Alpheus Madsen

(435)764-5172
alpheus.madsen@gmail.com

A Simple 3D Graphics Engine Written in Python and PyGame

Tuesday, March 3, 2009.

Introduction. Several years ago I wrote a simple 3D graphics engine in Python and Allegro. I wanted to grow that into a full-fledged game of some sort, but because of graduate studies, I had to put off working on this little program. Now that I have completed my doctorate in mathematics, I have spent some time trying to decide what I wanted to do next. I remembered this program, and how I liked the combination of linear algebra, computer graphics, and pretty pictures, and I decided that I should pursue computer graphics programming.

As I began to look for work, I stumbled onto "dual quaternions" and I decided that I wanted to learn more about them. Thus, I decided to dust off this program, implement dual quaternion math, and use it for a simple game. In the month that I have done this, I have made much progress; this document is a sampling of the source code of my project. Although I have corrected many bugs since then, this document only contains source code that represents major changes that I have made.

Unlike my original engine, which used the Allegro library via PyAllegro Python bindings, this new engine uses the SDL game library, through PyGame Python bindings. Although I liked PyAllegro, the project is currently slow and unstable, and I had problems getting it to work; thus I chose to use a more popular library. The conversion process went surprisingly well!

Progress on My Project. When I started my work on this engine, I decided to follow the suggestion of my wife and keep a journal of the progress I have made. Thus, I will not detail the changes or future goals of my project here; those are provided in a separate document. I will, however, provide a brief outline of what I have completed:

- I wrote wireframe data for worm segments to be used in a Nibbles3D game;
- I implemented dual quaternions, and wrote functions that would easily allow for arbitrary combinations of rotations and translations, and convert dual quaternions to matrices;
- I implemented a cursor system that would allow me to use the mouse to manipulate the movements of my worms;
- I separated wireframe data from position data, to minimize memory used;
- I discovered and corrected some subtle camera bugs that, once corrected, allowed things to begin to work in a very intuitive way;
- I added coordinate systems to the "movable" class, so that I could rotate objects (particularly cameras) by relative coordinates;
- I created a "worm" class that combines segments and two separate cameras (one for debugging purposes) that will be used for creating worms.

Personal Remarks. As I have revived this project, it has in some ways annoyed me greatly! Often, I couldn't stop thinking about the things I needed to do, even at times when I so desperately needed to sleep,

or other times when I needed to do my work as a computer programmer for EV Source. I simply wish that I had an infinite amount of time that would allow me to do everything I wanted with this project, and to fulfill all my other obligations to health, family, work, and whatever else I would like to do; at the very least, I wish that my friends and family could somehow box up extra time they had (a few minutes here and there) to send it to me.

On the other hand, working with quaternions, especially dual quaternions, has opened up a new world to me. Dual quaternions have a simplicity to them that you can't recognize until you really begin to work with them, and to use them in computer graphics especially. If I do nothing more with my project, this alone was worth reviving it!

"QUAT.PY". After converting my program to PyGame, the first thing I did with my project is implement dual quaternions; this was, after all, the reason I was inspired to revive this little program! I decided that, rather than implement two classes (say, perhaps "quat" and "dualquat"), that I would implement one class, and then test for special cases to determine where I could simplify multiplication or division.

