GUI Extension to Project 4

Tony H. Kim

April 26, 2007

Disclaimer and outstanding issues:

1. The application is to be loaded by opening "driver.scm" and by running the following command in the interactions window: "(setup 'name)".
2. When first loading the program, DrScheme will take considerable time (the Allegro library) simply downloading the necessary files from the PLaneT online repository. Please be patient. *This also means that the program can be ran only on computers where you have write-permissions to the default DrScheme download directory!*
3. When first running the program (after loading DrScheme) the colors seem to be off. It appears that if you re-load the program by calling 'Run' a second time and then calling "(setup 'name)" the color issue is fixed.
4. Sometimes (i.e. on my laptop) the game simply will not load, complaining of some bizarre issues. If this occurs, please try modifying the 'width' and 'height' parameters to smaller, common values, (i.e. 640x480 or 800x600) and lowering the requested 'color-depth'.
5. As I remarked during tutorial, the internal code is quite ugly – in particular because I have made several severe revisions to the program without taking the time of rewriting from scratch. Hence, there will be inconsistencies. For instance, in order to obtain the image from a graphics data associated with an object, I initially used "(ask (ask object 'SCREEN-DATA) 'IMAGE)." At a later point in the development, this became: (ask-sd object 'IMAGE) given the frequency of the above statement. However, although in later edits, I used the latter statement, I did not take the time to extensively hunt and replace the original statement. Furthermore, some variables, which were initially defined globally, were moved into different objects (example, to refer to the width of the screen I initially referred to the global variable 'width', but later retrieved the same information through the screen object); and the object system itself has undergone various changes. With all of these changes, without a single rewrite, the code has become disastrous. On the other hand, I will try to walk you through the rough organization of the program in this document, and to point out the pretty sights.

1 Introduction

The interface of the original Project 4 is extremely unintuitive. By forcing the user to interact through the underlying object system itself, we necessitate the use of distracting commands such as "thing-named" and cause the application to balk at the slightest typos in input. While to a certain extent this can be eliminated

by shortcut bindings, (i.e. (west) for (ask me 'GO 'west)) I found playing the original game to be quite a chore, simply because the input mechanism was so unwieldy.

Hence, I have created a graphical interface for the adventure game. The intent is to allow the user to interact via *only* the mouse, so that there would be no need for monstrosities such as "(ask me 'THROW (thing-named 'tons-of-code) (thing-named 'eric-grimson))". In this extension, I feel that I have achieved the beginnings of this endeavor, since movement and few other basic functions are achieved through the mouse. Below is the feature list of the project that I am submitting:

1.1 Feature list

1. Ability to navigate between rooms (north, south, east, west) by moving to the edge of each room.
2. The ability to pick up objects by moving over them.
3. A graphical representation of the avatar's inventory at the bottom of the screen.
4. The ability to select an item in the inventory by a left-click.
5. The ability to use a selected item as a weapon (by throwing) with a right-click on the main screen.
6. Enemies that vary in speed and in the manner in which they chase their target. (There are "normal" enemies that always move directly toward their target, and an "inertial" enemy that *accelerate* towards its target rather than moving directly at them – resulting in (damped) oscillations.
7. NPCs (that don't do anything yet). If they are provoked however, they become hostile.

While the above feature set is very limited (certainly, I have not implemented objects with more complicated syntax such as PDAs and hacks), I feel that the extension is still an improvement on the original because the interaction with the application is much more natural.

1.2 Playing the game

The user in the game is represented by a white rabbit, MIT class of 2009. Using the mouse, the user left-clicks on a location on the screen, causing the rabbit to move towards that location. This is the main mode of interaction with the application.

In order to navigate between rooms, one clicks the mouse beyond the gray rectangle at the edge of the screen, which causes the rabbit to run towards that point as usual. When the avatar is beyond the bounds, the player will move in the corresponding direction. If the exit does not exist, the player will get an appropriate message in the upper-left dialog.

In certain rooms, there will be computer-controlled characters. They come in two variants: hostile and nonhostile. Hostile NPCs will move towards its target, and cause its target to lose health when overlaps on the screen occur. Initially nonhostile characters can *become* hostile if it is provoked.

