Programming a video game

1. The nature of game programs
2. Terminal Escape Sequences
3. Using the termios module
4. Dealing with the board and screen.
5. The game loop
6. Adding keyboard control
7. Adding more players.
8. Keeping score. Burning holes.
9. The final version for now
10. Some ideas for further exploration.

This study concerns the development of a little video game. It's modeled on a game my family enjoyed in the 1980's called Lode Runner. We would sometimes play it by the hour, having a good time and wearing out keyboards. But Lode Runner, at least the version we had, assumed the cpu ran at a certain speed and once we had processors faster than about 8Mhz, the game was too fast to be played.

Lode runner consists of a vertical "board" containing ladders and "catwalks" between the ladders. In the original game there were about 100 different boards or levels, each one just a bit harder than the last. "You" were represented by a little person going up and down the ladders and running across the catwalks gathering the "lodes" of treasure. You used the arrow keys to indicate a change of direction. While doing this you were pursued by what I always thought of as robots. You won a level by retrieving all the lodes and getting to the top of the board before being tagged by a robot. You had one weapon at your disposal. You could burn a hole in a catwalk that a robot would fall into and get stuck. The robot would take a couple of seconds to get back out. And in a few seconds more the catwalk would self repair. If you fell into the hole you couldn't get out and with the repair tragedy was the result.

We are going to develop a game quite similar, using simplified graphics and keyboard input. The code only works for Linux and Python 2.0 or newer. This is because of the ansii escape sequences we'll use to display the game on the screen and the special termio module for monitoring the keyboard.

The nature of game programs

Game programs, actually games in general whether Lode Runner or Chess, have a certain structure. A game is a set of moves. There must be a way to visualize the game in progress (screen), a way to get input from the player(s), rules that determine what moves are possible and to determine when the game is over.

At the center of any game program is the "game loop" that repeatably performs the above steps until the game is terminated, that is, won, lost or otherwise interrupted.

Moves in chess require each player to examine the board, decide on the best next move and take it. Moves in Lode Runner are a little more dynamic. Each player (you and the robots) must be advanced one position and each player must have the opportunity to change direction. A certain time should elapse between moves so that you, the human, can keep up. Finally, it needs to be determined when the game is over. This will happen if a robot tags you or, as we'll see, if you fall off the board. You will have won if you gathered all the lodes before this happens.

We will build the program in stages so we can concentrate one aspect at a time. We'll start with some unusual I/O to the terminal.

Terminal Escape Sequences

Each character in the Ascii character set has a numeric value. You can use the "ord" function to find out what that value is.

```
>>> ord('A')
65
```

The reverse of the "ord" function is "chr". It will turn a numeric value from 0 to 255 into the corresponding character.

```
>>> chr(65)
'A'
```

Some characters do not display and in order to represent them in a Python string they must be quoted with the "\" and an octal (base 8) number. The escape key (ESC) has a numeric value of 27 decimal. In a Python string it looks like

```
>>> chr(27)
'\033'
```

You may be used to using ESC in Vim editor to slip out of insert mode. It is also the basis for escape sequences that do special things. Try this.

```
>>> print "\033[1;1H\033[JHello World"
```

Your screen or window should have erased and then printed "Hello World" in the top left corner, and leaving a new ">>>" prompt on the second line.

There are two escape sequences in the above string. "\033[1;1H" positions the cursor to row one and column one. The "1;1" may be replaced with other numbers. "\033[J" will erase the screen from the current cursor position.

Escape sequences are also used from the keyboard for function keys and the four arrow keys. The up arrow generates a string "\033[A", the down key "\033[B", the right key "\033[C" and the left key "\033[D". However you need to be in a special terminal mode to see the character sequence faithfully reproduced.

Using the termios module

We have two problems with the keyboard in a arcade-like game. One is to input the arrow keys so that the program can change your direction on the fly. The second is a little more subtle. While waiting for you to type a key the game still needs to keep going. If you use input functions like `input` or `raw_input` then you know that the program stops until you input a string followed by the return key. That won't work here.

The solution, which is specific to Unix implementations, consists of 3 parts. The first is to put the keyboard input into a mode called "non-canonical". Canonical mode means that you must input an entire line before the program sees any of it. That allows you to edit the line as you are typing without your program having to worry about seeing backspace characters followed by corrections. In non-canonical input your program gets characters just as soon as they are typed.

The second part of the solution is to input keystrokes without the system trying to "echo" them onto the screen. An up arrow will often show on the screen like "^[[A" which we don't want to see. Our program just wants to deal with the fact that you pressed the up arrow.

The final part of the solution lets the program keep going whether or not you press a key on each move. And at 10 moves a second this is of course a must. The secret is to use the timeout function, but set the timeout to zero. This will allow us to read what is in the look ahead buffer, that is anything typed since the last move 1/10th second earlier.