Perhaps it would be more efficient, computationally, to separate things into "quat" and "dualquat", and then adjust the code accordingly; for now, however, I like this implementation.

```
# quat.py

# This module defines an entirely new number:  the dual quaternion.
# Since a normal quaternion is just a dual quaternion with a zero
# idempotent part, I have chosen to define a single class---quat---and
# that will act as both quaternion and dual quaternion.

# Observe two things, though:
#    First, I tried to keep small numbers as "zeros"
# by rounding divisions (see __div__ and norm) to
# five significant digits.  So if a number is one
# like 5.2e-6, it will be rounded to 0.
#   To allow for experimentation, though, I decided to
# change the roundoff to ROUNDOFF, set in vertex.
#    Second, to make sure that division works
# appropriately, I initialized the original quaternion
# with float(etc).
from math import sqrt, acos
import exceptions
#from mm import multimethod

from vertex import *
from matrix import *


class DivideByPureDualException(exceptions.Exception):
   def __init__(self):
      return

   def __str__(self):
      print "","Attempt to find Inverse for Pure Dual Quaternion."

class DualDistanceException(exceptions.Exception):
```

```
    def __init__(self):
        return

    def __str__(self):
        print "","Attempt to find dist() for Pure Dual Quaternion."


class quat(object):
    #@multimethod(quat, float, vector, float, vector)
    def __init__(self, r=0, I=Zero, er=0, eI=Zero):
        # This initialization uses vectors rather than numbers.
        # This initializes to 0.
        self.r = float(r)
        self.I = I
        self.er = float(er)
        self.eI = eI

    #@multimethod(quat, float, float, float, float, float,
    #        float, float, float)
    #def __init__(self, r, i, j, k, er, ei, ej, ek):
    def real_quat(self, r, i, j, k, er, ei, ej, ek):
        # This is an initialization using numbers.
        self.r = float(r)
        self.I = vector(i, j, k)
        # Alternate to using vector definitions:
        #self.i = float(i)
        #self.j = float(j)
        #self.k = float(k)

        self.er = float(er)
        self.eI = vector(ei, ej, ek)
        # Alternate to using vector definitions:
        #self.ei = float(ei)
        #self.ej = float(ej)
        #self.ek = float(ek)
        # Note that this will be the only function that shows this
        # alternative, unless I decide to switch to this alternative.

    #@multimethod(quat, int, vector)
    #def __init__(self, beta, r):
    def rotation(self, beta, r):
        # This initialization creates a unit quaternion.
        self.r = tcos(beta/2)
        self.I = r.norm()*tsin(beta/2)
        self.er = float(0)
        self.eI = Zero

    #@multimethod(quat, vector)
    #def __init__(self, t):
    def vector(self, t):
        # This initialization creates a unit dual quaternion.
```

```
    # As __init__, it is meant to be used for two purposes:
    #  to multiply vectors as dual quaternions;
    # and to initialize traslation dual quaternions.

    # For the latter purpose, it's important to remember to
    # divide the vector by two before initializing the
    # quaternion.
    self.r = float(1)
    self.I = Zero
    self.er = float(0)
    self.eI = t

def pureVector(self, t):
    """This sets the quat to be a pure quaternion with a
    vector value of t; this is useful if all we want to do with
    a given vector is rotate it."""
    self.r = float(0)
    self.I = t
    self.er = float(0)
    self.eI = Zero

def extractVector(self):
    """When we translate and/or rotate a vector, we are going
    to want to get the new vector information somehow.  This
    function returns the vector from a unit dual quaternion.

    Recall that a vector as a dual quaternion has the form
             1 + eI.dot(v) where I = (i, j, k) and e**2==0."""
    return self.eI

def extractPureVector(self):
    """When we only rotate a vector, we are going
    to want to get the new vector information somehow.  This
    function returns the vector from a pure vector quaternion.

    Recall that a vector as a pure quaternion has the form
             I.dot(v) where I = (i, j, k)."""
    return self.I

def translation(self, t):
    # This initialization creates a unit dual quaternion
    # for translations.
    self.r = float(1)
    self.I = Zero
    self.er = float(0)
    self.eI = t/2

def purequat(self):
    # We'll check to see if each of these items are non-zero.
    # If any one of them is non-zero, then this is not a pure
    # quaternion.
```

```
    return not(self.er != 0 or self.eI.x != 0 or
               self.eI.y != 0 or self.eI.z != 0)

def puredual(self):
    # We'll check to see if each of these items are non-zero
    # If any one of them is non-zero, then this is not a purely
    # dual quaternion.
    return not(self.r != 0 or self.I.x != 0 or
               self.I.y != 0 or self.I.z != 0)

def puretrans(self):
    # This identifies a dual quaternion that is purely a translation.
    return (self.r == 1 and self.I.x == 0 and self.I.y == 0
               and self.I.z == 0 and self.er == 0)

def __add__(self, v):
    r = self.r+v.r
    I = self.I+v.I
    er = self.er+v.er
    eI = self.eI+v.eI
    return quat(r, I, er, eI)

def __sub__(self, v):
    r = self.r-v.r
    I = self.I-v.I
    er = self.er-v.er
    eI = self.eI-v.eI
    return quat(r, I, er, eI)

def __neg__(self):
    return quat(-self.r, -self.I, -self.er, -self.eI)

def __mul__(self, q):
    if isinstance(q, quat):
        # Do quaternion multiplication
        if self.purequat() and q.purequat():
            r = round(q.r*self.r - self.I.dot(q.I), ROUNDOFF)
            I = q.I*self.r + self.I*q.r + self.I.cross(q.I)
            return quat(r, I)
        else:
            r = round(self.r*q.r - self.I.dot(q.I), ROUNDOFF)
            I = q.I*self.r + self.I*q.r + self.I.cross(q.I)
            er = round((self.r*q.er - self.I.dot(q.eI)) + \
                       (self.er*q.r - self.eI.dot(q.I)), ROUNDOFF)
            eI = (q.eI*self.r + self.I*q.er+self.I.cross(q.eI) + \
                       q.I*self.er + self.eI*q.r + self.eI.cross(q.I))
            return quat(r, I, er, eI)
    else:
        # Do scalar multiplication
        r = round(self.r*q, ROUNDOFF)
        I = self.I*q
```

```
        er = round(self.er*q, ROUNDOFF)
        eI = self.eI*q
        return quat(r, I, er, eI)

    def conj(self):
        # Quaternionic conjugation
        return quat(self.r, -self.I, self.er, -self.eI)

    def pconj(self):
        # Pure Quaternionic conjugation
        return quat(self.r, -self.I)

    def econj(self):
        # Dual quaternionic conjugation
        return quat(self.r, self.I, -self.er, -self.eI)

    def bconj(self):
        # Quaternionic and Dual Quaternionic conjugation together
        return quat(self.r, -self.I, -self.er, self.eI)

    def dot(self, v):
        # I'm not sure if this makes sense for dual quaternions.
        return round(self.r*v.r + self.I.dot(v.I) + self.er*v.er \
               + self.eI.dot(v.eI), ROUNDOFF)

    def dist(self):
        if self.purequat():
            return round(sqrt(self.r*self.r + self.I.dot(self.I)), \
                       ROUNDOFF)
        else:
            r = round(sqrt(self.r*self.r + self.I.dot(self.I)), \
                       ROUNDOFF)
            if r == 0:
                raise DualDistanceException
            dot = float(self.I.dot(self.eI))
            if dot == 0:
                return round(r, ROUNDOFF)
            else:
                return quat(round(r, ROUNDOFF), Zero, \
                          round(dot/r, ROUNDOFF), Zero)