The player's only means of battling the hostile NPCs is to pick up objects and to throw them. An object can be picked up by moving over it: it will then enter the person's inventory, as visualized on the bottom bar. Among the obtained items, the one with the orange background represents the "currently selected" item, which is the one thrown when the player right-clicks on a point in the main screen.

Lastly, the health of the player is represented by the bar on the lower-left corner. When the player's health drops below zero, the player is transported to "heaven," as in the original Project 4.

2 Organization of the code

The central change from the original is the reconstruction of the 'SCREEN object which now takes care of the graphical loop, rather than simply emitting messages on the screen. It resides within "driver.scm."

Additional parts of the program are split into the following files. (In the order included by driver.scm.)

- objsys.scm
- objtype.scm
- vectors.scm
- resources.scm
- screendata.scm
- drawroutines.scm
- world.scm

Of all the files, objsys.scm has undergone the least amount of revision. It contains the implementation of the 6.001 object system. On the other hand, the object types (objtype.scm) for Project 4 has undergone major changes. (Or, perhaps more accurately, a major reduction.) For example, the subclasses beginning with the 'PERSON class have been completely rewritten, primarily to remove the methods that are not relevant in the limited feature set that I was aiming for. More importantly, objects such as 'THING, 'MOBILE-THING, 'AVATAR and 'NPC now posess a new object which encapsulates the graphical information associated with it. These new graphical-information objects will be discussed in the section "screendata.scm" below. But let's just cdr down the above list in order.

2.1 vectors.scm

Because I am dealing with entities to be displayed on a 2D screen, I have found it convenient to write a vector class, so that I can manipulate positions easily, and without having to do cumbersome computations component-wise. Hence, vectors.scm represents a collection of standard operations on vectors, i.e. normalization, finding angles, etc. Furthermore, vectors.scm contains other generic helper functions such as a number-to-integer caster ('my-round') and a counter to ensure proper timing in the game. (Section 3.1.1)

2.2 resources.scm

This file contains mainly two things. The first is a simple construct that makes it convenient for me to refer to colors. If I want to ask the Allegro drawing primitives to draw some object in white, I have to pass (image-color 255 255 255) as an argument. However, the 'colors' object allows me to refer to the same quantity by (colors 'white).

This file also contains the main structure that permits animation in my program. This, in my opinion, is the most elegant piece of the entire project (although that is not saying much). As you already know, the idea behind an animation is that we quickly swap images that vary slightly between frames. I have produced the 'make-animation' procedure which facilitates this function. It is to be used in the following manner:

Suppose the files tony0.png, tony1.png, tony2.png, ..., tony9.png represent ten frames of an animation sequence that is to be displayed in rapid succession. Then, to create the animation object, we call:

'(make-animation 10 n "tony")'

where the 10 refers to the number of frames to be incorporated, and the second parameter 'n' is some positive integer that allows for delays between frames. For instance, if the game loop is running at sixty frames per second, we may want the animation to change the frame only every 5 game-loops (n=5), otherwise the animation may be too rapid. Lastly, the "tony" refers to the filename associated with the sequence of images in the default image directory.

Suppose that the return value of the above expression is now saved in a local variable 'tony-anim' as in the following:

'(define tony-anim (make-animation 10 1 "tony"))'

tony-anim is now a *procedure* that returns, each time the procedure is called, the next image-object that is to be drawn on the screen. So, for instance:

```
(tony-anim) => #<image:tony0.png>
(tony-anim) => #<image:tony0.png>
(tony-anim) => #<image:tony1.png>
(tony-anim) => #<image:tony1.png>
(tony-anim) => #<image:tony2.png>
etc...
```

This structure allows me elsewhere to treat the animation – a sequence of images – as a single image, since the object takes care of the updates internally. As a matter of fact, even for non-animated images, i.e. a single shot, we can use the same construct. We do this by simply giving it a first argument of 1 (for 1 image, rather than 10 as in the case above. In this case the frame-delay argument has no effect).

