept is very important in an OOPL. Since the structure and methods of a class are inherited by all its subclasses, small changes may have wide-ranging effects. On the positive side, a well thought-out class system almost seems to program itself -- objects and classes which are accurate representations of the world which they are supposed to model behave logically in response their inputs. The programming process becomes a matter of telling objects how to behave rather than writing masses of complex code.

One rule of thumb is to not add a property or method to a class if it doesn't logically belong there -- create a subclass instead. As an example, consider the problem of moving a number of things around the screen in unison. We might decide to represent this as a Collection of Actors (see the class descriptions below) and move the Collection around the screen. This, however, requires that we add positional properties and a move method to the class Collection and write some fairly kludgy code to keep the actors in the proper positional relationship. If we do this, however, all Collections and subclasses of Collection carry around all this baggage, which really has nothing to do with the concept of a group of things.

A better way to implement this would be to define a subclass of Collection, say a Gang, which is a kind of Collection but with the positional and movement aspects incorporated. This leaves the concept of a Collection unsullied, but still leaves us with the problem of how to keep the Actors positioned properly. But by calling this concept a Gang and thinking of it in that respect rather than as just a Collection, it occurs to us that gangs are often a collection of followers following a leader. So we add a leader property to the Gang class, which will be the object ID of the Actor which is to be the leader. Now moving the group around the screen is easy -- we just move the leader and tell the followers (the elements of the Gang) to follow him:

```
(method (moveTo nX nY)
     (leader moveTo: nX nY)
     (elements eachElementDo: #setMotion: Follow leader)
)
```

We could even go on to further refine the Gang class by including code which handles a Gang without a leader by having the followers wander aimlessly about a point which is considered to be the position of the Gang.

The point here is that rather than mucking up the Collection class with a

lot of irrelevant code, we created a subclass of it with the desired extra properties and methods. In doing so, we stopped thinking of it as just some sort of abstract collection and started thinking of it as a gang, inspiring the thought of adding a leader to the gang and opening the way to many possibilities.

DON'T THINK OF CODE -- THINK OF CONCEPTS!

The RootObj Class

A RootObj is the most fundamental object possible in the Script language -- it has an identity as an object, but that's it. It is defined in the kernel, not in Script, and is the ultimate super-class of all objects. Since it is such a minimal object, it is only used as a basis for those objects in which memory size is a major consideration and which are hidden away in other objects in such a way that the fuller identity given by making them sub-classes of Object is not necessary.

In file:	in the kernel
Inherits from:	nothing
Inherited by:	Object
	InvItem

Properties:

species
: This is a pointer to the property dictionary for an object, which is contained in the defining class. Thus if two objects have equal species properties, they are both instances of the same class. Do not change this property, or the object will stop working.

superClass
: This is a pointer to the method dictionary of the object's superclass, and is used to look up a method which is not defined locally. Don't change it, or the object will stop working.

-info-
: This is esoteric bit-mapped information about the object. It is currently used to tell whether the object is a static instance or was created with new:.

The Object Class

Class Object is the super-class of the majority of the classes in Script. It defines the default behavior for these classes, ensuring that all objects will respond to a certain basic set of messages.

In file: system.sc
Inherits from: RootObj
Inherited by: All classes

Properties:

name
: This is a string representation of the name of the object or class. It is used by showSelf: to display the object. The compiler will use the symbol for an instance or class as its name unless name is explicitly set.

Methods:

new:
: Returns the ID of a new instance of the class or object. This new instance has the same methods and property values as the parent. The property values can be changed, the methods cannot.

init:
: Initializes the object. If the object is to show up on the screen (Views, Props, Actors and Egos), this is the message which will make the it show itself. The default does nothing.

doit:
: Do your thing, whatever that may be. Sometimes this is sent when the object is selected, sometimes (as for Actors) it is sent for each animation cycle. The default does nothing.

dispose:
: Dispose of the object. If the object was created using new:, this reclaims the memory occupied by that object. In this case it is an error to refer to the object ID being disposed of once the dispose: message has been sent. Subclasses of Object should make sure that they dispose of any objects which they created before doing a (super dispose:).

showStr: where
: Format a string describing the object at the storage pointed to by where. The default string is the object name.

showSelf:
: Display this object in a meaningful way. This is primarily used for debugging, in order to find out which objects are present. The default is to print the string produced by showStr:.

perform: code [args...]
: Execute code. Code must be an object of class Code with a doit: method whose first argument will refer to the object which is to perform the code. Thus, one way to implement showSelf: would be to create the Code object

```
(instance showMe of Code
     (method (doit theObj)
          (Print (theObj name?))
     )
```

```
)
```

You could then make the object foo show itself even if it did not have a showSelf: method by writing

```
(foo perform: showMe)
```

Up to four arguments (besides the Code object ID) may be passed to perform:.

myself:
Returns the object ID of the receiver. This is useful as the last in a series of messages to an object to force the entire send to return the ID of the object. For example, if you wish to add a newly created and initialized Actor to a List, you could write

```
(list add:
     ((Actor new:)
          posn:100 100
          view:5
          setCycle: EndLoop
          myself:
     )
)
```

understands: selector
Returns TRUE if the object has a method corresponding to selector, FALSE otherwise.

The Collection Class

The Collection class provides the ability to manipulate collections of objects. Objects which belong to a Collection are said to be elements or members of it. The Collection class has no particular order defined for its elements, so it should not be used for situations in which the objects should be ordered -- use a List instead.

In file:	system.sc
Inherits from:	Object
Inherited by:	List
	Set
	EventHandler

Properties:

elements
: A pointer to a kernel list (kList) of the elements of the Collection.

size
: The number of elements in the collection.

Methods:

add: element [elements ...]
: Add elements to the Collection.

delete: element [elements ...]
: Delete elements from the Collection.

eachElementDo: action [args...]
: Send the message action [args...] to each element of the Collection. There may be from zero to four arguments (args).

firstTrue: action [args...]
: Send the message action [args...] to each element of the collection in succession. When an object replies a non-zero value to the message, firstTrue: returns the object ID of that object and does not send the message to any more objects. Up to four arguments are allowed.

allTrue: action [args...]
: Returns TRUE if all objects in the collection reply a non-zero value to the message action [args...], FALSE otherwise. Up to four arguments are allowed.

contains: anObject
: Returns TRUE if anObject is an element of the collection, FALSE otherwise.

isEmpty:
: Returns TRUE if the collection has no elements, FALSE otherwise.

first: (private)
: Returns a pointer to the kernel node kNode which has the 'first' object in the collection as its value. The object can be obtained with the kernel call (NodeValue kNode). Note that since there is no order associated with a collection, the object which will be returned by first is unspecified.

next: kNode (private)

Returns a pointer to the kNode following kNode. As in first:, the object may be obtained with (NodeValue kNode). Also as in first:, the lack of ordering in a Collection means that the object which will be returned is unspecified. It is guaranteed, however, that it will not be an object which has been returned since the most recent call to first: (unless the object has been added to the Collection more than once -- see Set). If all elements in the Collection have been returned since the last call to first:, 0 is returned.

The List Class

A List is just a Collection which has a specified order to its elements.

In file: system.sc
Inherits from: Collection
Inherited by: Inventory
Set

Properties:

Methods:

add: element [element ...]
Adds elements to the end of the list in the order specified.

first:
Returns the kNode of the first element in the List.

next: kNode
Returns the kNode of the element which follows kNode in the List, 0 if kNode is the end of the List.

at: n
Returns the element (not the kNode) at position n in the List.

last:
Returns the kNode of the last element in the List.

prev: kNode
Returns the kNode of the element preceding kNode in the List, 0 if kNode is the first element of the List.

addToFront: element [element ...]
Add elements to the beginning of the List.

addToEnd: element [element ...]
Add elements to the end of the List.

indexOf: element
Return the index of element in the list. If the element is not in the list, return -1.

