Introduction To Game Programming

PyCon US 2013

Richard Jones

Today's tutorial

- Who am I?
- Some initial thoughts
- Constructing a game
- Other game types
- Packaging
- Content tools
- Where now?

Who am I?

Who am I?

- First computer was Commodore 64
- Played lots of games on it
- ... and I wrote games

Who am I?

- Then I got an Amiga (500, then 1200)
- Played lots of games on those
- ... and I wrote games

Who am I?

- Then I got a job, and didn't write games much but I did play them a lot
- Dabbled in making some web games
- Until 2004 or so, when I discovered the joy of game programming with Python

Who am I?

- I also run PyPI (the Cheese Shop)
- I wrote Roundup and a bunch of other stuff
- I run PyWeek

But First

We’re going to cover a bunch of ground today. You are being given the whole source to a simple game. Do what you will to it. Experiment, break it, whatever. Because once I was just like you, with almost no idea how to write a game.

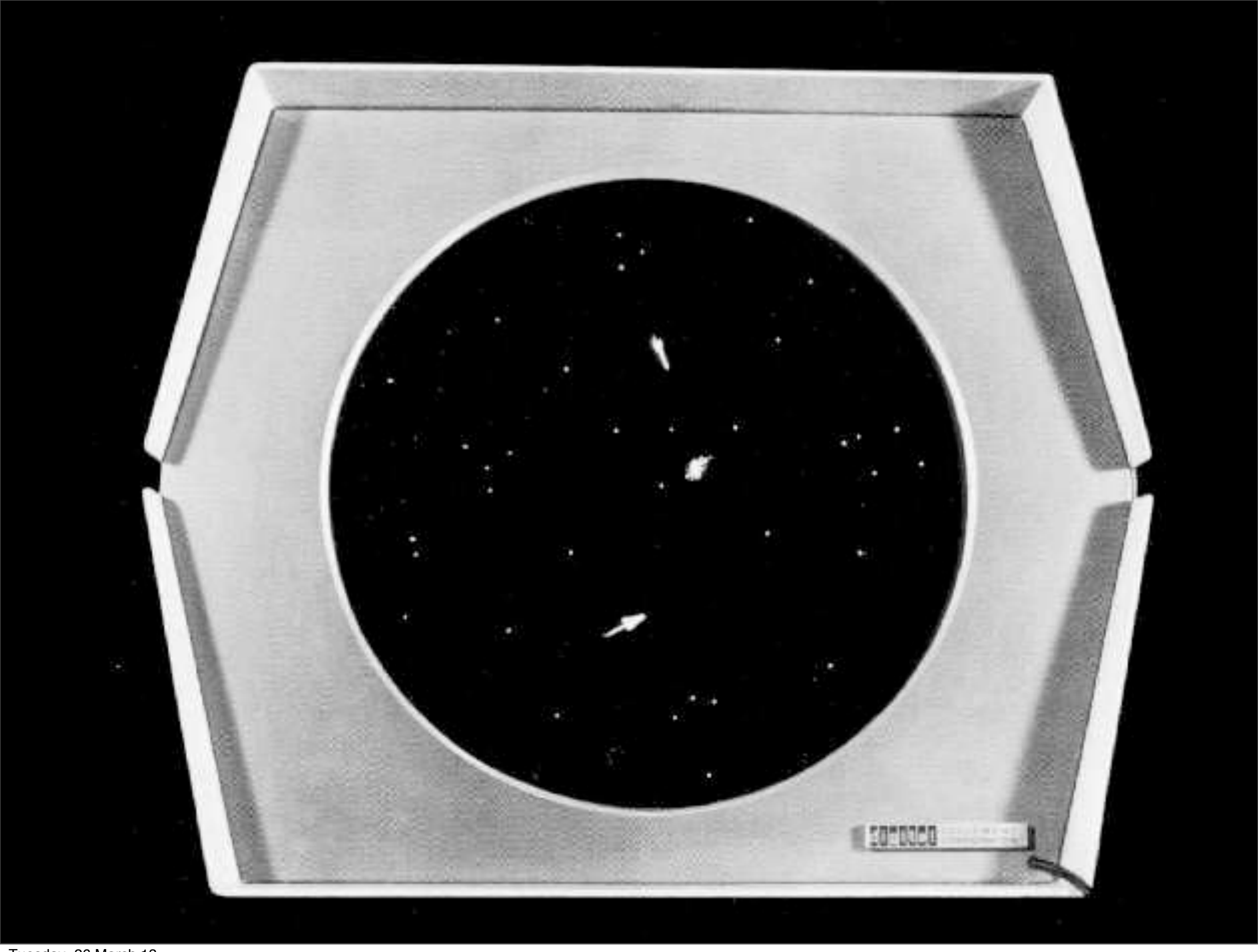
What to create?