    def norm(self):
        d = self.dist()
        if self.purequat():
            return quat(self.r/d, self.I/d)
        else:
            return quat(self.r/d, self.I/d, self.er/d, self.eI/d)

    def inverse(self):
        if self.purequat():
            # Pure quaternion inverse:  this is rather easy!
```

```
            return self.pconj() / sqrt(self.r*self.r + self.I.dot(self.I))
        else:
            # Dual quaternionic inverse:  a little more challenging.
            if self.puredual():
                # If the non-idempotent part is 0, we can't find
                # the inverse.
                raise DivideByPureDualException
            else:
                # the inverse is 1/q_0 - e(q_e/((q_0)^2) --- yikes!
                r_sq = self.r*self.r
                I_dot = self.I.dot(self.I)

                t = 1.0/(r_sq + I_dot)

                new_r = r_sq - I_dot
                new_I = self.I*(1.0/(new_r*new_r + 4*r_sq*I_dot))

                er = -self.er*new_r - self.eI.dot(new_I)
                eI = -(new_I*self.er + self.eI*new_r + self.eI.cross(new_I))

                return quat(self.r*t, -self.I*t, round(er, ROUNDOFF), eI)

    def __div__(self, q):
        # Theoretically, I ought to repeat the above, simplified, to
        # streamline execution.
        # For now, I am NOT going to do that!
        if isinstance(q, quat):
            return (self*q.inverse())
        else:
            r = round(self.r/q, ROUNDOFF)
            I = self.I/q
            er = round(self.er/q, ROUNDOFF)
            eI = self.I/q
            return quat(r, I, er, eI)

    def normalize(self):
        self = self.norm()

    def matrix(self):
        # Here are a few calculations that will be used to
        # convert our dual quaternion to a matrix!

        # To speed things up, we'll recognize three types of
        # matrices.

        # Note that, theoretically, we should normalize the
        # dual quaternion first.

        if self.purequat():
            # This is the rotation quaternion...
            w = self.r          # = tcos(beta/2)
```

```
        x = self.I.x        # = tsin(beta/2)*axis.x
        y = self.I.y        # = tsin(beta/2)*axis.y
        z = self.I.z        # = tsin(beta/2)*axis.z

        # Now we'll pull out the translation information...
        xx2 = 2*x*x; yy2 = 2*y*y; zz2 = 2*z*z
        wx2 = 2*w*x; wy2 = 2*w*y; wz2 = 2*w*z
        xy2 = 2*x*y; xz2 = 2*x*z; yz2 = 2*y*z

        quat_mx = matrix()
        quat_mx.mx = [ [1-yy2-zz2, xy2+wz2, xz2-wy2, 0],
                       [xy2-wz2, 1-xx2-zz2, yz2+wx2, 0],
                       [xz2+wy2, yz2-wx2, 1-xx2-yy2, 0],
                       [0, 0, 0, 1] ]

        return quat_mx

    elif self.puretrans():
        # This is a purely translational dual quaternion.
        quat_mx = matrix()
        quat_mx.mx = [ [1, 0, 0, 2*self.eI.x],
                       [0, 1, 0, 2*self.eI.y],
                       [0, 0, 1, 2*self.eI.z],
                       [0, 0, 0, 1] ]

        return quat_mx

    else:
        # This is the rotation quaternion...
        w = self.r          # = tcos(beta/2)
        x = self.I.x        # = tsin(beta/2)*axis.x
        y = self.I.y        # = tsin(beta/2)*axis.y
        z = self.I.z        # = tsin(beta/2)*axis.z

        # Now we'll pull out the translation information...
        ew = self.er
        ex = self.eI.x
        ey = self.eI.y
        ez = self.eI.z

        t = 2*(-ew*x + ex*w - ey*z + ez*y)
        u = 2*(-ew*y + ex*z + ey*w - ez*x)
        v = 2*(-ew*z - ex*y + ey*x + ez*w)

        xx2 = 2*x*x; yy2 = 2*y*y; zz2 = 2*z*z
        wx2 = 2*w*x; wy2 = 2*w*y; wz2 = 2*w*z
        xy2 = 2*x*y; xz2 = 2*x*z; yz2 = 2*y*z

        quat_mx = matrix()
        quat_mx.mx = [ [1-yy2-zz2, xy2+wz2, xz2-wy2, t],
                       [xy2-wz2, 1-xx2-zz2, yz2+wx2, u],
```

```
                [xz2+wy2, yz2-wx2, 1-xx2-yy2, v],
                [0, 0, 0, 1] ]

        return quat_mx

    def __str__(self):
        return " %s + I %s + e(%s + I %s)" % (self.r, self.I, \
                  self.er, self.eI)

    def get_angle_axis(self):
        # This returns (beta, axis) from a quaternion.  Note that
        # this assumes that the quaternion is pure; this function
        # should probably throw an exception if it isn't pure.

        # Note that we convert from radians to bradians.
        beta = int(round(2*acos(self.r)*128/pi))
        if beta == 0:
            # In this case, we have an "identity" rotation;
            # thus, we could use any vector we would like.
            # Here, we'll default to the Up vector.
            v = Up
        else:
            sin = tsin(beta/2)
            if sin == 0:
                # If beta is 1 or -1, then it will be a valid rotation;
                # in this case, we'll approximate the sine.
                sin = tsin(beta)
            x = self.I.x/sin
            y = self.I.y/sin
            z = self.I.z/sin

            v = vector(x, y, z)
            v = v.norm()

        return (beta, v)

    def get_translation(self):
        """This returns the translation vector from a translation
        quaternion. It assumes that the dual quaternion is purely
        translation.  If it isn't, I should probably throw an
        exception."""
        return vector(2*self.eI.x, 2*self.eI.y, 2*self.eI.z)

    def get_angle_axis_translation(self):
        """ This returns the angle-axis and tranlation information
        from a dual quaternion; this does NOT assume that this is
        purely a translation or a quaternion.

        Every dual quaternion represents a combination of rotation
        and translation; in thinking about the relationships of
        these two, I have been able to come up with this."""
```

```
        # First, we get the individual information; I use quaternions
        # for both for efficiency reasons.  (It takes less adds and
        # mults to multiply pure quaternions.
        rotation = quat(self.r, self.I) # this is the rotation information
        translation = quat(self.er, self.eI)  # This is ALMOST
                                              # the translation info

        # Now, we'll remove the rotation info from the
        # translation portion.
        translation = translation*rotation.inverse()

        beta, axis = rotation.get_angle_axis()
        pos = vector(translation.I.x*2, translation.I.y*2, \
                  translation.I.z*2)

        return beta, axis, pos

# Here are a few quaternion constants that are nice to
# define:  in particular, note that [Left, Up, Fwd]
# is a left-hand coord system, while [Right, Up, Fwd]
# represents a right-hand one.
BetaRight = [0, 64, 128, 192]
Identity = quat(1, Zero)

XTrans = quat(); XTrans.translation(vector(100, 0, 0))
YTrans = quat(); YTrans.translation(vector(0, 100, 0))
ZTrans = quat(); ZTrans.translation(vector(0, 0, 100))

XRot0 = quat(); XRot0.rotation(BetaRight[0], vector(1,0, 0))
XRot64 = quat(); XRot64.rotation(BetaRight[1], vector(1,0, 0))
XRot128 = quat(); XRot128.rotation(BetaRight[2], vector(1,0, 0))
XRot192 = quat(); XRot192.rotation(BetaRight[3], vector(1,0, 0))

YRot0 = quat(); YRot0.rotation(BetaRight[0], vector(0,1, 0))
YRot64 = quat(); YRot64.rotation(BetaRight[1], vector(0,1, 0))
YRot128 = quat(); YRot128.rotation(BetaRight[2], vector(0,1, 0))
YRot192 = quat(); YRot192.rotation(BetaRight[3], vector(0,1, 0))

ZRot0 = quat(); ZRot0.rotation(BetaRight[0], vector(0, 0, 1))
ZRot64 = quat(); ZRot64.rotation(BetaRight[1], vector(0, 0, 1))
ZRot128 = quat(); ZRot128.rotation(BetaRight[2], vector(0, 0, 1))
ZRot192 = quat(); ZRot192.rotation(BetaRight[3], vector(0, 0, 1))
```