The Set Class

A Set is a kind of List which does not contain duplicate elements: adding an object to a Set which already contains the object does not change the Set.

In file: system.sc
Inherits from: Collection
Inherited by: EventHandler

Properties:

Methods:

add: element [element ...]
Adds each element to the set only if it is not already a member of the set.

The EventHandler Class

The EventHandler is a kind of Set that has a handleEvent method. The handleEvent method for this Class invokes the handleEvent method for each of its elements. The return is TRUE if one of the elements claimed the event, otherwise FALSE.
The cast of an adventure game is an EventHandler consisting of Views, Props, Actors, and Ego. The features list is also an EventHandler consisting of Features and (possibly) PicViews.
(see class descriptions below).

In file: system.sc
Inherits from: Set
Inherited by: none

Properties:

Methods:

handleEvent: event
Passes on the handleEvent message to each of its members and returns TRUE if one of its elements claim the event, otherwise FALSE.

The Inventory Class

The Inventory class is a List of all the objects (see the InvItem class) which may be collected by ego in a game. There are two steps to setting up the inventory.

First, in the main header file for the game (typically game.sh), create an enum statement listing all the symbols which you will use to refer to the inventory items. In order to avoid name conflicts, these are traditionally the names preceded by an 'i': a rock might be referred to as "iRock".

Next, in the main game file (the file in which you define your instance of Game), you create the instances of InvItem which are the inventory items. These are added to the Inventory (whose ID is in the global variable inventory) in the init: of the game after the (super init:). The InvItems must be added to the inventory in the order in which they are listed in the enum statement above (a block copy works wonders...).

From now on you may get the object ID of any InvItem with the at: method of Inventory by using the symbol for the item defined in the enum statement: If you wanted to move the rock from its current position to room number 15, you could write

```
((inventory at:iRock) moveTo:15)
```

The get:, put:, and has: methods of the Ego class use the enumed symbols rather than the object ID. Thus, to see if ego has the rock, you would write

```
(ego has:iRock)
```

In file: invent.sc
Inherits from: List
Inherited by: none

Methods:

showSelf: whom
: Display the items possessed by whom. This is done by putting up a window with all currently possessed items in it. By clicking the mouse on an item, the user may see a picture of it and receive a more detailed description of it. Ego's possessions are done with

```
(inventory showSelf:ego)
```

saidMe:
: Return the object ID of the first item in the inventory whose said property matches what the user typed.

The Script Class

A Script is a kind of Object which has a state, methods to change that state, and code to execute when the state changes. It is used to model a sequence of actions which should be executed by an object, such as an Actor walking to the base of some stairs, walking up the stairs, and opening a door.

In file: system.sc
Inherits from: Object
Inherited by: none

Properties:

client
: The object (a Prop, Actor, Ego, Room, etc.) whose Script this is. Used to communicate with the Script's client.

state
: The state of the Script.

start
: The desired initial state of the Script.

timer
: The ID of a timer which has been set to cue: the Script after a certain interval.

register
: A property that allows you to store what ever you want. This is very useful in a Script since a temporary variable will not hold its value longer that the life of one method, thus without this property the only way to keep a value from state to state through a Script would have been a local or global variable. If more than one value needs to be stored the register property could be used as a pointer to a Collection, List or Set.

caller
: The is where the Script keeps the ID of the object to cue: upon completion.

seconds
: This contains the number of seconds until a state change will occur. This is generally set in one of the scripts states to signal the number of seconds until a state change should occur.

cycles
: This contains the number of animation cycles until a state change will occur. This is generally set in one of the scripts states to signal the number of animation cycles until a state change should occur.

script
: The ID of this script's script if it has one. This property is filled by the setScript method and should not be set directly.

Methods:

init: [client]
: Initialize the Script entering the initial state by doing a (self changeState:start). Set the Script's client to client if present.

changeState: newState
: Change the Script's state to newState. This method is where to

put the code to be executed on state transitions. It should store the new state in the state property, then switch to the appropriate code based on the new state:

```
(method (changeState newState)
     (switch (= state newState)
          transition code...
     )
)
```

cue:

Do a changeState: to the next state. Default is to invoke (self changeState:(+ state 1))

handleEvent: event

This method of an active script is called whenever there is an input event (see class Event). If (event claimed:) is TRUE, someone else has already responded to the event. Whether or not someone else has responded, if the Script responds to the event it should claim it by doing an (event claimed:TRUE). This method can be used to add additional responses to an Actor ,Prop etc.

setScript: newScript whoCares register

A Script can also have a script. This adds subroutine style capability to scripts. If the optional parameter whoCares is present the newScript will cue: whoCares and pass on the contents its register when it is disposed (i.e. (client cue:register)). In this case, the last state of the newScript should dispose of itself i.e. (self dispose:) or (client setScript:0). The third parameter (if present) sets the newScript's register property. If a Script has a newScript the newScript will receive all the events the the parenet script receives.

The Timer Class

The Timer class implements the concept of an alarm clock, allowing you to create an object which will cue: another object after a certain interval. Timers dispose: of themselves after cue:ing their client, and always check to see if their client is still present before cue:ing it.

An important concept relating to Timers is that of game time versus real time. Real time is just what it sounds like -- real time in real seconds. Game time is time adjusted to the performance of the user's computer -- it is the same as real time on a computer which is able to keep up with the animation demands of the game, but slows down in proportion to the speed of the user's computer when it is not able to keep up.

An example may help clarify this. Say that you're writing a game in which ego has only five seconds to leave a room before a bomb blows up. If you set this time interval as real time, it may give you just enough time to get out on your nice fast 286 or 386 machine. But on a Tandy 1000, where it takes 1/5th of a second to complete an animation cycle instead of the standard 1/10th of a second, the user will only be able to go half as far and thus has no chance of leaving the room before the bomb blows. The time interval should really have been set in game time, which would have given the user the same number of animation cycles to get out.

As a rule of thumb, time intervals which are meant to be a constraint on how long the user has to do something should be set in game time, whereas time intervals which are just meant to be a delay between two events should be real time. A user with a slow machine has no desire to watch a banner screen twice as long as one with a fast machine.

In file: system.sc
Inherits from: Object
Inherited by: none

Properties:

client
The object which the Timer will cue: at the end of its interval.

cycleCnt
The number of animation cycles remaining until the Timer cue:s its client. This is used by Timers which are timing either animation cycles or game time.

seconds
The number of seconds remaining until the Timer cue:s its client. This is used by Timers which are timing real time.

lastTime
Private. Used by a timer timing real time.

Methods:

set: who nSec [nMin [nHrs]]
Create a timer to cue: who after a specified interval in game time. Set it to cue: in nSec seconds. nMin and nHrs are optional minutes and hours until the cue:.

setReal: who nSec [nMin [nHrs]]
Like set:, but the time interval is in real time.

setCycle: who cycles
    Create a timer to cue: who after cycles animation cycles.

delete:
    Private. Does the actual deletion of a Timer. You should use
    dispose:, which sets the Timer up for a delete: by the system
    later on.

The Feature Class

The Feature class is simply a location on the the screen that has a handleEvent method. This is used when an object that should have a response to events is drawn into the picture. If the global variable useSortedFeatures is TRUE all elements of the cast and features will receive the event in the order of their proximity to ego.

