

## Moving a along a line

- consider the problem of making a barrels appear to roll across a plank
  - this is complicated by the issue of the ramp gradient



## Bresenham's line algorithm

- fortunately Bresenham discovered an algorithm which given two points
  - determines the elements of a 2-dimensional grid that should be selected to best approximate the line
- Bresenham's line algorithm also uses integer arithmetic which adds to its complexity

$$y = mx + c$$

- returning to the problem of making a barrel roll down a plank
  - we know the x position, but we need to compute the y value
  
- we know the start and end points of the ramp



$$y = mx + c$$

- in the previous slide the start position is (1, 2) and the end position is (5, 4)
- the dx value is  $5 - 1 = 4$
- the dy value is  $4 - 2 = 2$
- therefore our gradient  $m$  is  $\frac{dy}{dx}$

$$y = mx + c$$

- we need to calculate  $c$
- we know the point  $(1, 2)$  exists on the line
- using  $y = mx + c$
- $2 = 1m + c$
- $2 = \frac{1}{2} + c$
- $c = 2 - \frac{1}{2} = 1 + \frac{1}{2}$

$$y = mx + c$$

- we could use this formula to calculate the  $y$  value given an  $x$  value

- $m = \frac{2}{4} = \frac{1}{2}$

$$y = mx + c$$



x	y
1	2
2	2.5
3	3
4	3.5
5	4

$$y = mx + c$$

- notice how we need floating point values to compute it
  - also notice how we calculated the gradient
- Bresenham's algorithm hunts for the correct gradient by using integer arithmetic and by manipulating the numerator and denominator of the fractional value of  $m$



# Bresenham's algorithm in Python (version 1)



**bres0.py**

```
#!/usr/bin/env python3

def points (p0, p1):
    x0, y0 = p0
    x1, y1 = p1

    dx = abs(x1-x0)
    dy = abs(y1-y0)
    if x0 < x1:
        sx = 1
    else:
        sx = -1

    if y0 < y1:
        sy = 1
    else:
        sy = -1
    err = dx-dy
```

# Bresenham's algorithm in Python (version 1)



bres0.py

```
while True:
    print x0, y0
    if x0 == x1 and y0 == y1:
        return

    e2 = 2*err
    if e2 > -dy:
        # overshoot in the y direction
        err = err - dy
        x0 = x0 + sx
    if e2 < dx:
        # overshoot in the x direction
        err = err + dx
        y0 = y0 + sy
```

# Bresenham's algorithm in Python (version 1)

**bres0.py**

```
#  
# test code  
#  
points ([1, 2], [5, 4])
```

when we run this we see:

```
$ python3 bres0.py  
1 2  
2 2  
3 3  
4 3  
5 4
```

## Using Bresenham's algorithm to give us the next point

- we need to change the algorithm very slightly so that we can get the next point on demand
  - really a cosmetic change
- probably best to use a class and a few methods

# Python implementation of Bresenham's algorithm (version 2)



bres1.py

```
#!/usr/bin/env python3

class bres:
    def __init__ (self, p0, p1):
        self.p0 = p0
        self.p1 = p1
        self.x0 = p0[0]
        self.y0 = p0[1]
        self.x1 = p1[0]
        self.y1 = p1[1]
        self.dx = abs(self.x1-self.x0)
        self.dy = abs(self.y1-self.y0)
```

# Python implementation of Bresenham's algorithm (version 2)



**bres1.py**

```
if self.x0 < self.x1:
    self.sx = 1
else:
    self.sx = -1

if self.y0 < self.y1:
    self.sy = 1
else:
    self.sy = -1
self.err = self.dx-self.dy
```

# Python implementation of Bresenham's algorithm (version 2)



**bres1.py**

```
def get_next (self):  
    if self.x0 == self.x1 and self.y0 == self.y1:  
        return [self.x1, self.y1]  
  
    self.e2 = 2*self.err  
    if self.e2 > -self.dy:  
        self.err = self.err - self.dy  
        self.x0 = self.x0 + self.sx  
    if self.e2 < self.dx:  
        self.err = self.err + self.dx  
        self.y0 = self.y0 + self.sy  
    return [self.x0, self.y0]
```

# Python implementation of Bresenham's algorithm (version 2)



**bres1.py**

```
#
# test code
#
w = bres ([1, 2], [5, 4])
p = w.get_next ()
print p
while p != [5, 4]:
    p = w.get_next ()
    print p
print p
```



# Python implementation of Bresenham's algorithm (version 2)



```
$ python3 bres1.py  
[2, 2]  
[3, 3]  
[4, 3]  
[5, 4]  
[5, 4]
```

## Tidying up the Python interface

- as can be seen in the test code the use of the Python class is ugly
- we can improve this by introducing a method `finished()` which returns `True` if we have reached the end of the line
- we also note there is a slight bug in version 2 as it omits the initial point!

# Python implementation of Bresenham's algorithm (version 3)



**bres.py**

```
#!/usr/bin/env python3

class bres:
    def __init__ (self, p0, p1):
        self.initial = True
        self.end = False
        self.p0 = p0
        self.p1 = p1
        self.x0 = p0[0]
        self.y0 = p0[1]
        self.x1 = p1[0]
        self.y1 = p1[1]
        self.dx = abs(self.x1-self.x0)
        self.dy = abs(self.y1-self.y0)
```

# Python implementation of Bresenham's algorithm (version 3)



**bres.py**

```
if self.x0 < self.x1:
    self.sx = 1
else:
    self.sx = -1

if self.y0 < self.y1:
    self.sy = 1
else:
    self.sy = -1
self.err = self.dx-self.dy
```

# Python implementation of Bresenham's algorithm (version 3)



**bres.py**

```
def get_next (self):  
    if self.initial:  
        self.initial = False  
        return [self.x0, self.y0]  
  
    if self.x0 == self.x1 and self.y0 == self.y1:  
        self.end = True  
        return [self.x1, self.y1]
```

# Python implementation of Bresenham's algorithm (version 3)



**bres.py**

```
self.e2 = 2*self.err
if self.e2 > -self.dy:
    self.err = self.err - self.dy
    self.x0 = self.x0 + self.sx
if self.e2 < self.dx:
    self.err = self.err + self.dx
    self.y0 = self.y0 + self.sy
return [self.x0, self.y0]

def get_current_pos (self):
    return [self.x0, self.y0]

def finished (self):
    return self.end
```

# Python implementation of Bresenham's algorithm (version 3)



**bres.py**

```
#  
# test code  
#  
w = bres ([1, 2], [5, 4])  
while not w.finished ():  
    p = w.get_next ()  
    print p
```

## Test run of our new Python module bres.py



```
$ python3 bres.py  
[1, 2]  
[2, 2]  
[3, 3]  
[4, 3]  
[5, 4]  
[5, 4]
```



## Homework and tutorial work

- comment this code
- use this module `bres.py` together with the `explosions.py` code we looked at last week to generate explosions and track the mouse
- declare a `missile` class which can be given two points: `start`, `end` to traverse along a chosen path
- introduce an `update` method for the `missile` class which moves the missile onto the next coordinate
- tie these pieces together and allow the mouse to fire off a missile from the left, middle, and right silos