"POSITION.PY". When I started adding segments to my worm, I realized that I was repeatedly reading the same data from the hard drive and then putting it in position; to avoid roundoff error, I would leave the original position information unchanged! This seemed to be a great waste of both computer memory and computer processes (it takes a lot of work loading files from the disk, just to put things in memory), so I decided to separate the wireframe data from the position data.

```
# movable.py  -- This module defines a class from
# which 3D objects, such as cameras and wireframes,
# can inherit.  In doing so, they become manipulable
# by various means...particularly by matrices.

from movable import *

class position(movable):
   # This class should have position information, including its
   # own matrix! but it shouldn't have much more than that.
   def __init__(self, wf_data, beta=0, axis=Up, pos=Zero,
                beta_f=0, axis_f=Up, fwd=Fwd, left=Left, up=Up):
      """This class separates the position of a data set from
      the data set itself.  Since my nibbles program will be
      using the same vector data set for worms, I realized that
      I just needed to keep track of the position information for
      each segment!"""
      movable.__init__(self, beta, axis, pos, beta_f, axis_f, fwd, left, up)

      self.wf_data = wf_data

   # This function should draw the wireframe onto the camera!

   # Come to think of it, this function assumes that position is a
   # wireframe of some sort.  Instead, I should call this
   # draw_wf_Onto(self, camera) or something like that.

   # Of course, if I create some sort of world, that world will
   # probably be responsible for drawing things anyway, so
   # I'm not sure what to think exactly...
   def drawOnto(self, camera):
      # Note that this might be a good time to use a line like
      # self.setworld()
      # On second thought:  No, it wouldn't be a good place!
      # If the world matrix hasn't changed, there is no reason
      # to reset it!

      viewport = camera.camera_mx*self.world

      vtcs = []
      for i in self.wf_data.vertices:
         vtx = viewport.proj(i)
         # For non-homogeneous vertices:
         #    vtcs.append([319*(vtx.x/vtx.e-1)/2,
         #                 399*(vtx.y/vtx.e-1)/2])
         vtcs.append([vtx.x, vtx.y])
         # For non-homogeneous vertices:
         #   vtcs.append([vtx.x/vtx.e, vtx.y/vtx.e])

      # Now let's draw the lines!
      for i in self.wf_data.edges:
```

```
            pygame.draw.line(camera.film, egacolor[i.color], \
                     (vtcs[i.v1][0], vtcs[i.v1][1]), (vtcs[i.v2][0], vtcs[i.v2][1]))

        # Finally, I should draw the faces! but not for now...

    def sketchOnto(self, camera):
        """This function 'sketches' the wireframe onto the given camera.
        That is, this function sorts through the faces and edges, and
        decides what should be drawn first, or even what should be drawn
        at all.

        This will allow whatever is drawing the scene to sort through all
        the objects, to determine what order things should be drawn.

        I *think* this assumes objects are simply connected and convex!

        Rather, this function *ought* to 'sketch' a wireframe!  For now,
        it just draws it."""
        viewport = camera.camera_mx*self.world

        pVertices = []
        zValues = []
        for i in self.wf_data.vertices:
            vtx = viewport.proj(i)
            pVertices.append([vtx.x, vtx.y])
            # zValues.append([vtx.z])

        # Now let's figure out which lines and faces get drawn!
        # First, we initialize two arrays; they default to "True"
        #visibleFaces = [ True for i in range(len(self.wf_data.faces))]
        #visibleEdges = [ True for i in range(len(self.wf_data.edges))]

        # I should probably calculate the zValues for the faces
        # and edges; at least, the ones that will be drawn!

        # Next, we go through the faces, to see which ones will
        # be drawn:
        #fwd = camera.getFwdCoord()
        #viewCenter = camera.getCenterCoord()
        #for i, f in enumerate(self.wf_data.faces):
            #dot = f.normal.dot(fwd)
            #if dot > 0:
                #visibleFaces[i] = False
                #for j in f.edges:
                    #visibleEdges[j] = False
            ## We now calculate the vector from the center of the camera
            ## to the center of the face.
            #elif dot == 0:
                #center = viewCenter - f.center
                #if f.normal.dot(center) > 0:
                    #visibleFaces[i] = False
```

```
            #for j in f.edges:
               #visibleEdges[j] = False
      #print dot,
   #print
   #print "visFace ", visibleFaces
   #print "visEdges ", visibleEdges

   # Actually, instead of deciding which faces and edges I *won't*
   # draw, I will decide which I *will* draw.  This will be especially
   # important for the edges, because if I decide not to draw a face,
   # I may still want to draw one of its edges.  Thus, I will
   # initialize these as false.
   visibleFaces = [ False for i in range(len(self.wf_data.faces))]
   visibleEdges = [ False for i in range(len(self.wf_data.edges))]

   # I should probably calculate the zValues for the faces
   # and edges; at least, the ones that will be drawn!

   # Next, we go through the faces, to see which ones will
   # be drawn:
   fwd = camera.getFwdCoord()
   viewCenter = camera.getCenterCoord()
   for i, f in enumerate(self.wf_data.faces):
      dot = f.normal.dot(fwd)
      center = viewCenter - f.center
      #if dot < 0 or f.normal.dot(center) < 0:
      if dot >  0:
         visibleFaces[i] = True
         for j in f.edges:
            visibleEdges[j] = True
      # We now calculate the vector from the center of the camera
      # to the center of the face.
      #elif dot == 0:
      #   center = viewCenter - f.center
      #   if f.normal.dot(center) < 0:
      #      visibleFaces[i] = True
      #      for j in f.edges:
      #         visibleEdges[j] = True
      # print dot,
   # print
   # print "visFace ", visibleFaces
   # print "visEdges ", visibleEdges

   # Now, let's draw the faces that will be shown!
   for i, f in enumerate(self.wf_data.faces):
      if visibleFaces[i]:
         vList = []
         for j in f.vertices:
            vList.append(pVertices[j])
         pygame.draw.polygon(camera.film, egacolor[f.color], vList)
```

```
        # Next let's draw the edges!  At least, the ones that deserve to be drawn!
        for i, e in enumerate(self.wf_data.edges):
            if visibleEdges[i]:
                pygame.draw.line(camera.film, egacolor[e.color],
                     (pVertices[e.v1][0], pVertices[e.v1][1]),
                     (pVertices[e.v2][0], pVertices[e.v2][1]))

        # Finally (for debugging purposes) let's draw the normals!
        for i, e in enumerate(self.wf_data.normals):
            if visibleFaces[i]:
                pygame.draw.line(camera.film, egacolor[e.color],
                     (pVertices[e.v1][0], pVertices[e.v1][1]),
                     (pVertices[e.v2][0], pVertices[e.v2][1]))

        # Finally, let's draw the coordinate frame for the camera.
        #fwdAxis = viewport.proj(camera.getFwdCoord())
        #upAxis = viewport.proj(camera.getUpCoord())
        #leftAxis = viewport.proj(camera.getLeftCoord())
        #centerPt = viewport.proj(camera.getCenterCoord())

        # The above axis information ignores an important fact:
        # the axis information is centered at the world center
        # coordinate (0, 0, 0), but it needs to be centered at
        # the center coordinate!  Thus, we need to change it to:
        centerOfAxisVector = camera.getCenterCoord()
        fwdAxisVector = camera.getFwdCoord()*50 + centerOfAxisVector
        upAxisVector = camera.getUpCoord()*50 + centerOfAxisVector
        leftAxisVector = camera.getLeftCoord()*50 + centerOfAxisVector

        # Now we project these four vectors onto the screen.
        centerPt = viewport.proj(centerOfAxisVector)
        fwdAxis = viewport.proj(fwdAxisVector)
        upAxis = viewport.proj(upAxisVector)
        leftAxis = viewport.proj(leftAxisVector)

        pygame.draw.line(camera.film, egacolor['blue'],
              (upAxis.x, upAxis.y), (centerPt.x, centerPt.y))
        pygame.draw.circle(camera.film, egacolor['blue'], (int(centerPt.x), int(centerPt
        pygame.draw.line(camera.film, egacolor['green'],
              (fwdAxis.x, fwdAxis.y), (centerPt.x, centerPt.y))
        pygame.draw.line(camera.film, egacolor['red'],
              (leftAxis.x, leftAxis.y), (centerPt.x, centerPt.y))

        # print "Fwd ", camera.getFwdCoord(), "Up ", camera.getUpCoord(), "Left ", camer

    def __str__(self):
        return movable.__str__(self)
```