In file: actor.sc
Inherits from: Object
Inherited by: View
PicView

Properties:

x
y
z The position on the screen.

The x property should be set to the x coordinate of the object on the screen. The y property should be set to the y coordinate of the objects projection onto the ground if it is elevated, otherwise it's y coordinate on the screen. If the object is elevated above the ground then the amount of elevation should be placed in the z property.

Methods:

handleEvent:event

All elements of the EventHandler features receive the handleEvent message. If the feature is to receive events it should be added to the current rooms features list via the rooms setFeatures method. (i.e.) (curRoom setFeature:myFeature)
This is generally done in the init method of a room. If the global variable useSortedFeatures is TRUE all elements of the features list and the cast will receive the handleEvent message in the order of their proximity to ego.

The View Class

The View class contains the minimum functionality to put a visible object on the screen in a reversible manner. Thus, as a View, a particular view, loop, and cel may be drawn on the screen and later erased. A View, however neither cycles (like a Prop) nor moves (like an Actor).

In file: actor.sc
Inherits from: Feature
Inherited by: Prop

Properties:

x
y
: The position of the View on the screen.

z
: The Views elevation above the ground.

view
: The number of the View's current view.

loop
: The number of the View's current loop.

cel
: The number of the View's current cel.

priority
: The visual priority of the View.

underBits
: A handle to the storage for saving the background of the View. Don't monkey with this.

nsTop
nsLeft
nsBottom
nsRight
: The coordinates of the top, bottom, left, and right edges of the rectangle which encloses the current (nowSeen) cel of the View. Don't monkey with this.

lsTop
lsLeft
lsBottom
lsRight
: The lastSeen rectangle for the View -- its nowSeen rectangle from the previous animation cycle. Once again, do not touch.

brTop
brLeft
brBottom
brRight
: This is the base rectangle (or baseRect) -- the rectangle which is considered to be the base of the View for detecting collisions with it. This you can monkey with. The default for the base of a View is a rectangle whose left and right sides are the sides of the nowSeen rectangle, and whose bottom and top are the y

coordinate of the View's origin.

signal
A bit-mapped word for communicating with the kernel animation routines and various methods. Handles start- and stop- updating, fixed priorities, etc. This property should not generally be manipulated directly -- the various methods (setPri:, setCel:, stopUpd:, etc.) should be used instead.

Methods:

init:
Sets the baseRect of the View and adds it to the cast so that it will be drawn on the screen at its current position.

posn: x y z
Position the View at (x, y, z).
Note that the Actual x,y screen coordinates of the object will be x, (y - z). Thus leaving z set to zero will give the traditional x y positioning.

stopUpd:
This method sets the signal property to indicate to the interpreter that the image of the object on the screen is not to be updated anymore. This is done to speed up program execution by reducing the number of things to be animated.

startUpd:
This undoes a previous stopUpd: of an View.

forceUpd:
This does a one-animation-cycle startUpd: of the View, making a change in the View's view, loop, cel, or position visible, but leaving it in a stopUpd: condition if it was so before.

setPri: [newPri]
This is used to set the priority of a View, based on the value of newPri. Normally, a View is assigned a visual priority by the kernel based on its y position on the screen. If a call to setPri: is made with no newPri specified, the priority is fixed at the current priority of the View. If a value of newPri is specified, the priority is set to that value. A newPri of -1 is used to undo a setPri:, returning control of the priority to the kernel.

setCel: newCel
Sets the cel of the View to newCel.

setLoop: newLoop
Sets the loop of the View to newLoop.

ignoreActors: [n]
If n is absent or TRUE, this allows the View's baseRect to intersect that of Actors. If n is FALSE, the View's baseRect may not intersect that of any Actors -- the Actor will collide with it (unless the Actor has itself done an ignoreActors:). The default state of a View is equivalent to ignoreActors:FALSE, i.e. an Actor may not intersect its baseRect.

show:
Place a view that has been hidden with hide: back on th screen.

hide:

Remove the View from the screen, but not from the cast.

dispose:
Tell the kernel to remove the View from the screen. Also set a bit in signal to tell the main animation loop to remove this View from the cast after animation has been completed.

delete:
If the proper bit is set in signal, remove this View from the cast and do a (super dispose:).

showStr: where
Format a string describing the View in the storage pointed to by where. This method gives classes inheriting from View the ability to inherit the View part of showSelf: rather than reimplement it.

check: other
This method is used to check for intersection of the baseRect with other. It returns TRUE if the object does not intersect anything. The method for a View or Prop always returns TRUE.

handleEvent: event
All elements of the cast get the handleEvent: message whenever a non-direction event is produced. Only the User's alterEgo is sent direction events.

addToPic:
Add this View to the picture in an irreversible manner, then delete the View. The Views important properties are copied into a PicView object created by the system, and the View is then disposed of. Note that this meens that all methods that have been redefined for the View are lost (including, but not limited to the handleEvent method). Thus if you wish to give an addToPic a specialized handleEvent method a PicView object should be used instead.

The Prop Class

Props are Views which can cycle but not move.

In file: actor.sc
Inherits from: View
Inherited by: Actor

Properties:

cycleSpeed
The number of animation cycles between successive cels of the Prop. Normally this is 0, meaning that the Prop cycles each animation cycle. If you want the Prop to cycle more slowly, set this to a larger value.

cycler
This is the object ID of an instance of a Cycle class which determines the next cel to display for the Prop. Don't modify it directly, but use the setCycle: method to install an instance of one of the cycling classes. The object pointed to by this property is sent the message (cycler doit:) by the Prop each animation cycle.

script
The object ID of a Script associated with a Prop. If a Prop has a script, it will be called with (script doit: self) by the doit: method of the Prop each animation cycle.

timer
The object ID of any timer which is set to cue: the Prop.

Methods:

doit:
If the Prop has a script, send it the message doit: self. If the Prop has a cycler, send it the message doit:.

cue: [newState]
If the Prop has a script and newState is absent, cue: the script. If newState is present, execute (script changeState:newState).

setScript: script
Sets a Prop's script to script and initializes the script. Doing a setScript:0 disposes any current script without installing a new one.

setCel: [newCel]
Normally, the cel is set by the code pointed to by cycler. Sending the setCel: message with no newCel specified overrides cycler and fixes the cel at its current value. Sending a message with a value of newCel fixes the cel at that value. A newCel of -1 returns control to cycler.

setCycle: cycle [caller]
Sets the Prop's cycler property to an instance of a Cycle class deleting any former instance. Doing a setCycle:0 disposes any current cycle class without installing a new one. The optional caller argument may be provided for Cycle classes which terminate, in which case the caller will be cue:ed when the Cycle terminates.

The Actor Class

The Actor class is the embodiment of an animated object.

