Meat Engine Overview

Author: Dave LeCompte

Contact: pyweek at bigdicegames dot com

Contents

Introduction

One practical definition of a “game engine” is “the code you reuse on your second game”. That’s mostly tongue-in-cheek, but it isn’t far from the philosophy of what’s currently included in MeatEngine.

I’ve included a variety of pieces of code, not because they’re all appropriate for the game you want to make, and they’re certainly not all appropriate for any particular game, but instead, each piece might be useful for some game, or for some game-related-project.

MeatEngine is more of a toolbox than a framework - you’re responsible for the main loop of your program. Some of the code makes certain assumptions about being called periodically. This should not be difficult to handle, regardless of the structure of your game.

Included (currently) are modules for adaptive music, GUI display, AI, and low-level math. Also included is the beginnings of a ray tracer - not that you’d want to use a ray tracer in your game (certainly not this ray tracer, anyway), but it may be of benefit in creating assets. Also, it serves as a test harness for the math module.

Logic.fsmMgr

This is sample code that can be used to manage a hierarchical Finite State Machine (hFSM). I use states for the notional screens that I display to the user, but also for levels and for things that can interrupt gameplay. If a dialog box or menu pops up over gameplay, that’s a state that allows its parent to draw, then draws a small part of the screen on top of the parent’s display.

When doing a normal (non-hierarchical) transition, pop the current state off, push a new state on. When transitioning to a child state, simply push the new child state on. When done with a child state, pop it back off again.

Pushing States On the Stack

Create objects subclassed off the State class during gameplay. Push them on the stack using fsmMgr.pushState. Within your main game loop, call the update, draw, handleKey and handleMouse functions of the Logic.fsmMgr code.

Popping States Off the Stack

When the state is complete, pop it off the stack using fsmMgr.popState.

Things To Implement in Your Code

Within your State objects, implement the handleMouseButton, handleKey, update, and draw functions.

Example

```
#main loop

from MeatEngine.Logic import fsmMgr

fsmMgr.pushState(MyState(None))

gameOver=False
while not gameOver:
  ms=myClock.tick(MAX_FRAME_RATE)

  for event in pygame.event.get():
    if event.type == KEYDOWN:
      fsmMgr.handleKey(event.key, event.unicode)
    elif event.type == MOUSEBUTTONDOWN:
      fsmMgr.handleMouseButton(True, event.button)
    elif event.type == MOUSEBUTTONUP:
      fsmMgr.handleMouseButton(False, event.button)

  if not fsmMgr.update(ms): #re-
turns False if no more states left.
    gameOver=True
  glClear(GL_COLOR_BUFFER_BIT)
  fsmMgr.draw()
  pygame.display.flip()


# MyState

class MyState(fsmMgr.State):
  def __init__(self):
    fsmMgr.State.__init__(self)

  def update(self, dt):
    # dt is in seconds
    # do your update logic here
    pass

  def handleMouseButton(self, bDown, buttonIndex):
    # you must return True if you handle the mouse event, other-
wise
```

```
    # the event gets raised to the next state down the stack.

    return True

  def handleKey(self, key, unicode):
    # return True if you handle the key event, false other-
wise for
    # the key to go to the next state down the stack.


    return True

  def draw(self):
    # you can call self.parent.draw() if this state doesn't draw the
    # entire screen (e.g. a dialog box)
    pass
```

Math.Voronoi

This is a class to generate Voronoi and Delaunay meshes. A Voronoi Diagram for a set of points is a partitioning of space into regions. Each point p has an associated region which is all the points in the plane that are closer to p than to any other point in the set.

See: http://en.wikipedia.org/wiki/Voronoi_diagram

The "dual graph" of the Voronoi Diagram is the Delaunay triangulation. This means that for every region in the Voronoi Diagram, there is a point in the Delaunay triangulation and vice versa. Delaunay triangulations are useful to generate triangular meshes with few sliver triangles.

See: http://en.wikipedia.org/wiki/Delaunay_triangulation

Setup

Create three points that define a triangle more than big enough to encompass the data points you wish to use. You can think of this as stretching out a canvas within which to work.

Create a Math.Voronoi.subdivision.subdivision object, passing in the three points.

Adding Points

For each point in your data set, add it to the subdivision object using insertSite.

Retrieving the Voronoi Edge List

At any time, you may call the subdivision's dumpEdges() method, which will return a list of edge objects. e.org() and e.dest() are the endpoints of the edges.

Example

```
from MeatEngine.Math.vector import Vec2f

p1=Vec2f(-1000.0, -1000.0)
p2=Vec2f( 1000.0, -1000.0)
p3=Vec2f(-1000.0,  1000.0)

s=MeatEngine.Math.Voronoi.subdivision.Subdivision(p1,p2,p3)

for p in myDataPoints:
  s.insertSite(p)

edgeList=s.dumpEdges()

for e in edgeList:
  origin=e.org()
  destination=e.dest()

  canvas.drawLine(origin, destination)
```

Math.Vector

Classes are provided for two- and three-dimensional vectors of floating point numbers. Note that, at this time, these vectors are mutable, so are not suitable for being keys in dictionaries. Also, care should be taken to make copies of the vectors (perhaps by multiplying by a scale factor of 1.0) when a copy is desired.

MoodMusic

MoodMusic implements a Finite State Machine that can be used to play music, selecting appropriate music based on the state of the game.

This is accomplished by first defining states and associating music (individual songs or entire directories) with each state.

At runtime, tick the player object, and signal state changes when the mood of your game changes.

Example

```
from MeatEngine import MoodMusic
lib=MoodMusic.ibraryPyGame.LibraryPyGame()
playerObj=MoodMusic.player.Player(lib)

pygame.init()

playerObj.addState("intro")
playerObj.addState("happy")
playerObj.addState("medium")
playerObj.addState("tense")

playerObj.addMusicDirectoryToState("intro","Music/Ogg/Intro")
playerObj.addMusicDirectoryToState("happy","Music/Ogg/Happy")
playerObj.addMusicDirectoryToState("medium","Music/Ogg/Medium")
playerObj.addMusicDirectoryToState("tense","Music/Ogg/Tense")

playerObj.setState("intro")


gameOver=False

while not gameOver:

  # game-define update function
  updateGame()

  curState=playerObj.getState()

  # game-defined mood function -
returns a string matching our four
  # state labels.
  m=getMoodOfGame()

  if m != curState:
    playerObj.pushState(m)
  playerObj.tick()
  pygame.display.flip()
```

Widgets.wordWrap

A modular word wrap function is provided. Depending on your text display system, the width of a string may be calculated based on font metrics, or it may be that all characters have the same width. In the case where a fixed-width font is being used, you can use the default width calculator. Otherwise, you need to pass in a function that returns a width value when passed a string.

Example

```
text=open("lorem_ipsum.txt").read()

#wrap for 72 column fixed width display
lines=wrap(text, 72)

for oneLine in lines:
  print oneLine

#define a width calculation function

def myWidthFunc(testString):
  # find the width in pixels of this string

  return getwidth(testString)

#wrap for 400 pixel GUI display

lines=wrap(text, 400, myWidthFunc)

yPos=0
for oneLine in lines:
  canvas.drawText((0,yPos),oneLine)
  yPos += getheight(oneLine)
```