I'd like to have you spend a few minutes thinking about what sort of game you might create. This will help you focus your attention today on important aspects of what I'm talking about; but it might also prompt questions that would be best asked today.

Genre

Tuesday, 26 March 13

The shoot-em-up. A staple of video gaming. You move your little avatar around and shoot at others while avoiding their bullets and other hazards.

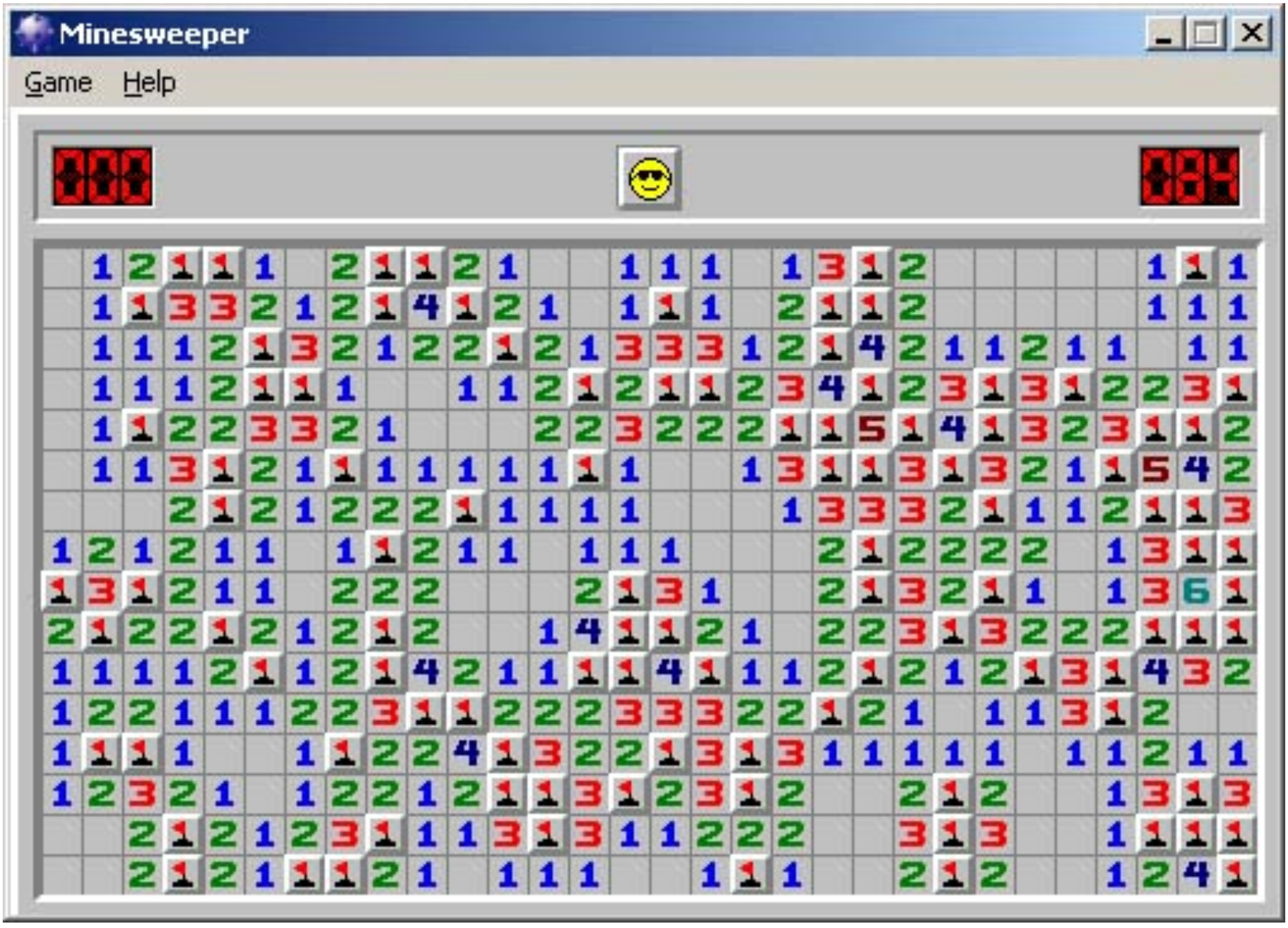


This is one of the earliest known video games; Spacewar! It's a shoot-em-up.

Beat-em-up. Kinda like the shoot-em-up but more constrained. Usually emphasise rock-paper-scissors style attack and defence with memory-game combinations for extra power.

Tuesday, 26 March 13

Platformer. Basically the action game while walking and jumping and swinging and rolling and jetpacking and whatevering around.



Tuesday, 26 March 13

Puzzle games are more about contemplation and finding tricky solutions. A lot of games include puzzles in them.