In file: actor.sc
Inherits from: Prop
Inherited by: Ego

Properties:

xLast
yLast
: The coordinates of the Actor's previous position.

xStep
yStep
: The x and y step sizes for the Actor. Default is (= xStep 3) and (= yStep 2).

heading
: The compass direction in which the Actor is headed. North (toward the top of the screen) is 0, south 180.

moveSpeed
: The number of animation cycles between motions of the Actor. Normally this is 0, meaning that the Actor moves at each animation cycle. If you want the Actor to move more slowly, set this to a larger value. By playing with this and cycleSpeed, many strange effects may be discovered.

illegalBits
: A bit-mapped word which specifies which controls an Actor CANNOT be on. The low bit ($0001) corresponds to control 0 and the high bit ($8000) to control 15. By convention, control 0 is generally accessible to all Actors, whereas control 15 is off-limits to all Actors -- thus, the default value of illegalBits is $8000.

baseSetter
: The object ID of an instance of BaseSetter which is used to compute the base rectangle of the Actor. The default for the base of an Actor is a rectangle whose left and right sides are the sides of the nowSeen rectangle, whose bottom is the y coordinate of the Actor's origin, and whose top is the Actor's yStep above the bottom. This default is used if baseSetter is 0.

mover
: The ID of an instance of a Motion class which is used to determine an Actor's path of motion. Don't modify this directly, but use the setMotion: method to set the type of motion to be executed by the Actor. The code pointed to by this property will be invoked with (mover doit: self) by the Actor in each animation cycle.

looper
: The object ID of an instance of class Code which sets the Actor's loop based on the direction in which it is headed. The code is called with (looper doit: self theAngle) by the Actor during the initialization period of any setMotion:. It uses the current position of the Actor and the direction in which the Actor is heading (theAngle) to determine which loop to display. The

default, which gets installed during the init: phase of an Actor, will handle virtually every case you'll encounter.

viewer
The object ID of an instance of class Code which sets the Actor's view. This is used, for example, when an Actor should wade or swim when on certain controls. The code pointed to by viewer is called with (viewer doit: self) and can test for the Actor being on a control and set the view accordingly.

avoider
The object ID of an instance of class Avoider. If a moving Actor collides with something (control in the picture, another Actor, etc.) which prevents it from moving, the avoider attempts to find a way around it.

Methods:

init:
Does the usual setup for a Prop, sets looper to DirLoop, and sets avoider to an instance of class Avoider for all but ego.

doit:
If an Actor has any of the following, it invokes them in order: a script, a viewer, an avoider or mover (the avoider, if present will call the mover), and a cycler.

setLoop: [newLoop]:
Normally, an Actor's loop is set by invoking the looper method, which does so based on the Actor's direction. Sending the setLoop: message with no argument fixes the loop at its current value. If newLoop is present, the loop is set to its value. If the newLoop is -1, loop determination is returned to the looper method.

ignoreHorizon: [n]
If n is absent or TRUE, this allows the Actor to be above the current Room's horizon. If n is FALSE, the Actor will not be allowed to go above the horizon of the current Room and will be repositioned to the horizon if an attempt is made to position it there. The default state of an Actor is equivalent to ignoreHorizon:FALSE.

setMotion: motion [caller]
Sets the Actor's motion property to an instance of a Motion class, deleting any former instance. Doing a setMotion:0 disposes any current motion without installing a new one. The optional caller argument is the object to be cue:ed when the motion is completed.

setAvoider: avoider
Set the Actor's Avoider to avoider. To remove an Avoider from the Actor, do a setAvoider:0.

isStopped:
Return TRUE if the Actor is in the same position as it was during the previous animation cycle, FALSE otherwise.

isBlocked:
Returns TRUE if the Actor tried to move, but was unable to because of something blocking it. If this is TRUE, the avoider will generally be invoked.

canBeHere:
Return TRUE if the Actor can be in its current position, FALSE otherwise. This checks to see whether the Actor's baseRect is on any illegal controls (specified by illegalBits), whether it has collided with a View, Prop, Actor or Ego, and whether it has collided with one of the Room's Blocks. In any of these cases, FALSE will be returned.

findPosn:
Reposition the Actor until canBeHere: returns TRUE. This is called whenever canBeHere: returns FALSE during animation. It searches in concentric rectangles outward from the Actor's position for a legal position.

check: other
Check to see if the Actor's baseRect intersects that of other. Return TRUE if it does not, indicating that the Actor has not collided with other.

distanceTo: anObject
Returns the distance (in pixels) to anObject (which had better have x and y properties). [Note: this is the real distance, not the kludge used in the previous interpreter. Because this uses the square-root function, it is slow -- don't use it 100 times per animation cycle, or you'll notice a degradation in performance.]

inRect: left top right bottom
Returns TRUE if the Actor's x and y are within the bounding rectangle, FALSE otherwise.

onControl: [origin]
Returns a bit-mapped word (mapped like the illegalBits property) indicating which controls the baseRect of the object encloses. If the optional origin argument is used (just use the word "origin"), the control under the x, y coordinate of the object is returned.

The Ego Class

The Ego class is the class of Actors which can be controlled by a User.

In file: actor.sc
Inherits from: Actor
Inherited by: none

Properties:

prevDirKey
: The key pressed by the User to start the Ego moving in its current direction. This property is needed to implement the "press the same key a second time to stop" control of an Ego, and set to -1 when a joystick or mouse starts the motion.

edgeHit
: This property is set to the edge of the screen which the Ego has moved off of, or to 0 if the Ego is still on the screen. The edges are NORTH, SOUTH, EAST, and WEST.

Methods:

init:
: Does the standard Actor init:, then a (self setCycle:Walk).

doit:
: This just passes the doit: message along to the Actor doit:, then sets edgeHit based on where the Ego is on the screen.

get: item [item ...]
: Make the Ego the owner of items.

put: item [where]
: Assuming that the Ego is the current owner of item, if where is specified, it is made the owner of item. Otherwise, the item is put in "limbo" (equivalent to a where of -1).

has: item
: Return TRUE if the Ego is the current owner of item, otherwise FALSE.

The PicView Class

Objects of class PicView keep information on Views which have been added to the picture with addToPic: so that they will be present in a restored game. In order to use the PicView class explicitly it must be added to the addToPics List and the addToPics doit method must be invoked.

```
  (i.e.
     (instance myPicView of PicView
        (properties
           view  vMyView
           x     160
           y     100
           z     10
        )
     )
     and in the rooms init method
     (addToPics
        add:  myPicView,
        doit:
     )
  )
```

In file: actor.sc
Inherits from: Feature
Inherited by: none

Properties:

view
loop
cel
x
y
z
priority
signal
: The values of the corresponding properties in the View which was the progenitor of this PicView.

Methods:

The Cycling Classes

The Cycle class is the basic class which implements cycling behavior in Actor and its sub-classes. Sub-classes of Cycle implement specific cycling behavior.

Cycle

In file: motion.sc
Inherits from: Code
Inherited by: Forward
Reverse
CycleTo

Properties:

caller
The object to be cue:ed when the Cycle completes.

client
The Prop, Actor, etc. which is being cycled by an instance of Cycle.

cycleDir
The direction in which the client's cels are to be cycled.
1 forward (cel numbers should increase)
-1 backward (cel numbers should decrease)

cycleCnt
The number of animation cycles since the current cel of the client was displayed. When this equals the cycleSpeed of the client, the nextCel: method is invoked.

Methods:

init:
Do any necessary initialization of the Cycle class and ensure that the client is updating.

doit:
Called by the client as doit: self, this method sets the cel of the client to the appropriate value based on the particular Cycle class.

nextCel:
Used by the doit: method to move the cel number in the direction indicated by cycleDir.

cycleDone:
This is invoked when the client's cel reaches a destination cel. If the Cycle class is one of the cyclic ones, this just wraps the cel to either the beginning or end of the current loop. If it is a terminating Cycle class, this cue:s the caller.

Forward

Cycles an Actor through its cels in the 'normal' order -- from 0 through the end of the loop and back to 0. Set with (actor setCycle:Forward).

In file: motion.sc
Inherits from: Cycle
Inherited by: Walk

Walk

This is a Forward cycle type which only cycles an Actor when the Actor is moving. Set with (actor setCycle:Walk).

In file: motion.sc
Inherits from: Forward
Inherited by: none

Reverse

Cycles an Actor through its cels in the reverse order of that in Forward, i.e. from the highest cel number to 0 and then repeating. Set with (actor setCycle:Reverse).

