Making things move

Time and Space
Animation

• Persistence of Vision/ Illusion of Motion

  • showing a human a sequence of still images in rapid succession is perceived as motion.

  • screen refresh rate: how often the screen contents are updated (from frame buffer memory)

  • NTSC (TV) video is 30 frames per second (29.97), film: 24fps

  • more is better.

  • fusion frequency = ~20Hz
x = 0.

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            exit()

    screen.blit(background, (0, 0))
    screen.blit(sprite, (x, 100))
    x += 10.

    # If the image goes off the end of the screen, move it back
    if x > 640.:
        x = 0.

    pygame.display.update()
Time-based motion

- on different machines of differing capabilities, the time it takes to render a frame can differ
- want motion to be a function of time not frame rate
- can use pygame Clock object
  \[
  \text{clock} = \text{pygame.time.Clock()}
  \]
- call \text{clock.tick()} once per frame
- returns time since last call to \text{tick()}, in milliseconds
  \[
  \text{time\_passed} = \text{clock.tick()}
  \]
- call with an argument to force a constant frame rate (will wait until given number of msecs have elapsed since last call to \text{tick()})
  \[
  \text{time\_passed} = \text{clock.tick(30)} \quad \# \text{30fps}
  \]
- divide return value by 1000. to convert to seconds
  \[
  \text{time\_in\_seconds} = \text{time\_passed} / 1000.
  \]
# X coordinate of our sprite
x = 0.

# Speed in pixels per second
speed = 250.

while True:

    for event in pygame.event.get():
        if event.type == QUIT:
            exit()

    screen.blit(background, (0, 0))
screen.blit(sprite, (x, 100))

    time_passed = clock.tick()
time_passed_seconds = time_passed / 1000.0

distance_moved = time_passed_seconds * speed
x += distance_moved

    # If the image goes off the end of the screen, move it back
if x > 640.:
    x = 0.

pygame.display.update()
framereatecompare.py
 ```python
x, y = 100., 100.
speed_x, speed_y = 133., 170.

while True:
    for event in pygame.event.get():
        if event.type == QUIT:
            exit()

    screen.blit(background, (0,0))
screen.blit(sprite, (x, y))

time_passed = clock.tick(30)
time_passed_seconds = time_passed / 1000.0

    x += speed_x * time_passed_seconds
    y += speed_y * time_passed_seconds

    # If the sprite goes off the edge of the screen,
    # make it move in the opposite direction
    if x > 640 - sprite.get_width():
        speed_x = -speed_x
        x = 640 - sprite.get_width()
    elif x < 0:
        speed_x = -speed_x
        x = 0.

    if y > 480 - sprite.get_height():
        speed_y = -speed_y
        y = 480 - sprite.get_height()
    elif y < 0:
        speed_y = -speed_y
        y = 0

    pygame.display.update()
```
Vectors

- Point = x and y coords of a point
- Vector = x and y coords of a change in position
- has a direction and a magnitude (length)

Creating Vectors

You can calculate a vector from any two points by subtracting the values in the first point from the second. Let's demonstrate with an example from a fictional game. The player character—a cybernetically enhanced soldier from the future named Alpha—has to destroy a Beta class sentry droid with a sniper rifle. Alpha is hiding behind a bush at coordinate A (10, 20) and aiming at Beta at coordinate B (30, 35). To calculate a vector AB to the target, Alpha has to subtract the components of B from A. So vector AB is (30, 35) – (10, 20), which is (20, 15). This tells us that to get from A to B we would have to go 20 units in the x direction and 15 units in the y direction (see Figure 5-1). The game would need this information in order to animate a projectile weapon or draw a laser beam between the two points.

Creating a vector:

Figure 5-1. Creating a vector

Storing Vectors

There is no built-in vector type in Python, but you can store a vector in a tuple or list of two values, or you can define a vector class. Defining a class is probably the best option because you can refer to the components by name (x or y) rather than as an index ([0] or [1]). Listing 5-5 demonstrates how we might begin defining a vector class. I called it Vector2 because vectors are also used in 3D games and we may want to have a 3D version of the vector class called Vector3. In addition to the constructor there is a __str__ method, which turns a Vector2 object into a string when it is printed; without it, we would have to print each component individually.

Listing 5-5. Simple Vector Definition

class Vector2(object):
    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y
    def __str__(self):
        return "(%s, %s)"%(self.x, self.y)
import math

class Vector2(object):
    def __init__(self, x=0.0, y=0.0):
        self.x = x
        self.y = y
    def __str__(self):
        return "(%s, %s)"%(self.x, self.y)
@classmethod
    def from_points(cls, P1, P2):
        return Vector2(cls, P2[0] - P1[0], P2[1] - P1[1])
    def get_magnitude(self):
        return math.sqrt( self.x**2 + self.y**2 )

A = (10.0, 20.0)
B = (30.0, 35.0)
AB = Vector2.from_points(A, B)
print AB
print AB.get_magnitude()
Unit Vectors

• directions are usually described with vectors of magnitude of 1.0

• normalize a vector == keep its direction the same, but change its length to 1.0
  
  • divide x and y components by the magnitude

```python
def normalize(self):
    magnitude = self.get_magnitude()
    self.x /= magnitude
    self.y /= magnitude
```
Vector Addition

• add components separately: (for Vector2 class:)

```python
# rhs stands for Right Hand Side
def __add__(self, rhs):
    return Vector2(self.x + rhs.x, self.y + rhs.y)
```

• same effect as the two vectors in sequence: head-to-tail

```
A = (10.0, 20.0)
B = (30.0, 35.0)
AB = Vector2.from_points(A, B)
print "Vector AB is", AB
print "Magnitude of Vector AB is", AB.get_magnitude()
AB.normalize()
print "Vector AB normalized is", AB
```

Executing this script produces the following output:

```
Vector AB is (20.0, 15.0)
Magnitude of Vector AB is 25.0
Vector AB normalized is (0.8, 0.6)
```
Vector Subtraction

- result vector for $A - B$:
  - head at $A$
  - tail at $B$

- component-wise subtraction

```python
def __sub__(self, rhs):
    return Vector2(self.x - rhs.x, self.y - rhs.y)
```
Vector Negation

• flips direction
• magnitude the same

def __neg__(self):
    return Vector2(-self.x, -self.y)
Scalar Multiplication and Division

- multiply (divide) each component by the scalar

  ```python
def __mul__(self, scalar):
    return Vector2(self.x * scalar, self.y * scalar)

def __div__(self, scalar):
    return Vector2(self.x / scalar, self.y / scalar)
```

- keeps direction, changes magnitude
- multiply (divide) by 1.0 == no change
- multiply (divide) by -1.0 == negate
- multiply by value > 1.0 == increase magnitude
- multiply by value < 1.0 == decrease magnitude
Using Vectors for Motion

- linear interpolation
- def positionAt
numpy for vectors

• numpy arrays act like vectors
  • addition, subtraction, scalar mult, etc.
  • also dot(), cross()
  • Use numpy.linalg.norm() to normalize (BE SURE IT HOLDS FLOATS!)

```python
def normalize(v):
    len = math.sqrt(v[0]*v[0] + v[1]*v[1])
    v[0] /= len
    v[1] /= len

A = numpy.array([10., 5.])
print A
print numpy.linalg.norm(A)
print A * 2.
print 2. * A
```
other ways to vectors

• roll your own

• mcGugan's "gameobjects"
  • http://code.google.com/p/gameobjects/

• Vector2 ala' textbook, lots more...but does it all work with latest version!?!?
vectormovement.py