Adventure games give the player a world to explore. They might also be given things to do that change the world. Typically there are puzzles to solve. Then there's action-adventure which blends the action genre with the adventure genre.

Tuesday, 26 March 13

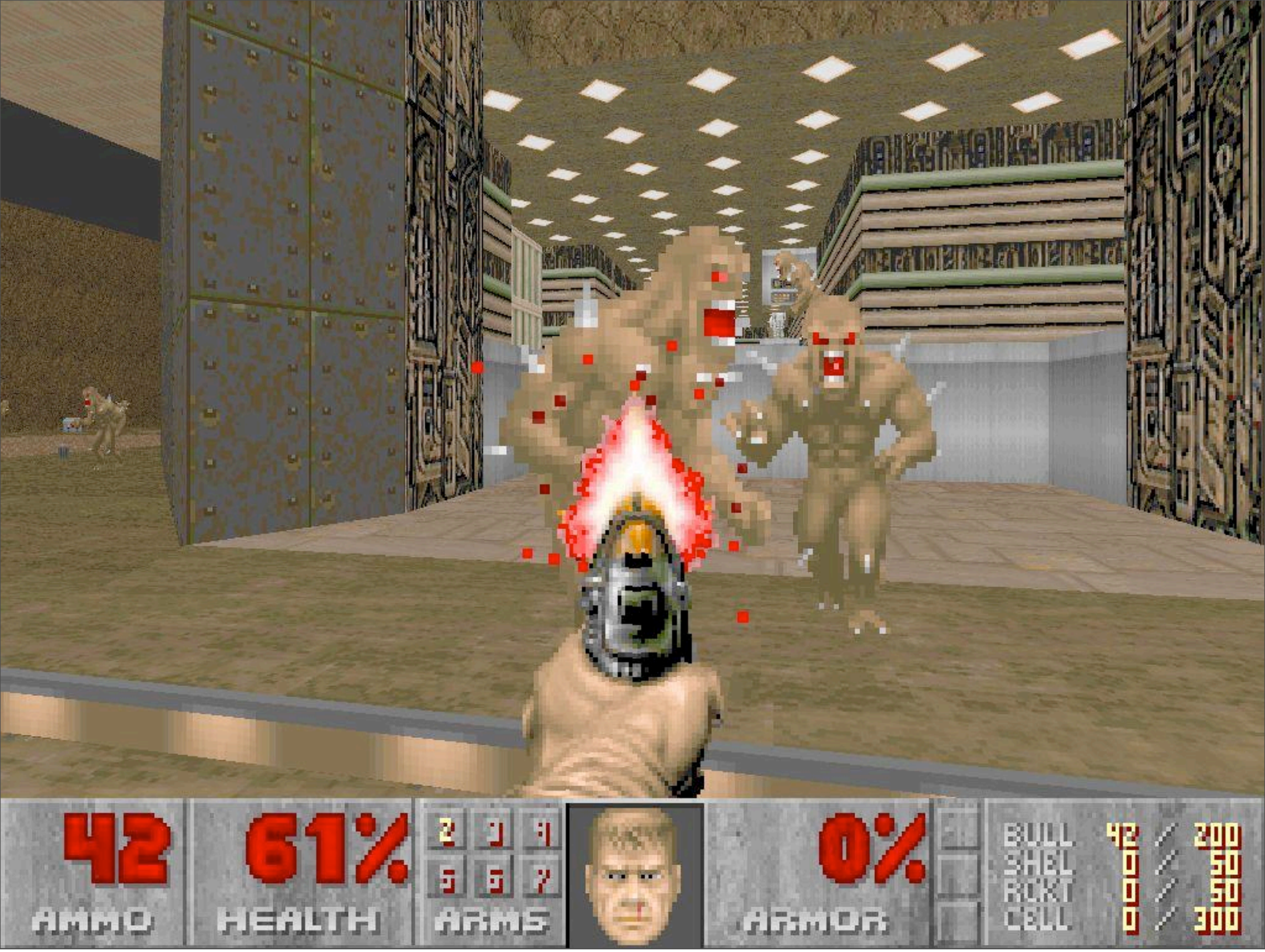
Strategy games rely on the player's decision-making skills, thinking and planning. Often about an empire-level struggle against other empires; they can also be quite abstract like chess or Go. They can be slower, more contemplative turn-based games or real-time games which blend strategy and action. Some strategy games lean strongly towards puzzles - tower defence games are a good example of this.

Tuesday, 26 March 13

Simulation. Focuses on the mechanisms that makes the game world tick. Gives the player pretty direct control over those mechanisms. An emphasis on attempting to realistically model real situations. See also vehicle simulation, a variant of the action genre.

Tuesday, 26 March 13

Role-Playing Game, or RPG. The player's character(s) in the game grow or are customisable through the game. Often the player can change the world in some way. Also, JRPG which follow their own specific formula (though a lot of them are more like simpler action adventure games.)