These ideas are encapsulated in the module ttyLinux.py which you should now examine.

There are 3 functions for the keyboard input. The function "setSpecial" puts the keyboard into our non-canonical, non-echo, lookahead mode. But first it saves the prior settings so that the function "setNormal" can restore them. I don't want to go into detail on the "termios" module but, if you are interested, "man termios" will give you lots of information. Suffice it to say that setSpecial turns individual binary bits off in a flag word and sets a byte value for the timeout function. The function "readLookAhead" simply reads a maximum of three characters, enough to hold an arrow key escape sequence.

You should check at this time whether termios is available in your Python.

```
>>> import termios
>>>
```

If you don't get any error message then everything is ok. Otherwise you will have to rebuild and reinstall Python to include it. Briefly, you must edit Modules/Setup and uncomment the line containing "termios". Then do a "make" from the top directory followed by a "make install". If this means nothing to you, check with your system administrator.

Let's look at a little program to use ttyLinux.py. The program key1.py will set the terminal to our special mode, retrieve look ahead for 10 one second intervals and then restore the terminal to normal. Here is a sample run. After the import I typed the up arrow, down arrow and then the string "this is a test of everything". You can see how it reads 3 characters at a time.

```
>>> import key1
>>> key1.test()
Got ['\033[A']
Got ['\033[B']
Got ['thi']
Got ['s i']
Got ['s a']
Got [' te']
Got ['st ']
Got ['of ']
Got ['eve']
Got ['ryt']
>>>
```

There is a problem with key1.py which is annoying. If the loop doesn't complete for any reason and ttyLinux.setNormal() is not called (say you typed a control-C) then your terminal is left in the special state even though control has returned to Python or to the shell. You can still issue commands, they just don't echo. Nor can you correct mistakes. One way out is to get to the shell and "exit" to the login prompt. A new login will reset the terminal characteristics.

But a better way to do this can be found in key2.py and this is the model we'll use in the game program. Here we use a try/finally clause (no except) to guarentee that setNormal() is called whether or not the call to loop() completes normally. Without an "except" clause we still get any traceback messages.

```
>>> import key2
>>> key2.test()
Got ['\033[A']
Got ['\033[B']   (Control-C typed here)
Traceback (most recent call last):
File "stdin", line 1, in ?
File "key2.py", line 14, in test
loop()
File "key2.py", line 7, in loop
time.sleep(1)
KeyboardInterrupt
```

```
>>> # The keyboard still works fine!
```

Dealing with the board and screen.

The module boards.py contains our board as a list of python strings, each string representing one row on the screen. This format is convenient for two reasons. One, it is easy to edit the board since the rows are aligned vertically and, two, reading the board into the program is done by simply importing the module. If you want to edit the board with "vi" or "vim", use the "R" command to get into "replace" mode. Within the strings '_' are catwalks, '|' are ladders and '*' are the lodes of treasure. The players are not represented on the initial board.

The list "boards" references board0 in a single item. This is meant to leave room for expansion to allow you to have several boards and let the game switch between them. In the traditional Lode Runner game winning at one level automatically advanced you to the next.

The module util.py has some functions to automatically handle the board and the screen so that the main game program does not have to worry about petty details. The function setBoard sets the global "board" to a fresh copy of the nth board in the module boards. The next two functions let us read and write a single spot in the board. The try/except clauses keep out of bounds references from crashing the program.

```
>>> import util
>>> util.getSpot(13,38)
'|'
>>> util.setSpot(13,38,'X')
>>> util.writeBoard()
                             __*___|_
                       |           |
                       |           |
                      _|__*________|______*___
                       |           |
               ___*____|___________|_
                       |           |
                       |           |
                       |           |
 ____*_________________|___________X______*____
>>>
```

Our call to util.setSpot changed a single character on the board.

Notice that the function writeBoard uses the escape sequence to erase the screen followed by a function writeScreen to position the cursor at each line and then write the characters for that line. This is followed by a "flush" call which guarentees the output is not cached in memory but rather output immediately. Without the flush the game can be quite jerky.

The function writeScreen will also be used in the game program to move the players on the screen. For example, if you, (represented by "^"), are on a catwalk and are to move one column to the right, we use writeScreen once to place a "_" where you currently are and then again to place a "^" at your new location, one position to the right.

The game loop

(lode1.py) is the first of five programs on our way to making a real computer game.

We first import everything from the util module which, in turn, imports the board. We also import the time module for its sleep function. We'll have a single player represented by the '^' character which simply moves according to where it is sitting on the board. In open air (' ') it falls down (and to a higher row), on a catwalk ('_') it moves to the right and finally, on a ladder ('|') it climbs up. In between each move the program pauses for .1 second. Just before pausing the program calls writeScreen(20,0,'') to simply get the cursor out of the picture.