In file: motion.sc
Inherits from: Cycle
Inherited by: none

CycleTo

This class allows you to cycle to an arbitrary cel and stop. It is set by (actor setCycle:CycleTo toCel direction [caller]).

In file: motion.sc
Inherits from: Cycle
Inherited by: EndLoop
BegLoop

Properties:

endCel
The number of the cel at which the cycling is to complete.

EndLoop

This sub-class of CycleTo advances the cel of an Actor from the current cel to the cel which is the end of the current loop for the Actor, then stops and cue:s its caller (if it has one). Set with (actor setCycle:EndLoop [caller]).

In file: motion.sc
Inherits from: CycleTo
Inherited by: none

BegLoop

This sub-class of CycleTo decrements the cel number from its current value to 0, then stops and cue:s its caller, if any. Set with (actor setCycle:BegLoop [caller]).

In file: motion.sc
Inherits from: CycleTo
Inherited by: none

The Motion Classes

The sub-classes of class Motion implement the various kinds of motion which Actors can execute.

Motion

The Motion class is the basis for all the specialized motions. If a caller is specified in the setMotion: for a motion, it will be notified of the motion's completion by being sent the cue: message.

In file: motion.sc
Inherits from: Object
Inherited by: MoveTo
Wander
Chase
Follow
Jump
Orbit
Path

Properties:

x
y
The coordinates towards which the Actor is moving.

client
The Actor being moved by the instance of motion.

caller
The object to cue: when the motion is complete.

dx
dy
b-moveCnt
b-i1
b-i2
b-di
b-xAxis
b-incr
These properties are used internally by the modified Bresenham line algorithm which moves the Actors.

Methods:

moveDone:
Executed when the motion completes or is blocked. Cue:s the caller, if there was one, and dispose:s of the motion class.

MoveTo

The MoveTo class just moves an object to a particular position. An Actor is moved to a position with (actor setMotion: MoveTo x y [caller]).

In file: motion.sc
Inherits from: Motion

Inherited by: none

Wander

The Wander motion class implements a random wander for Actors. It is invoked for an Actor by (actor setMotion: Wander [distance]) where the optional argument, distance, is the maximum distance of a leg of the wander (default is 20 pixels). This motion never completes -- the Actor will keep wandering until a new motion type is set.

In file: motion.sc
Inherits from: Motion
Inherited by: none

Properties:

distance
The maximum distance to be wandered on a given leg of the wander.

Follow

This class is for making one Actor follow another. It is invoked for an Actor with (actor setMotion: Follow anotherActor [distance]), where anotherActor is the Actor to be followed and the optional distance specifies how closely to follow. If actor is further from anotherActor than distance, it will start moving towards anotherActor. Otherwise it will stop moving. The default distance is 15 pixels. As for Wander, this motion never completes.

In file: motion.sc
Inherits from: Motion
Inherited by: none

Properties:

who
The ID of the Actor being followed.

distance
The distance to try and maintain from the Actor specified by who.

Chase

This class implements the concept of trying to catch another Actor. It is invoked with (actor setMotion: Chase anotherActor distance [caller]). As in Follow, anotherActor is the Actor to chase. Distance is the distance from anotherActor at which it is considered to be caught. When anotherActor has been caught, caller is cue:ed.

In file: motion.sc
Inherits from: Motion
Inherited by: none

Properties:

who
: The ID of the Actor being chased.

distance
: The distance from who at which who is considered caught.

Jump

This class simulates motion under a gravitational field. It is useful to make an Actor simultate a falling motion. The fall is straight down by default. This simulates, for example, falling off of a cliff. A horizontal gravitation can also be simulated by setting the gx property.

In file: jump.sc
Inherits from: Motion
Inherited by: JumpTo

Properties

x
: x coord of finish. This property is set by the init method to reflect to clients velocity.

y
: y coord of finish. This property is also set internally.

gx
: gravitational acceleration in pixels/(animation cycle)**2. The default is no horizontal gravity, or 0.

gy
: gravitational acceleration in pixels/(animation cycle)**2. This is set according to how intense a gravitation field you wish to have simulated. The default is 3 pixels per (animation cycle)**2, so after 1 cycle the Actor would be moving 3 pixels/animation cycle 6 after 2 cycles etc.

xStep
: horizontal step size. Used by Jump to calculate next position.

yStep
: vertical step size. Used by Jump to calculate next position.

signal
: save area for client's signal bits. Don't set this property.

illegalBits
: save area for illegal bits of client. Don't set this property.

The following two properties, when true, indicate that we are to check for motion completion only after the apogee of the motion. This is necessary if we are to be able to jump onto things. Set these to FALSE if you do NOT want this behavior.

WaitApogeeX TRUE
waitApogeeY TRUE

setTest
: private -- set up the jump completion test.

JumpTo

This class sets up a Jump which will reach a certain x,y position. The kernel call (SetJump) does this in order to use long integer arithmetic. The Motion can the cue: a caller to signal then completion of the jump.

In file: jump.sc
Inherits from: Jump
Inherited by: none

Methods

init

This method is called by the Actors setMotion method.
a JumpTo is initiated by,
(actorName setMotion: JumpTo x y whoCares)
This will make an Actor jump to the coordinates x y, and then cue whoCares.

Orbit

This class sets up an elliptical path for an Actor to follow.
This allows the simulation of an object orbiting around another
object. This mover takes into account the global variable perspective.

In file: orbit.sc
Inherits from: Motion
Inherited by: none

Properties

centerObj
The object (Feature View PicView Prop or Actor) to orbit around.

radius
The radius of the orbit along its major axis (the widest part).

xTilt
The horizontal tilt of ellipse. 0 would be circular and 90 would be edge on.

yTilt
The horizontal tilt of ellipse. 0 would be circular and 90 would be edge on.

angleStep
Angle degree increment per animation cycle.

winding
The direction of the orbit. clockwise=1 counterclockwise=-1

curAngle
Where the Actor will be along the Orbit. 0=north, 90=east etc.

Methods

init
This method is called by the Actors setMotion method.
a JumpTo is initiated by,

```
(anActor setMotion:
            Orbit       ;class Orbit or an instance of it

            theCtrObj   ;some object with x and y properties
            theRadius   ;of the orbit
            theXTilt    ;counterclockwise from x-axis in degrees
            theYTilt    ;same from y-axis
            theStep     ;in degrees, default is 10
            theWinding  ;clockwise=1 counterclockwise=-1
            theAngle    ;0=north, etc.
)
```

Path

This class gives the Actor a Path to follow and allows cue'ing of an object at the intermediate points as well as at the end of the path. It requires less memory than a Script and is also easier to use. The points of the Path are specified in a local array.

In file: path.sc
Inherits from: MoveTo
Inherited by: RelPath

Properties

intermediate
Object to cue at intermediate endpoints.

value
index into path array

Methods

at
Returns the nth element of control array. This method must be redefined to point to the users path array.

next
Move to next point in path. This method is used internally.

atEnd
Returns TRUE if at the end of the path.
This method is used internally.

Example:

To create a square Path (assuming that ego is positioned at 100,100) we do the following:

```
(local
   sPath =  [
            100   60
            160   60
            160   100
            100   100
            PATHEND
            ]
)

(instance squarePath of Path
   (method (at n)
      (return [sPath n])
   )
)
```

The following will make ego walk the path once and cue obj1 at completion of entire path and will cue obj2 at all intermediate pts.

```
(ego posn:100 100, setMotion: squarePath obj1 obj2)
```

RelPath

This class gives the Actor a Path to follow and allows cue'ing of an object at the intermediate points as well as at the end of the path. It requires less memory than a Script and is also easier to use. The relative offsets of the Path are specified in a local array.