"WF_DATA.PY". This reads the data that would be used by a "position" class; unlike my original "wireframe" class, this doesn't inherit from "movable"; hence, a "wf_data" object would have no position whatsoever. It's up to "position" to place it, and then to draw it when the time comes.

```
# This module should contain everything I need to
# create a wireframe object.

# This is the first major step in creating my game
# engine.

# Special note:  For some reason, in
# "wireframe.__init__", the function "re.sub"
# insists on adding '' whenever there's whitespace.
# I don't know if this is a 'feature' or if I'm
# doing something wrong there...but it's nonetheless
# annoying.

from vertex import *
# from movable import *

import re

# Finally, I create the infamous wireframe class!

# This is created as data rather than as an item:
# Thus, it is no longer movable.  Since my worm
# will have lots of segments based on the same
# data, I realized that it would be more memory-efficient
# to separate the data from the position information..
class wf_data(object):
   def __init__(self, filename):
      """The class that represents 3D objects; this version
      only keeps track of vertex data, though:  position data
      is kept in the 'position' class."""
      self.vertices = []
      self.edges = []
      self.faces = []

      # We'll also want to keep track of normals, as edges.
      self.normals = []

      def calcFaceCenterNormal(face):
         """Calculates the Center and Normal for a given face.

         Ideally, this would be a function for the face class;
         unfortunately, the face class needs access to information
         that only a wireframe class will have--namely, access
         to the vertices!"""

         # Now, we'll calculate the center, by taking
         # the average of the vertices.
         face.center = Zero
```

```
        for v in face.vertices:
            face.center += self.vertices[v]
        face.center = face.center/len(face.vertices)

        # Next, we'll find the normal to this face.
        # We're assuming that the vertices of the face are
        # in a clockwise order, viewed from outside
        # the wireframe.
        v0 = self.vertices[face.vertices[0]]
        v1 = self.vertices[face.vertices[1]]
        v2 = self.vertices[face.vertices[2]]

        edgeV0 = v1-v0
        edgeV1 = v2-v1

        face.normal = edgeV0.cross(edgeV1)
        face.normal = face.normal.norm()

        face.normVertex = face.normal*25 + face.center

        # Now we'll store the normals as a part of the wireframe
        i = len(self.vertices)
        self.vertices.append(face.normVertex)
        self.vertices.append(face.center)
        self.normals.append(edge(i, i+1, 'black'))

    # Each stage of processing the data file has a
    # special function that will be referenced in
    # a special loop.
    def vertices(ln):
        'Converts a list to a vertex format for wireframe.'
        v = vector(float(ln[0]), float(ln[1]), float(ln[2]))
        self.vertices.append(v)

    def edges(ln):
        'Converts a list to an edge format for wireframe.'
        e = edge(int(ln[0]), int(ln[1]), ln[2])
        self.edges.append(e)

    def faces(ln):
        'Converts a list to a face format for wireframe.'
        # First, we create the list of vertex indices
        vlist = []
        for n in ln[:-1]:
            vlist.append(int(n))

        # Next, we create the list of edge indices
        length = len(vlist)
        elist = []
        for index, v in enumerate(vlist):
            i = (index+1) % length
```

```
            e = edge(v, vlist[i])
            if e in self.edges:
                elist.append(self.edges.index(e))

        # Then we append the new face to our list!
        # Recall that ln[-1] is the color of the face.
        self.faces.append(face(vlist, elist, ln[-1]))

        # Finally, we calculate the center and normal for
        # our new face!
        calcFaceCenterNormal(self.faces[-1])

    def finished(ln):
        pass

    stage = [vertices, edges, faces, finished]

    wirefile = file(filename)
    i = 0 # This is the stage reference index.
    for eachline in wirefile:
        # First, remove comments that start with '#'
        eachline = re.sub('#.*', '', eachline)
        # Next, we parse the line by white-space
        eachline = re.split('\s+', eachline)
        # since sub insists on putting lots of empty string '' in
        # the lists, we need to add this loop.
        mylist = []
        for item in eachline:
            if item:  mylist.append(item)

        # This part really bugs me:  it's rather "clever", which is
        # to say, it's rather "un-Pythonic", but I can't think of a
        # more straightforward way of doing this without repeating
        # the first part of the loop!

        # To understand what's going on:  if we reach an 'end',
        # we advance to the next stage; stage[] is an array of
        # functions, each function of which adds vertices, edges
        # and faces to the wireframe.  I suppose later I could add
        # other stages (such as bitmaps for faces) without changing
        # this portion.  (I would just have to define a new function,
        # and then add it to the end of the stages array.)
        if mylist:
            # if we reach the end of one stage, go to
            # the next
            if mylist[0] == 'end':
                i+=1
            else:
                # Add vertex, edge or face according to
                # the right stage
                stage[i](mylist)
```

```
    def __str__(self):
        string = ''
        for vtx in self.vertices:
            string += str(vtx) + '\n'
        for edge in self.edges:
            string += str(edge) + '\n'
        for face in self.faces:
            string += str(face) + '\n'
        return string
```