Make sure you have ttyLinux.py, util.py and lode1.py. Run the program with

```
>>> import lode1
>>> lode1.main()
```

The board appears on the screen and a couple of seconds later "^" appears falling to the catwalk, moving to the right, and climbing the ladder. You will need to press contol-C to stop the program.

Adding keyboard control

One problem with lode1.py is that information about You is contained in variables 'row' and 'col'. If we have multiple players we need to do better.

A great approach is to have an object (instance of a class) represent each player. Each object can remember not only row and column but also lots of other information and with Python this can be very open-ended.

With lode2.py we define a class "You" which has attributes for the above as well as the direction you want to travel. A method "setDirection" uses the latest keyboard input available (keys) and sets your direction accordingly. Notice that direction is contained in a 2 value tuple with horizontal and vertical components.

A separate method "move" is used to change your location on the screen. You can't always move in the direction you want. In free space you fall. On a ladder you can only go up or down, on a catwalk left or right. Later we'll fix that so that you can transfer onto ladders from catwalks and vice versa.

The main function is the try/finally clause discussed earlier in the program key2.py. The game loop is in the function playGame.

Run the program. You will fall to the catwalk and stop. A right arrow starts you to the right but you stop at the ladder. An up arrow climbs and finally a control-C will exit.

```
>>> import lode2
>>> lode2.main()
```

Adding more players.

The next version lode3.py adds a robot to play against you. Since it makes sense to have "you" as a class instance it makes just as much sense to do the same for the robot. In fact both you and the robot can share the same "move" method. So let's use inheritance to have a superclass "Player" and subclasses "You" and "Robot".

This game, even though not complete, is fun to play. Let's look closely at the robots setDirection method which is where most of the new code resides.

The global "inPlay" is true if the game is still going. If the robot sees that he has tagged you then "inPlay" is set to false. Next the robot checks to see you are on the same level and runs towards you if so. If the robot finds itself on a ladder, it attempts to match your vertical location.

This strategy works pretty well on this board, even though it is quite simple. If you make more complex boards you will find the robots overly challenged and may want to program in some additional "smarts".

The move method in the Player class also has an addition. The use of variables lspot and rspot (for left/right) enable both "you" and the robot to transfer to and from ladders and catwalks.

Finally there is a variable "clock" which just keeps track of howmany moves have taken place. It is used in this version to give you a 4 second (40 moves) head start before the robot is created and comes after you. However the availability of a clock can greatly enhance what you can do with the game.

Keeping score. Burning holes.

Our next version lode4.py adds score keeping and a defense mechanism against the robot. Score is kept as an attribute of "you" and ten points are added each time you pass over a lode ('*'). The lode is then erased from both the screen and the board. Just before taking its .1 second nap the program writes the score to the screen in lower left corner where it moved to get the cursor out of the way.

The defense mechanism lets you burn a hole in the catwalk, either to your left ("a" key) or right ("s" key). The idea is that you'll work the arrow keys with your right hand and the 'a' and 's' keys with your left. These holes don't repair themselves and the robot and you fall through them to the next level down or off the screen. Your falling off the screen ends the game.

The final version for now

Our last version of the game will demonstrate a couple of possible ways to extend the game.

lode5.py will create 2 robots to chase you. The list "players" keeps track of "you" and whenever its length is under 3 it adds a robot. You can kill a robot by making it fall through the lowest catwalk. When a robot sees that it is dead it politely removes itself from the "players" list. But once that happens the list length is less than 3 and a new robot drops from the top at a random column (look for "random" in the code).

Because this game is a little too challenging (for me anyway) I also made the robots move at half speed. This is done by having a Robot.move method that calls Player.move every other time, when "clock%2 == 0"

Some ideas for further exploration.

The best thing about programing your own games is that you can extend them any way you want, at least if you can figure out how. Here are a just a few ideas.

> Play around with the speed of the game (time.sleep) and the number of robots. Let the user control some of this on starting the game. Level=easy
>
> Make more boards and add them to the boards list. Create a mechanism to have the program advance a level when you win or let you choose a level. Level=easy
>
> Change the rules. Perhaps instead of burning holes drop a "stun bomb" that stuns a robot for a few seconds keeping it from moving. Level=medium
>
> Make the robots more intelligent, keeping track of the last ladder used by "you" and heading toward it when other options fail. Level=medium
>
> The game "PacMan" is actually quite similar. Use this code as a basis for a PacMan like game. Level=medium
>
> Make "you" an automatic player like the robots. You must avoid the robots and try to reach treasure. Level=hard
>
> Make the game for two "You" players running on autopilot. Each may use a different strategy in their setDirection method. Level=hard
>
> Make the robots work cooperatively, anticipating moves of the other robots and your responses. Level=PhD Thesis

If you are interesting in the more challenging possibilities take a look at the following website.

http://icfpcontest.cse.ogi.edu/task.html