In file: path.sc
Inherits from: Path
Inherited by: none

Example:

The following uses a RelPath to make ego do loop-the-loops across the screen:

```
(local
   lPath =  [
            -10   -10
            0     -10
            10    -10
            10    0
            10    10
            0     10
            -10   10
            PATHEND
            ]
)

(instance loopPath of Path
   (method (at n)
      (return [lPath n])
   )
)


(instance rm1 of Room
   (properties
      picture 1
   )

   (method (init)
      (ego
         posn: 60 100,
         init:,
         setMotion: aPath self
         )
      (super init:)
   )

   (method (cue)
      (ego setMotion: aPath self)
   )
)
```

The Avoider Class

The Avoider class is a class which helps Actors get around obstacles. If an Avoider is installed in avoider in an Actor, it gets called instead of the usual motion class when it is time to move the object. The Avoider then calls the motion and checks to see if the Actor was able to move. If the Actor is blocked by something ((client isBlocked:) is TRUE), the Avoider takes over and attempts to get around whatever is blocking the Actor. In some cases it looks really smart, in others astoundingly stupid. As we work on it, it should get better.

In file: motion.sc
Inherits from: Object
Inherited by: none

Properties:

client
The Actor to whom this Avoider is assigned.

heading
The direction (clockwise or counter-clockwise) in which the Avoider turns while trying to get around the obstacle. This is picked randomly when the Actor first encounters the obstacle and remains constant as long as the Avoider is in control. The Avoider attempts to get around the obstacle by turning in 45 degree increments in the chosen direction until it is able to move.

The Event Class

The Event class is the class of user input events (key presses, mouse clicks, etc.)

In file: system.sc
Inherits from: Object
Inherited by: none

Properties:

type
: The kind of input event. The event types are defined in kernel.sh:
  - mouseDown
    : Mouse button was pressed.
  - mouseUp
    : Mouse button was released.
  - keyDown
    : Key was pressed.
  - keyUp
    : Key was released.
  - direction
    : A direction event, meant to move the player.
  - saidEvent
    : User typed a line of input, which has been parsed.
  - menuStart
  - menuHit

message
: The 'value' of the event. For a key, this is the ASCII value corresponding to the key which was pressed.

modifiers
: Bit-mapped property containing any modifier keys which were down when the event occurred. Bits are shiftDown, ctrlDown, and altDown.

x
y
: The coordinates of the mouse when the event occurred.

claimed
: TRUE if some object has already responded to the event, FALSE otherwise. You should set the claimed property of the event to TRUE whenever you respond to an event. In general, you will not want to respond to an event whose claimed property is TRUE (although there may be exceptions), so you'll want to test its value before doing anything based on the event.

Methods:

new:
: If an input event is in the event queue, return it. Otherwise, return 0.

The User Class

A User is an object which corresponds to the person playing the game and acts as the intermediary between the person and the other objects in the game. In the current games there is only one User, and thus we use the class User rather than an instance of the class.

In file: user.sc
Inherits from: Object
Inherited by: none

Properties:

alterEgo
The ID of the Ego which is controlled by the User.

canInput
TRUE if the User should accept input lines from its player, FALSE otherwise. Any time you don't want the player to be able to type an input line, just set canInput to FALSE. Default at startup is FALSE.

canControl
Set to TRUE to let the User control the alterEgo with direction keys, mouse, etc. Anytime you want to set the alterEgo on a pre-programmed motion, set canControl to FALSE to prevent the player from interrupting the motion and taking control. Default at startup is FALSE.

Methods:

doit:
The doit: method for User checks for an event. If one is present, User sends the handleEvent: event message in succession to TheMenuBar (the game's menu bar), alterEgo, and regions (the list of regions, beginning with the current room). If any of these objects respond to the event, they should set the claimed property of the event to TRUE to let the other objects receiving the event know that it has been dealt with.

If, after everyone has had a look at it, the event is still unclaimed, User puts up an input line and waits for the player to type a line of input. If a line is entered and the kernel is able to parse the line, the User turns the event into a saidEvent, indicating that valid input has been entered, and again sends the handleEvent: event message to the above objects. Once it has done so, it disposes of the event and returns.

The Game Class

The Game class implements the game which is being written. The game author creates a source file with script number 0 which contains the instance of the class Game which is the game. This instance is where, for example, input not handled by any Actor, Room, Region, etc. will be handled.

In file: game.sc
Inherits from: Object
Inherited by: none

Properties:

score
The user's current score.

possibleScore
The maximum number of points possible in the game.

speed
The number of timer ticks (1/60th of a second) between animation intervals. Don't set this directly -- use changeSpeed:.

timers
A Set of the currently running Timers. The doit: method of the Game is what runs the timers.

Methods:

play:
Your game is started by the kernel issuing the play: message to the object whose ID is in entry 0 of script.000. The play: method, as defined in the Game class, is basically

```
(self init:)
(while (not quit)
     (self doit:)
     (Wait speed)
)
```

init:
Initializes the game, setting up some global variables and initializing the menu and inventory.

doit:
This method is invoked once per animation cycle and passes the doit: method along to the rest of the game. It is responsible for changing rooms when a newRoom: has been done by the current Room.

newRoom: n
Make room number n the current room. This should not be called directly from user code. This method deletes the cast and the old Room from the heap, then invokes the startRoom: method to load and initialize the new Room.

startRoom: n
Load and initialize room number n. When this method is invoked, the heap has been cleaned up from the previous room. Reimplement this to load any Regions which will span several Rooms and create heap fragmentation problems. Loading such regions below the Rooms lets them stay in the heap without fragmenting it.

changeScore: n
Add the number n (which may be negative) to the user's current game score. If the StatusLine is being displayed, update the score shown there.

handleEvent: event
After passing an event to the MenuBar and the cast, the User class sends the handleEvent: event message to the game, which in turn sends it to the regions. This method may be modified in the game instance to do a (super handleEvent:event) followed by processing to handle unclaimed events by printing an "I don't understand you" message.

setSpeed: n
Set the number of timer ticks (1/60th of a second) between animation cycles to n. This also resets any Timers in the timers Set to account for the change in speed.

restart:
This restarts the game at the beginning, allowing the user to start again without rebooting the game.

save:
Saves the current state of the game to disk, allowing the user to quit the game and restart at the same point later using restore:. Not implemented.

restore:
Restores a saved game state. Not implemented.

showMem:
Displays the amount of memory remaining in heap and hunk space.

The Region Class

A Region is an area of a game which is larger than a Room (see below) -- for example a forest, island, or a planet. It is used to provide standard behavior (i.e. responses to user input) in an area of the game without having to code the behavior into each room in the region. For example in Space Quest, we might define the town of Ulence Flats and the planet of Kerona as regions. The Ulence Flats region would be used to handle the response to 'look city' while in the town, whereas the Kerona region would handle the response to 'look sky'. Regions take the place of the 'dynamic logics' of the AGI interpreter. A Region differs from a Locale (see below) in that a Region has a doit method that will be invoked every animation cycle by the game. This is where to put any checks or code that encompasses more than one room and needs to be done each animation cycle.