"WORM.PY". Since I would like to draw 3D worms that move around gobbling up numbers, I needed a class that would combine the segments and cameras associated with each worm; this class also includes the functions that changes the direction of the worm or its "fly camera" (a camera meant to move independently of the worm for debugging purposes). Since I had to debug the camera, these functions are still very much works in progress.

```
#  worm.py.

# This is the class that produces the worm I will
# likely use in my worms3D game.
from wf_data import *
from position import *
from camera import *

# First, let's define some directional constants
# that will likely be useful:

CLOCKWISE = 0
COUNTERCLOCKWISE = 1
FORWARD = 2
BACKWARD = 3
UP = 4
LEFT = 5
DOWN = 6
RIGHT = 7

BORDER = 8

# Since worm is a collection of objects, it doesn't have a position; we
# will keep track of our position via the nose.
class worm(object):
    def __init__(self, beta=0, axis=Up, pos=Zero, beta_f=0, axis_f = Up,
                 length = 3, velocity = 100, up = Up, fwd = Fwd, left = Left,
                 sx=Sx, sy=Sy, near=Near, farfov=Far, eye=vector(0, 0, 500), camtype='stand

        self.nose_data = wf_data('worm_head_tail.dat')
        self.middle_data = wf_data('worm_middle_segment.dat')
        self.elbow_data = wf_data('worm_elbow.dat')
```

```
        self.nose = position(self.nose_data, beta, axis, pos, beta_f, axis_f, up, fwd, l
        self.nose.rotate_by_Up(128)
        self.nose.setworld()

        # The Orientation axis of our worm; this is important for figuring out
        # translations and rotations!
        #self.Up = up
        #self.Fwd = fwd
        #self.Left = left

        self.length = length
        self.velocity = velocity

        self.flyVelocity = 10
        #self.flyUp = up
        #self.flyFwd = fwd
        #self.flyLeft = left

        # This is the camera that will follow our worm!
        # Note that the camera position needs appropriate coordinates! based
        # on the initial position of the nose.


        cam_beta = beta
        cam_axis = axis
        cam_pos = pos
        cam_beta_f = beta_f
        cam_axis_f = axis_f
        self.noseCamera = camera(sx, sy, near, farfov,
                 cam_beta, cam_axis, cam_pos, cam_beta_f, cam_axis_f, camtype, fwd, left

        self.flyCamera = camera(sx, sy, near, farfov,
                 cam_beta, cam_axis, cam_pos, cam_beta_f, cam_axis_f, camtype, fwd, left

        self.cameraList = [self.noseCamera, self.flyCamera]
        self.curCamera = 0
        self.camera = self.cameraList[self.curCamera]

        # This is where we'll keep the segments of our worm.
        self.segments = []

    def add_segment(self, direction):
        # We'll use the direction and our current position to determine
        # what pieces we'll add to self.segments.

        # Note that clockwise and counterclockwise will only rotate the camera;
        # although they should also rotate the orientation of the nose (so that
        # UP doesn't become RIGHT)!
        if direction == CLOCKWISE:
            # rotate self.Up, self.Left by the bradian-axis pair (192, self.Fwd)
```

```
        # rotate camera by the bradian-axis pair (192, camera.fwd)
        print "clockwise"
    elif direction == COUNTERCLOCKWISE:
        # rotate self.Up, self.Left by the bradian-axis pair (64, self.Fwd)
        # rotate camera by the bradian-axis pair (64, camera.fwd)
        print "counterclockwise"
    elif direction == FORWARD:
        new_segment = position(self.middle_data, self.nose.beta,
                  self.nose.axis, self.nose.pos,
                  self.nose.beta_f, self.nose.axis_f)
        self.segments.append(new_segment)

        self.nose.add_translation(self.Fwd*self.velocity)
        self.nose.setworld()

        # Note that this works *opposite* of what I would expect!
        # I need to figure out why...
        self.noseCamera.add_translation(-self.Fwd*self.velocity)
        self.noseCamera.setworld()
    elif direction == UP:
        # First, add the elbow
        new_segment = position(self.elbow_data, self.nose.beta,
                  self.nose.axis, self.nose.pos,
                  self.nose.beta_f, self.nose.axis_f)
        new_segment.add_init_rotation(64, self.Fwd)
        new_segment.setworld()
        self.segments.append(new_segment)

        # Rotate the nose and the noseCamera
        self.nose.add_init_rotation(64, self.Left)
        self.noseCamera.add_init_rotation(64, self.Left)
        print self.nose.axis, self.noseCamera.axis

        # Now we'll want to rotate the nose's coordinate system!
        rotate_system = quat(); rotate_system.rotation(64, self.Left)
        rotate_mx = rotate_system.matrix()
        self.Up = rotate_mx * self.Up
        self.Fwd = rotate_mx * self.Fwd
        # self.Left is fixed by this rotation!

        # Now, we'll advance the nose and camera!

        self.nose.add_translation(self.Fwd*self.velocity)
        self.noseCamera.add_translation(-self.Fwd*self.velocity)

        self.nose.setworld()
        self.noseCamera.setworld()
    elif direction == LEFT:
        # First, add the elbow
        new_segment = position(self.elbow_data, self.nose.beta,
                  self.nose.axis, self.nose.pos,
```

```
                self.nose.beta_f, self.nose.axis_f)
        # new_segment.add_init_rotation(64, self.Up)
        new_segment.setworld()
        self.segments.append(new_segment)

        # Rotate the nose and the noseCamera
        self.nose.add_init_rotation(64, self.Up)
        self.noseCamera.add_init_rotation(64, self.Up)
        print self.nose.axis, self.noseCamera.axis

        # Now we'll want to rotate the nose's coordinate system!
        rotate_system = quat(); rotate_system.rotation(64, self.Up)
        rotate_mx = rotate_system.matrix()
        self.Left = rotate_mx * self.Left
        self.Fwd = rotate_mx * self.Fwd
        # self.Left is fixed by this rotation!

        # Now, we'll advance the nose and camera!

        self.nose.add_translation(self.Fwd*self.velocity)
        self.noseCamera.add_translation(-self.Fwd*self.velocity)

        self.nose.setworld()
        self.noseCamera.setworld()
        print "left"
        pass
    elif direction == DOWN:
        print "down"
        pass
    elif direction == RIGHT:
        print "right"
        pass
    elif direction == BACKWARD:
        print "backward"
        pass
    elif direction == BORDER:
        print "border"
        pass

def remove_segment(self):
    # This will remove a segment from the end of the snake, and
    # advance the tail accordingly
    pass

def set_length(self, length):
    self.length = length

def set_velocity(self, velocity):
    self.velocity = velocity

def drawWorm(self):
```

```
        # This is the function that will draw itself; by default it will draw on
        # its own camera.

        # Now, the nose is the front of this thing; the next segment, however, is
        # the last item in this list; and the item before that is there, too.
        #  Thus, we need to reverse it!
        self.segments.reverse()

        # Draw the nose...
        self.nose.sketchOnto(self.camera)
        for segment in self.segments:

            # First, we'll need to check to see if we should draw it...

            # If so, we'll then do this:
            segment.sketchOnto(self.camera)

        # We now restore the list to its original order.
        self.segments.reverse()
        # Now we draw the tail.
        # self.tail.drawOnto(self.camera)

    # For debugging purposes, I wanted a camera I could move independently
    # of the main camera.
    def move_flyCamera(self, direction, button):
        """For debugging purposes, I wanted a camera I could move independently
        of the main camera."""
        if direction == CLOCKWISE:
            self.flyCamera.rotate_by_Fwd(2)
            self.flyCamera.setworld()
        elif direction == COUNTERCLOCKWISE:
            self.flyCamera.rotate_by_Fwd(-2)
            self.flyCamera.setworld()
            print "counterclockwise"
        elif direction == FORWARD:
            if button == 1:
                self.flyCamera.translate_by_Fwd(self.flyVelocity)
                self.flyCamera.setworld()
            elif button == 3:
                self.flyCamera.rotate_by_Fwd(2)
                self.flyCamera.setworld()
                print "clockwise"
        elif direction == BACKWARD:
            if button == 1:
                self.flyCamera.translate_by_Fwd(-self.flyVelocity)
                self.flyCamera.setworld()
            elif button == 3:
                self.flyCamera.rotate_by_Fwd(-2)
                self.flyCamera.setworld()
                print "counterclockwise"
        elif direction == UP:
```

```
            if button == 1:
                self.flyCamera.translate_by_Up(self.flyVelocity)
                self.flyCamera.setworld()
            elif button == 3:
                self.flyCamera.rotate_by_Left(2)
                self.flyCamera.setworld()
        elif direction == LEFT:
            if button == 1:
                self.flyCamera.translate_by_Left(self.flyVelocity)
                self.flyCamera.setworld()
            elif button == 3:
                self.flyCamera.rotate_by_Up(2)
                self.flyCamera.setworld()
        elif direction == DOWN:
            if button == 1:
                self.flyCamera.translate_by_Up(-self.flyVelocity)
                self.flyCamera.setworld()
            elif button == 3:
                self.flyCamera.rotate_by_Left(-2)
                self.flyCamera.setworld()
        elif direction == RIGHT:
            if button == 1:
                self.flyCamera.translate_by_Left(-self.flyVelocity)
                self.flyCamera.setworld()
            elif button == 3:
                self.flyCamera.rotate_by_Up(-2)
                self.flyCamera.setworld()

    def cycleCamera(self):
        self.curCamera = (self.curCamera+1)%len(self.cameraList)
        self.camera = self.cameraList[self.curCamera]

    def pop(self):
        """The sole purpose of this function is to check whether or not
        the wireframe data given to wf_data is really independent of the position
        data given to each wf_data instance.  If it is, then this should cause
        the vertex to "pop out" for all the segments drawn; otherwise, it
        would have no effect at all.

        The wireframe data *is* independent, as desired!"""
        self.middle_data.vertices[0] = vector(100, 100, 100)
```