Furthermore, the file contains the definition of the 'animation-collection.' This is a container structure that allows me to associate many different, but related, animations. For example, the avatar on the screen can move in various directions, which has associated with it distinct animations. The animation-collection is implemented as an association list with pairs: an animation and a corresponding keyword (i.e. 'idle-left'). The more complicated objects on the screen, such as the avatar and the NPCs, make use of this animation collection to retrieve the appropriate image to be printed. The usage of the animation-collection is evident in the following code:

```
(define (rabid-animations)
  (let ((frame-delay 3))
    (make-collection 'idle-left  (make-animation 42 frame-delay "rabid/idle")
                     'idle-right (make-animation 42 frame-delay "rabid/idle-right")
                     'north      (make-animation  8 frame-delay "rabid/north")
                     'east       (make-animation  8 frame-delay "rabid/east")
                     'west       (make-animation  8 frame-delay "rabid/west")
                     'northeast  (make-animation  8 frame-delay "rabid/northeast")
                     'northwest  (make-animation  8 frame-delay "rabid/northwest")
                     'southeast  (make-animation  8 frame-delay "rabid/southeast")
                     'southwest  (make-animation  8 frame-delay "rabid/southwest")
```

```
    )
  )
)
```

2.3 screendata.scm

This file contains a *new* class hierarchy, which is distinct from the NAMED-THING, THING, MOBILE-THING,... class system that we developed in the original project. However, it is not entirely separate, since there is a correspondence between the two class families. The screendata-classes are containers for the graphical information associated with the original objects. Architecturally, the classes in the original system *has* its corresponding screendata-object, as an internal variable.

Initially I have considered merging the two class libraries, but the nature of the application has kept them separate. If the program were entirely a real-time scroller-game, then it would make much more sense to place the screen information such as the x- and y-coordinates within the object itself. However, my program represents a mix between a real-time game and the turn-based nature of project 4. While the entities *in the same room as the avatar* move and react in real time, entities in other rooms are treated in a turn-based manner as in the original.. Because of this duality, I have found it more appropriate to divide the graphical information and the turn-based game information of an object.

The hierarchy of the screendata-classes go as follows:

- SD-MESSAGE
- SD-BACKGROUND
- SD-BASIC
  - SD-MOVABLE-THING
    - SD-AVATAR
    - SD-NPC
      - SD-NPC-INERTIA

The SD-MESSAGE object is peculiar among this set in that they do not have a corresponding member in the original class system. Rather, they are simply encapsulations of the messages to be displayed on the upper-left of the game screen. Associated with SD-MESSAGE objects are a string, the font-color, and the display duration.

The SD-BACKGROUND object is associated with the PLACE object of the original; and indicates how the background of that room is to be displayed. Two currently implemented drawing styles are: flat-color background and tiling of an image.

SD-BASIC object represents the most basic unit of the actual icons displayed on the screen. It contains position information and the means to change its position. In the game world provided, the trees in the great court are represented by SD-BASIC.

The SD-MOVABLE-THING is an extension on the SD-BASIC object. It has additional information, such as 'target' and 'velocity' that permits motion. For instance, when the user clicks on a location on the screen, the (x,y) coordinates of the click is sent as the target, and the SD-AVATAR's 'MOVE method will utilize

the 'TARGET and 'VELOCITY features of SD-MOVABLE-THING to cause the character to move towards the location.

SD-AVATAR is a subclass of SD-MOVABLE-THING. The main distinction is that its 'IMAGE method – which returns the next image to be printed to the screen – is set to deal with an animation-collection rather than a single animation.

SD-NPC corresponds to the NPC class in the original. It is similar to SD-AVATAR, except that it deals with a fewer set of animations in its animation collection.

SD-NPC-INERTIA overrides the 'MOVE method of SD-MOVABLE-THING such that the character progresses towards its target via accelerating towards it, rather than heading directly at it.

2.4 drawroutines.scm

This file is divided into two sections. At the lower level are a few wrappers for the Allegro library bindings, so that I can call drawing routines with the convenience of vectors, rather than explicitly pulling out components as required by the Allegro functions. The file also contains many higher-level drawing routines such as 'draw-messages', 'draw-sidebar', 'draw-healthbar', etc. that are called from the main drawing loop. (Section 3.2)

2.5 world.scm

This file contains the code necessary to load graphical resources specific to the world that I have created. The syntax of the resource creation is not difficult and can easily be inferred from the example code. Furthermore, despite the addition of the GUI, the creation of the world in the 'create-world' procedure is very similar to that in the original.

Finally, this file also contains the 'setup' procedure that initializes and runs the game.