In file: game.sc
Inherits from: Object
Inherited by: Room

Properties:

script
The ID of a Script for the current room.

number
The region number. Set by the setRegions: method discussed in class Room below, it is used when we dispose the region.

timer
The ID of a Timer set to cue: this Region.

keep
Set this to TRUE if you want the Region to remain in the heap through a newRoom:, to FALSE if you want the Region unloaded.

initialized
This property is set to TRUE when the Region is sent the init: message after a setRegion: involving it has been done. The init: method is not invoked once initialized is TRUE, so a Region will only be initialized once for the Rooms which require it.

Methods:

init:
When a Region is added to the region list regions by the setRegions: method discussed in the class Room below, it is sent the init: message in case any setup is needed. The default init: method does nothing.

doit:
Each active Region is sent the doit: message in each animation cycle. The default method passes the doit: message along to the script, if there is one, but does nothing else.

handleEvent: event
The default method for this sends handleEvent: event to the Region's script, if there is one. You may handle events either in the Region itself or its script. Remember that in general you will not want to handle an event which is claimed, and that you should set the claimed property of the event to TRUE if you

```
     respond to it.

setScript: script
     Set the script of the Region to script and init: it.

cue: newState
     Cue this Region's script.  If the optional newState is present,
     invoke (script changeState:newState) instead.
```

The Locale Class

A Locale is similar to a Region in that it may encompass many Rooms, but its only purpose is to provide default responses to user input. Thus, a forest locale will provide generic responses to input like 'look forest', 'look tree', 'climb tree', etc. A locale is attached to a Room with the setLocales: method.

The Room Class

The Room class is the most specific Region. It is the point at which a picture, or 'scene' is defined for the player to walk around in.

In file: game.sc
Inherits from: Region
Inherited by: none

Properties:

picture
: The number of the picture corresponding to the room. Usually, this will just be the room number.

style
: If style is specified, it overrides the global showStyle when picture is drawn during the Room's init:. Any other picture drawn once the Room is running must have its show style explicitly specified.

horizon
: The y coordinate which is considered to be the room's horizon. Actors which have not had an ignoreHorizon: done on them cannot be positioned above this coordinate. Also, an Ego which hits this coordinate will have its edgeHit property set to northEdge.

blocks
: A Set of software blocks (Blocks), rectangles which Actors are not allowed to enter.

north
east
south
west
: The number of the room to which you wish to automatically change when ego hits the specified edge of the room. If you wish to control the room change yourself when ego hits a certain edge, do not specify the room for that edge in the Room instance and check (ego edgeHit?) yourself. The edgeHit property will have the value northEdge, southEdge, eastEdge, or westEdge if ego hit one of the edges.

controls
: A set of dialog items (DItems) which are push-buttons for this

picAngle
: This property is the angle above level that you are viewing the picture. It is used to set the global perspective upon room entry.

vanishingX
vanishingY
: These are set according to where the vanishing point is in the picture to be shown. They determine the angle that the arrow keys will move ego in a room. The Up key will always move ego towards the vanishing point and the down arrow away. The default give a straight up and down in response to these keys.

Methods:

init:
The default init: for a Room is to create an empty Set of Blocks, draw the picture specified by picture, and set ego's position based on its position in the previous room. You will want to supplement this with the following:

Load all views, sounds, pictures, etc. which will be required by the room so as to get all disk access done at Room initialization time.
Initialize all Actors and Actors which are to be in the Room initially.
Set a script for the Room.
Add any required Blocks to the Room.

doit:
Send the doit: message to this Room's script, if it has one, then check to see if ego has hit the edge of the Room. If ego has hit an edge and there is a room number specified for the edge, invoke the newRoom: method to change to that room.

dispose:
This is called by the game when changing from this Room to a new Room. The game handles disposing of the current cast, then does a (curRoom dispose:). The default dispose: simply disposes of the Room's script, blocks, and controls. You will want to supplement this to dispose: of anything else which you have created which is specific to the Room, and any modules which the Room has loaded.

setRegions: region [region ...]
Sets the Regions which contain this room. This should be invoked in the init: method of the room. The parameters region are the script numbers of the regions. They should be listed from most specific to most general, since the order in which they are listed is the order in which they will be invoked. Thus, for a room in Ulence Flats in Space Quest, we might write

```
(self setRegions: ULENCE_FLATS KERONA)
```

to say that we are in the town of Ulence Flats on the planet Kerona.

The global variable regions contains the ID of a List of regions which are sent the doit: and handleEvent: messages in order when appropriate. The newRoom: method of Game puts the current room at the head of this list, and setRegions: adds Regions to the end of the list in the order in which they are listed in the parameter list. If the Region was used in the previous Room, it will be neither reloaded nor reinitialized. At the end of the newRoom: method, all Regions in the list which do not refer to the current Room are deleted.

handleEvent: event
The default Room method passes the event along to the Room script (if there is one) and then to any controls in the room.

newRoom: n
This method is the one to use when changing Rooms. It is invoked when the Room automatically changes to a new Room when ego hits the edge. Reimplement this method if you need to do some processing (such as dispose:ing a Region) before changing rooms.

overlay: picture
This method overlay a picture over the current picture.

The Timer Class

The Timer class implements the concept of an alarm clock -- you can set it to cue: an object at some future time.

In file: system.sc
Inherits from: Object
Inherited by: none

Properties:

cycleCnt
The interval to be timed measured in animation cycles. This is set by the set: and setCycle: methods, and should not be manipulated directly.

seconds
The interval to be timed measured in seconds. This is set by the setReal: method and should not be manipulated directly.

lastTime
Used internally when timing seconds.

client
The object to cue: when the timer goes off.

Methods:

The InvItem Class

InvItems are the items which ego can gather and use as he moves through the game. As described in the description of the Inventory class, they are generally referred to by their position in the Inventory list rather than by their object ID (which is not known in all modules).

In file: invent.sc
Inherits from: RootObj
Inherited by: none

Properties:

name
: The string which is the InvItem's name. This is what will be shown in the inventory window if you do an (inventory showSelf:ego) and ego has the item.

said
: A said-string describing the way in which the User may refer to the item.

description
: The description of the object to be displayed if the User clicks the mouse on the item's name in the inventory window.

view
loop
cel
: The view, loop, and cel to be displayed for the item if the User clicks the mouse on the item's name in the inventory window.

owner
: The current 'owner' of the item. This is either a room number (the item is in the room) or an object ID of an Ego (the Ego possesses the item).

script
: The object ID of a Script for the item. This can be used to keep track of the state of a changeable item and to change the view, loop, cel, and description of the item. Thus, an electronic device might have the following script attached to it:

```
(instance deviceScript of Script
     (method (changeState newState)
          (switch (= state newState)
               (deviceOn
                    (client
                         description:"The device is on."
                         cel:0
                    )
               )
               (deviceOff
                    (client
                         description:"The device is off."
                         cel:1
                    )
               )
          )
     )
)
```

Thus, in response to user input of 'turn device on', you can write

```
((inventory at:iDevice) changeState:deviceOn)
```

Methods:

saidMe:
Return TRUE if the user input referred to this item, FALSE otherwise.

ownedBy: whom
Return TRUE if the item is owned by whom (either a room number or an object ID), FALSE otherwise.

moveTo: whom
Set the item's owner to whom (either a room number of object ID).

showSelf:
Display the item's view, loop, cel, and description in a window.

changeState: newState
Send the changeState:newState message to the item's script.

The Block Class

The Block class implements ability to keep Actors out of certain areas on the screen without controls being added to the picture. In particular they are the only way of preventing an Actor who is allowed to have his baseRect off the bottom of the screen from walking into something, since the Actor's baseRect will not encounter any blocking controls.

In file: actor.sc
Inherits from: Object
Inherited by: none

Properties:

active
: Set to TRUE if the Block is active, FALSE otherwise. Actors can move into inactive Blocks.

top
bottom
left
right
: The bounding coordinates of the Block rectangle.