"WORMSEGMENTS.PY" The main program that draws and controls the worm. This file draws the cursor and the "cursor crosshairs", and reads movements and clicks from the mouse, (and to a lesser extent, the keyboard), and then moves the camera appropriately. The worm's "nose camera" moves click-by-click (I'm not yet to the point where I want to move forward automatically), while the "fly camera" moves in a continuous manner.

```
#!/usr/bin/python2.5
```

```
# In this Python program, I test my "camera" by
# drawing a coordinate axis and several rotating
# cubes.

# As much as I wanted to use PyAllegro, I had to move
# to PyGame instead; alpy just needs too much work, is
# too unstable, and is too slow in making progress.

import sys
import pygame

# This module imports all the little modules I will
# need for this program.
import math3d
from vertex import egacolor
from math import sqrt
from worm import *

# First, we need to initialize the graphics system.
pygame.init()

# size = width, height = 320, 400
size = width, height = 625, 400
xoffset = 50
yoffset = 50
offset = xoffset/2, yoffset/2
screen = pygame.display.set_mode(size)

black = 0, 0, 0
white = 255, 255, 255
bgcolor = white

## For visual reference, we will also draw a
## coordinate axis.
coords = math3d.wireframe('coords.dat')

## This is where I will be experimenting with movement...
nextscreen = 0
pygame.mouse.set_visible(False)
center_x = width/2; center_y = height/2
inner_r = 50*50; outer_r = 100*100
m_0 = (float(height)/width); m_1 =-(float(height)/width)

clockwise_image = \
         pygame.image.load('../images/arrow_rotate_clockwise.png')
counter_clockwise_image = \
         pygame.image.load('../images/arrow_rotate_anticlockwise.png')
fwd_image = pygame.image.load('../images/add.png')
back_image = pygame.image.load('../images/money_yen.png')

up_image = pygame.image.load('../images/arrow_up.png')
```

```
left_image = pygame.image.load('../images/arrow_left.png')
right_image = pygame.image.load('../images/arrow_right.png')
down_image = pygame.image.load('../images/arrow_down.png')

cursor_image = up_image

dest_offset = up_image.get_size()
dest_offset = (dest_offset[0]/2, dest_offset[1]/2)
#m_offset = (0, 0)
show_cursor = False

hyp = sqrt(height*height + width*width)
line_y = 100 * height / hyp
line_x = 100 * width / hyp

def direction(pos):
   m_x = pos[0]
   m_y = pos[1]
   cir_x = pos[0] - center_x
   cir_y = pos[1] - center_y
   line_0 = m_0*m_x
   line_1 = m_1*m_x + height
   circle = cir_x * cir_x + cir_y * cir_y

   m_offset = (m_x - dest_offset[0], m_y - dest_offset[1])
   # print m_offset

   if m_x < offset[0] + dest_offset[0] \
         or m_x > width - offset[0] - dest_offset[0] \
         or m_y < offset[1] + dest_offset[1] \
         or m_y > height - offset[1] - dest_offset[1]:
      return BORDER, m_offset
   else:
      if circle < inner_r:
         if m_y < center_y:
            return FORWARD, m_offset
         else:
            return BACKWARD, m_offset
      elif circle < outer_r:
         if m_y > center_y:
            return CLOCKWISE, m_offset
         else:
            return COUNTERCLOCKWISE, m_offset
      elif m_y < line_0 and m_y < line_1:
         return UP, m_offset
      elif m_y > line_0 and m_y < line_1:
         return LEFT, m_offset
      elif m_y > line_0 and m_y > line_1:
         return DOWN, m_offset
      elif m_y < line_0 and m_y > line_1:
         return RIGHT, m_offset
```

```
def cursor(direction):
    if direction == CLOCKWISE:
        return clockwise_image
    elif direction == COUNTERCLOCKWISE:
        return counter_clockwise_image
    elif direction == FORWARD:
        return fwd_image
    elif direction == BACKWARD:
        return back_image
    elif direction == UP:
        return up_image
    elif direction == LEFT:
        return left_image
    elif direction == DOWN:
        return down_image
    elif direction == RIGHT:
        return right_image

def drawScene():
    my_worm.camera.film.fill(bgcolor)
    my_worm.camera.draw_wf(coords)

    my_worm.drawWorm()
    my_worm.camera.drawOnto(screen, offset)

    # This is debug info; it prints the positions of noseCam and flyCam
    nosePos = 'noseCamera:  ' + str(my_worm.noseCamera.beta)  \
              + ' ~ ' + str(my_worm.noseCamera.axis)
    noseTxt = font.render(nosePos,
              False, egacolor['white'], egacolor['black'])
    screen.blit(noseTxt, (10, 10))
    nosePos = str(my_worm.noseCamera.pos)
    noseTxt = font.render(nosePos, False,
              egacolor['white'], egacolor['black'])
    screen.blit(noseTxt, (10, 30))
    nosePos = str(my_worm.noseCamera.beta_f) \
              + ' ~ ' + str(my_worm.noseCamera.axis_f)
    noseTxt = font.render(nosePos, False,
              egacolor['white'], egacolor['black'])
    screen.blit(noseTxt, (10, 50))

    flyPos = 'flyCamera:  ' + str(my_worm.flyCamera.beta) \
              + ' ~ ' + str(my_worm.flyCamera.axis)
    flyTxt = font.render(flyPos, False,
              egacolor['white'], egacolor['black'])
    screen.blit(flyTxt, (10, 70))
    flyPos = str(my_worm.flyCamera.pos)
    flyTxt = font.render(flyPos, False,
              egacolor['white'], egacolor['black'])
    screen.blit(flyTxt, (10, 90))
```

```
    flyPos = str(my_worm.flyCamera.beta_f) \
             + ' ~ ' + str(my_worm.flyCamera.axis_f)
    flyTxt = font.render(flyPos, False,
             egacolor['white'], egacolor['black'])
    screen.blit(flyTxt, (10, 110))

    if show_cursor:
       screen.blit(cursor_image, m_offset)

    pygame.draw.circle(screen, egacolor['black'],
             (center_x, center_y), 50, 1)
    pygame.draw.circle(screen, egacolor['black'],
             (center_x, center_y), 100, 1)
    pygame.draw.line(screen, egacolor['black'],
             (center_x-100, center_y), (center_x-50, center_y), 1)
    pygame.draw.line(screen, egacolor['black'],
             (center_x+50, center_y), (center_x+100, center_y), 1)

    pygame.draw.line(screen, egacolor['black'], offset,
             (center_x-line_x, center_y-line_y), 1)
    pygame.draw.line(screen, egacolor['black'],
             (center_x+line_x, center_y+line_y),
             (width-offset[0], height-offset[1]), 1)
    pygame.draw.line(screen, egacolor['black'],
             (offset[0], height-offset[1]),
             (center_x-line_x, center_y+line_y), 1)
    pygame.draw.line(screen, egacolor['black'],
             (center_x+line_x, center_y-line_y),
             (width-offset[0], offset[1]), 1)

    pygame.display.flip()

font = pygame.font.Font(None, 24)
my_worm = worm(sx=width-xoffset, sy=height-yoffset)
useFlyCamera = False
while 1:
    events = pygame.event.get()
    for event in events:
       if event.type == pygame.QUIT:
          sys.exit()
       elif event.type == pygame.KEYDOWN:
          # print event
          print event
          if event.unicode == 'f' or event.unicode == 'F':
             print 'Toggle useFlyCamera!!!'
             if useFlyCamera:
                useFlyCamera = False
             else:
                useFlyCamera = True
             my_worm.cycleCamera()
          elif event.unicode == 'n' or event.unicode == 'N':
```

```
                nextscreen = 1
            elif event.unicode == 'p' or event.unicode == 'P':
                my_worm.pop()
        elif event.type == pygame.MOUSEMOTION:
            # print event.dict
            # print event.pos, event.buttons
            mouse_dir, m_offset = direction(event.pos)
            if mouse_dir == BORDER:
                show_cursor = False
            else:
                show_cursor = True
                cursor_image = cursor(mouse_dir)
                # print event.pos, event.button
        #elif event.type == pygame.MOUSEBUTTONDOWN:
            # print event.dict
        elif  event.type == pygame.MOUSEBUTTONDOWN:
            if useFlyCamera:
                print "hello!"
                # i = 0
                button_down = True
                while button_down:
                    # print i,
                    # i = (i+1)%256
                    next_direction, m_offset = direction(event.pos)
                    my_worm.move_flyCamera(next_direction, event.button)
                    for up_event in pygame.event.get():
                        if up_event.type == pygame.MOUSEBUTTONUP:
                            button_down = False
                    drawScene()
            else:
                next_direction, m_offset = direction(event.pos)
                my_worm.add_segment(next_direction)
                # print event.pos, event.button
    drawScene()

    if nextscreen == 1:
        break

pygame.mouse.set_visible(True)
```