3 The 'SCREEN object

As mentioned previously, the screen object within driver.scm is the focal point of my extension. Two main points of interest are the game cycle logic ('cycle-logic') and the drawing routine ('on-paint').

3.1 Cycle-logic

3.1.1 Timing issues

An interesting feature of the cycle-logic is how I achieve control of the timing. In particular, the operation of the game is active, rather than passive, in the sense that at each cycle, I explicitly carry out a number of operations to *check* whether inputs are being made, rather than *waiting* for events perhaps via a message queue. This scheme is computationally inefficient, and has an additional disadvantage.

Because the cycle-logic function is called in rapid succession, typically over 40 times a second, the loop is able to deluge the underlying game system with needlessly repeated commands. For instance, suppose that we would like to go south from a room that does not have an exit in that direction. The desirable outcome is that if such a request is made, the application will inform the user that there is no exit in that particular direction through the display system. But because it is the game loop that actively checks whether the current position of the avatar is beyond the bounds, if we were naively to do the processing in the 'cycle-logic' procedure, we will make over 40 requests per second to change rooms, which would result in over 40 messages from the program that there is no exit in that particular direction. This is definitely undesirable.

In order to overcome this difficulty, I have implemented an internal timer, (identical to the ones we have seen when first discussing objects with state) and limited operations to occur in certain multiples of that timer. By using the framerate of the game as a measure of time, we can actually limit certain operations to a fixed number of times *per second.* For example, as implemented, the request to change rooms is made only three times per second, as is the request for the avatar to pick up an object in its vicinity.

However, a closer examination will show that I actually employ two separate "timer" devices. I have already mentioned the internal-clock which is used to limit internal processes. We need a second mechanism for timing, in order to *separately* control the rate of inputs *from the user.* One instance in which we would like to limit the rate of input from the user is in the playback of sound: whenever the user clicks with the left-button, the game plays a sound file, "left.wav." However, suppose that the game proceeds at sixty frames per second and that the click lasts half a second. This means that the soundfile will be played 30 times, each starting 1/60th of a second after the other. This results in a nasty cacophony. So we require a way to limit user input, but this cannot be achieved by the internal clock. This is so because *we would like the program to respond promptly to user input* even while limiting the rate.

Suppose we were to cap user input using an internal timer, limiting it to two processings per second. *As implemented, this means that the program will poll for user input only when its internal clock is a multiple of 30 (assuming a framerate of 60 fps).* Clearly, we do not want the user input to be limited by the internal mechanism; in other words, the game will simply feel unresponsive if the user's commands are processed only if it happens to occur when the internal clock is a multiple of 30. So, instead, I employ a second mechanism: it consists of a counter ('poll-timing') that counts down towards zero each frame, at which point user permit is allowed. If there is no user input, then the counter *remains* at zero. The counter is set to a pre-set value (''poll-rate') whenever a user input is made.

3.1.2 Operations

At every frame, the following processes are called from 'cycle-logic':

- Process user input (controlled by the poll-timing scheme)
- Check for moving between rooms (three times per second)
- Check for picking up an item (three times per second)
- Move the avatar (every frame)
- Autoequip an item in the inventory (every frame)
- Ask hostile enemies in the room to move towards and hurt their targets (every frame)
- Check for collisions with thrown items (every frame)
- Check whether a request to quit has been made (every frame)

3.2 On-paint

The following paint procedure is called every frame by the main loop. It draws onto the screen by using methods in drawroutines.scm. If we draw more than two things on the same screen space, the most recently made request will be printed over the previous one. I take advantage of this; the calls are made in the following order:

- Draw the room (i.e. background, gray border, etc...)
- Draw the sidebar (i.e. health bar, inventory)
- Draw the 'THINGS in the room. (We can achieve proper depth, by sorting the objects to be drawn by their y-coordinate!)
- Draw messages to the screen.
- Draw the cursor.

4 Closing remarks

I hope that I have given a somewhat decent overview so that you don't become bewildered by trying to decipher a program which is a strong contender for one of my personal worsts. While the end product I am submitting is poorly written, I've become intrigued with the idea of a top-down scroller game while working on this project. (Reminds me of my Diablo days!) So, I will probably toy around with this further in the future.

Thanks for taking the time to review this!