Methods:

init:
: Add the Block to the set of blocks for the current room and enable the Block.

doit: actor
: Return TRUE if actor is outside of the Block (and is thus in a legal position) or FALSE if it is inside the Block (and thus must be moved out). This is called in each Actor's canBeHere: method.

dispose:
: Delete the Block from the set of blocks for the current room and dispose of it if it is a dynamic instance.

enable:
: Set the active property of the Block to TRUE, so that Actors cannot enter it.

disable:
: Set the active property of the Block to FALSE, so that Actors can enter it.

The Cage Class

The Cage class implements ability to keep Actors in a rectangular region.

In file: actor.sc
Inherits from: Block
Inherited by: none

top
bottom
left
right
The bounding coordinates of the enclosing Cage.

Methods:

init:
Add the Cage to the set of blocks for the current room and enable the Cage.

doit: actor
Return TRUE if actor is inside of the Block (and is thus in a legal position) or FALSE if it is outside the Cage (and thus must be moved back in). This is called in each Actor's canBeHere: method.

dispose:
Delete the Cage from the set of blocks for the current room and dispose of it if it is a dynamic instance.

enable:
Set the active property of the Cage to TRUE, so that Actors cannot leave it.

disable:
Set the active property of the Cage to FALSE, so that Actors can leave it.

The Sound Class

This is the class which allows sound to be played.

In file: game.sc
Inherits from: Object
Inherited by: none

Properties:

done
: Set to TRUE after the sound has finished playing, FALSE while it is playing.

number
: The number of the sound.

priority
: The priority of the sound if multiple sounds are playing. Most sound effects should have the default priority of 0. More important sounds (say an error beep) should have positive priorities -- the more important the higher the priority. Less important sounds (background music) should have negative priorities. This will allow sound effects to override background music (which will resume at the appropriate point) and error beeps to override sound effects.

signal
: This is a bit-mapped property used to set characteristics of the sound. Its bits are defined in base.sh. Only one bit, blockingSnd, is currently defined. When it is set, the sound blocks when it is playing -- all game play stops until the sound is complete. Sampled sound effects are blocking sounds by default, and needn't have this bit set.

Methods:

init:
: Initialize the sound and set any appropriate bits in signal to indicate the sound's nature.

play:
: Start the sound playing and add it to sounds, the list of sounds which are playing.

dispose:
: Stop the sound.

The StatusLine Class

The StatusLine class provides a status line at the top of the screen which is programmer-definable. When enabled, it overlays the menu bar. The user may still access the menu by pressing Esc or positioning the mouse pointer in the status line end pressing the mouse button. The status line usually shows the player's score. To use a status line in a game, create an instance of class Code which has the interface described in code below and set the code property of StatusLine to it. To display the status line, execute (StatusLine enable:).

In file: game.sc
Inherits from: Object
Inherited by: none

Properties:

state
Whether or not the status line is being displayed. Do not manipulate this directly.

code
An instance of Code which formats the status line. It will be called from StatusLine's doit: method with a pointer to string storage as a parameter:

```
(code doit: @theLine)
```

The code should format the status line into the provided string.

Methods:

doit:
Format the status line and display it if it has been turned on.

enable:
Turns the status line on. Invokes the doit: method.

disable:
Turns the status line off. Invokes the doit: method.

The File Class

The File class allows you to open and write to a file on disk. This is useful for logging user input for which you have no response in the development or beta-test phase, writing utilities which allow you to position Actors on a picture and then write out the coordinates, etc.

In file: file.sc
Inherits from: Object
Inherited by: none

Properties:

name
The name of the file. Defaults to "gamefile.sh".

handle
The handle by which the operating system refers to the opened file. Don't monkey with this, or you may blow away the OS.

Methods:

open: [flag]
Open the file whose name is name for writing. The optional parameter flag may be either fAppend, in which case writing begins at the end of an existing file, or fTrunc, in which case the contents of the file are deleted before writing to it. If flag is not present, fAppend is assumed.

write: str [str ...]
Write the strings pointed to by str to the file.

close:
Close the file.

The Code Class

The class Code encapsulates the concept of a subroutine into an object which can be executed or passed to an object for execution. This class is also used for making loopers and viewers.

In file: system.sc
Inherits from: Object
Inherited by: none

Properties:

Methods:

doit: [args...]
The doit: method of a Code object is the code to be executed. Any number of arguments is allowed.

Global Variables

Global variables 0 through 49 are reserved for use by the system classes. Game-specific global variables start at 50. The following global variables are defined by the system. For up to date global list see s:system.sh.

ego
: The ID of a static instance of class Ego defined in game.sc. This is the protagonist of the game.

curRoom
: The ID of the current Room.

userFont
: The number of the font to be used in Print statements, etc. Default is 1. Set it to the font you wish to use in the init: method of your Game, or change it at will in the game.

cast
: The ID of a Set of Actors and Egos which constitutes the characters on the screen.

quit
: The main loop of the game is

```
(while (not quit)
     (theGame doit:)
     (Wait speed)
)
```

: so setting quit to TRUE breaks out of the main loop and terminates the game.

addToPics
: A Set of PicViews which have been added to the current picture.

debugOn
: A generic debugging flag. I usually have a Debug menu item to set it, and trigger any debug display that I want off of it, rather than creating a special trigger whenever I want to debug something.

sounds
: The Set of Sounds currently playing.

inventory
: The ID of the Inventory class or instance which is the Set of all InvItems (inventory items) in the Game. Inventory related issues are not well defined yet...

theGame
: The ID of the Game instance.

regions
: The Set of Regions currently in effect.

curRoomNum
: The number of the current Room.

prevRoomNum
: The number of the previous Room. (So you know how you got where you are.)

newRoomNum

Used by a Room to signal to the Game that it should change to a new Room.

showStyle
: The global style for the transition from one picture to another. This may be overridden by the style property of a given room. See the DrawPic kernel function for the possible styles.

aniInterval
: The number of ticks it took to do the last animation cycle. This is used to test animation speed on a particular machine.

speed
: The number of ticks between animations. This is set, usually as a menu option, to determine the speed of animation. The default is 6.

timers
: The List of timers in the game.

score
: The player's current score.

possibleScore
: The games highest possible score.

theCursor
: The number of the current cursor.

normalCursor
: Number of normal cursor form.

waitCursor
: Cursor number of "wait" cursor.

smallFont
: Small font for save/restore, etc.

lastEvent
     The last event (used by save/restore game).

modelessDialog
     The modeless Dialog known to User and Intrface.

bigFont
     The number for a large font.

volume
     Sound volume on supported machine.

version
     The SCI version string.

locales
     Set of current locales.

curSaveDir
     Address of current save drive/directory string.

aniThreshold

perspective
     The Player's viewing angle. This is set according
     to the Rooms picAngle property. Degrees away from
     vertical along y axis.

features
     Locations that may respond to events.

sortedFeatures
     The features and cast sorted by "visibility" to ego.

useSortedFeatures
     If TRUE enable cast & feature sorting.

egoBlindSpot
     Used by sortCopy for sortedFeatures to exclude actors
     behind ego within angle from straight behind. Default
     zero is no blind spot.

overlays
     The overlay List.

doMotionCue
     A motion cue has occured - process it. The Games doit method
     gives the motionCue: message to each element of the cast if this
     is set to TRUE.

systemWindow
     ID of standard system window. If you wish to supply your own window
     for system messages set this to the ID of your window.

lastSysGlobal
     Defines the end of Global variable space.

Index