Task 2 – Lab: FSM & Python

Finite State Machines

Individually, or in small groups, develop a finite state machine design (on paper) for a single NPC of a computer game genre of your choice. Your design must:

- Start with identification of at least three states (write a list) ("defending", "attacking" …),
- List the variables needed (at least two) (eg "thirst" level, "money", "damage" …),
- Identify the actions or variables that will cause a state change (in words),
- Create a state diagram to describe your system,

Show your design to the tutor and make sure you are on the right track.
Keep your design for use in later work. (It's okay if you Word or pen/paper for your design)

Getting Started with Python

How you get started with Python will depend on your previous experience with programming and the type of languages you know. The lab machines should have python and a number of tools installed. However, for this first lab, you can simply use a web-based python interpret so that you don't have to worry about software. Use Python 3+ if possible.

If the required software is not setup on the lab machines (or on your own computer), you will need to download and install this first. It *should* be installed.

Is Python Installed and Working OR Web-Based?

- Check the start menu for Python. Run the IDLE program if available
- If not installed, get Python from http://www.python.org/

Python From the Command Line?

- If you open a command window and type "python" does the shell start?
- If not, you may want to set the environment path so that it knows where to find python.
- Create a simple "hello world" python script file and run it from the command line

Create a Simple FSM in Python

- Using a simple loop with if statements be able to implement part, or all of, your FSM design

Optionally, you might want install and try:

- Pyglet from http://www.pyglet.org cross-platform multimedia (with OpenGL)
- PyGame from http://www.pygame.org/ (try the game demos)
- PyDev – a plugin module for the Eclipse editor (if you know Eclipse already).
- PyTools – a plugin module for Visual Studio (if you know Visual Studio already.)

By the end of this lab you should have created a basic FSM in python and be able to run (and debug it) using simple tools from the command line.

Python Micro-Quick Reference Sheet

Keywords:

```
and, assert, break, class, continue, def, del, elif, else, except, exec, finally,
for, from, global, if, import, in, is, lambda, not, or, pass, print, raise,
return, try, while.
```

Basic Concepts

- python IS CaSe SeNsiTiVe
- indentation (whitespace) defines blocks (not braces {…} or begin/end)
- `#` for single line comments (no block comment)
- `=` is for assignment, and assignments creates references – not copies – of objects
- `==` is for logical comparison, also `<= != < > >=`
- Python uses ***reference counting*** and ***garbage collection***
- Logical operators in words `and`, `or` and `not`, with `True` and `False` types
- ; can be used to terminate lines or separate statements (but not recommended)
- : is used to define structure or control (see `if`, `else`, `while` flow control)
- `None` is a special constant (think NULL, void or nil), logical `False`
- "Duck" typing; *if it looks like a duck, walks like a duck, quacks like a duck… it is a duck.*

Sequence Types

- index with [index], start at zero, range selection ("slice") with [start:end]
- `tuple1 = (1,'a',None,3.42,'hello')` # tuple – immutable, ordered, mixed types
- `list1 = ['abc', None, 5, 7.4]` # list – mutable, ordered, mixed types
- `str1 = 'have a nice day!'` # string, '…', "…", or triple "'…"', """…"""
- `c = a + b` # will create new tuple/list/string (lists can have .extend(..) or .append(…))
- `c = a * 3` # will create new multple of tuple/list/string

Dictionary

Stores a mapping (hash) between key/values. We can define, view, delete etc.
Unordered. Mutable. Mixed types. (Nested.) Keys are unique.

```
dict1 = {'key1':'yippy', 'key2': 1234, 'key12', None}
```

Procedures/Functions

```
def my_function(param1, param2, param3=10):
      """ Documentation string… don't' state the obvious.  """
      print "params: ", param1, param2, param3
      return param1 + param2 / param3 # default return None if not specified
```

No function overloading (but you can specify default parameter values etc)
Functions ***are*** objects and can be passed around just like variables

Flow Control

`if test1:`
`  …`
`elif test2:`
`  …`
`else:`
`  …`
	`while test_is_true:`
	`  …`
	`  if move_on_test:`
	`  continue`
	`  if break_test:`
	`  break`
		`for item in collection:`
		`  print item`
		# works on any collection or
		# iterable object
		`for i in range(10):` # creates an iterable range
		`  print i`

```
assert (current_score < 10), 'Hey, why did that happen?'
x = true_value if condition else false_value # new in 2.5
```

List Comprehensions

Easy way to create lists, often using existing list/iterables. It's a very popular python technique. They look a bit like a `for` loop and an `if` statement (optional). They can be nested (gets messy).

```
nums = [1,3,4,6,8,9]
new_squared_list = [elem*2 for elem in nums] # or
new_odd_list = [elem for elem in nums if num % 2 ]
```

Import Modules

```
import somefile # can then refer to somefile.className1 or somefile.function3
from somefile import className1, variable5, function3 # or * to include
```