A NOTE ON "CAMERA.PY". Although I have made extensive changes to my camera, I have chosen not to include it here; nonetheless, I wanted to make a brief comment on this file. The changes to my camera came mostly as a result of discovering that I had reversed my x-axis and my z-axis in my definitions in "vertex.py". Thus, most of the changes to this class were the result of adjusting constants and removing negative signs. I also moved Fwd, Left and Up vectors to my "movable" class, so that I could use them for keeping track of orientation of objects in general.

I probably still have work to do to (especially with the functions) to make sure I could do everything with my camera that I would hope to do!

"WORM_HEAD_TAIL.DAT". To implement my worm, I need a head (which could be a tail, too), a middle segment, and an elbow. This is one of the files I use to create my worm; for incompleteness, I have chosen to leave the other two out of this document.

```
# Worm Head or Tail Segment Wireframe file

# First we list the vertices, one per line.
# Note that the order of these vertices are crucial!
-25 -50 -50      # v0
 25 -50 -50
 50 -25 -50
 50  25 -50      # v3
 25  50 -50      # v4
-25  50 -50
-50  25 -50
-50 -25 -50      # v7

-25 -50  25      # v8
 25 -50  25
 50 -25  25
 50  25  25      # v11
 25  50  25      # v12
-25  50  25
-50  25  25
-50 -25  25      # v15

-25 -25  50      # v16
 25 -25  50
 25  25  50
-25  25  50      # v19
end vertices

# Now we can add our edges:
 0  1                    blue
 1  2                    blue
 2  3                    blue
 3  4                    blue
 4  5                    blue
 5  6                    blue
 6  7                    blue
 7  0                    blue             # Front end

 8   9          blue
 9  10          blue
10  11          blue
11  12          blue
12  13          blue
13  14          blue
14  15          blue
15   8          blue            # Back end
```

```
 0  8                    blue
 1  9                    blue
 2 10            blue
 3 11            blue
 4 12            blue
 5 13            blue
 6 14            blue
 7 15            blue            # All the sides

16 17            blue
17 18            blue
18 19            blue
19 16            blue
15 16            blue
 8 16            blue
 9 17            blue
10 17            blue
11 18            blue
12 18            blue
13 19            blue
14 19            blue            # The nose of the worm
end edges

# This segment has only side faces.
# Note that some day I might want to add "normals"
 0  1  9  8              lightblue
 1  2 10  9      lightblue
 2  3 11 10      lightblue
 3  4 12 11      lightblue
 4  5 13 12      lightblue
 5  6 14 13      lightblue
 6  7 15 14      lightblue
 7  0  8 15      lightblue       # The sides of the worm

15  8 16                 lightblue
 8  9 17 16      lightblue
 9 10 17                 lightblue
10 11 18 17      lightblue
11 12 18                 lightblue
12 13 19 18      lightblue
13 14 19                 lightblue
14 15 16 19      lightblue
16 17 18 19      lightblue
end faces
```