Tuesday, 26 March 13

FPS. Which is just another way of looking at things, really. The first FPS games were actually RPGs, but with limited freedom compared to Doom. See also "3rd person."

Strategy

Puzzle

Action

Multiplayer

Adventure

Simulation

These are, roughly speaking, six flavours you can mix into a video game.

Final Death

Real-Time

Stealth

New Game+

Nintendo Hard

World Altering

Character Growth

Quicktime Events

Mini Game

Physics

Programming

Turn-Based

Procedural Content

Sandbox

Rock-Paper-Scissors

Auto-Scrolling

Artificial Intelligence

Then there's little extras you can throw in; single mechanics that extend those basics.

Instant Surprise Death

Zombie Closets

Invincible Boss

Limited Save Points

Ice Level

Invisible / Vanishing / Moving Platforms

Goddamned Bats

Artificial Stupidity

Grind

Hidden Passages

No Mercy Invulnerability

Tuesday, 26 March 13

Things to avoid. Ice Level where only you are affected. Bosses that are temporarily invincible or are unkillable (JRPG.) Bats that attack from above/below when you can only shoot sideways. Platforms you can't see or that move suddenly (see also ISD.) AI that's just dumb, esp combined with escort missions.

Setting and Theme

And then there's the setting and general flavour. How the game is rendered is part of this - 2d (top-down, side-on), 2.5d, 3d, 1st person, 3rd person or fixed perspective.

Space Opera

Aliens

Futuristic

Land

Horror

Western

Pirates

Islands

Space

Romance

Steampunk

Medieval

Zombies

Flying

Spies

Superhero

... and beyond all this is the art style you're aiming for.

Start small and be ready to fail

Learning how to design and create video games takes time and practise. Your first game will most likely not be amazing. My first half-dozen certainly were not. This is absolutely fine. Shoot for something simple; focus on a single gameplay mechanic; build that.

Let's construct a game!

The Basics

- Displaying something
- User input
- Gameplay mechanics
- Playing sound effects and music
- Game architecture

These are the basic elements that almost every game has. I'll be covering each of these in the course of this tutorial.

Displaying Something

- Opening a window
- Draw something
- Move it around

So, let's start off with the basics. Images on the screen.

Displaying Something

- Copying an image to the screen
- Drawing pixels using pygame.draw
- Rendering text using pygame.font.Font
- Bypassing all that and using OpenGL

These are the basic methods of displaying something on the screen

Displaying Something

- Load image from some file (PNG, JPG, ...)
- Create an image using pygame.surfarray
- Or draw to a proxy surface using pygame.draw

You can either load your images from a file or construct them in memory.

Opening a Window

```
import pygame

pygame.init()
screen = pygame.display.set_mode((640, 480))
```


01-open-window.py Did you catch sight of that window as it flashed open?
pygame.init() should appear at the very start of any pygame program.
pygame.display.set_mode() takes a bunch of options, but the simplest usage is to just provide a window size. Because there is no further code in our program, it exits immediately after the window is opened.

Main Loop

```
 pygame.init()
 screen = pygame.display.set_mode((640, 480))
```

```
+running = True
+while running:
+    for event in pygame.event.get():
+        if event.type == pygame.QUIT:
+            running = False
+        if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
+            running = False
```


02-main-loop.py
To keep the window alive we need an event loop so the program doesn't quit until we're ready. This is the simplest form of event handling in pygame, and you will rarely need to get more complex.
Explain the two different event types (QUIT and KEYDOWN.)

Structure

```
import pygame

pygame.init()
screen = pygame.display.set_mode((640, 480))

running = True
while running:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False
        if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
            running = False
```

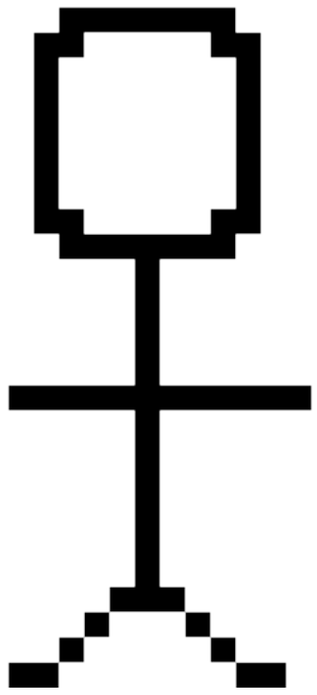

Here we introduce some structure to the code. Where we had this code... 03-structure.py

Structure

```
import pygame

class Game(object):
    def main(self, screen):
        while 1:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    return
                if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                    return

if __name__ == '__main__':
    pygame.init()
    screen = pygame.display.set_mode((640, 480))
    Game().main(screen)
```

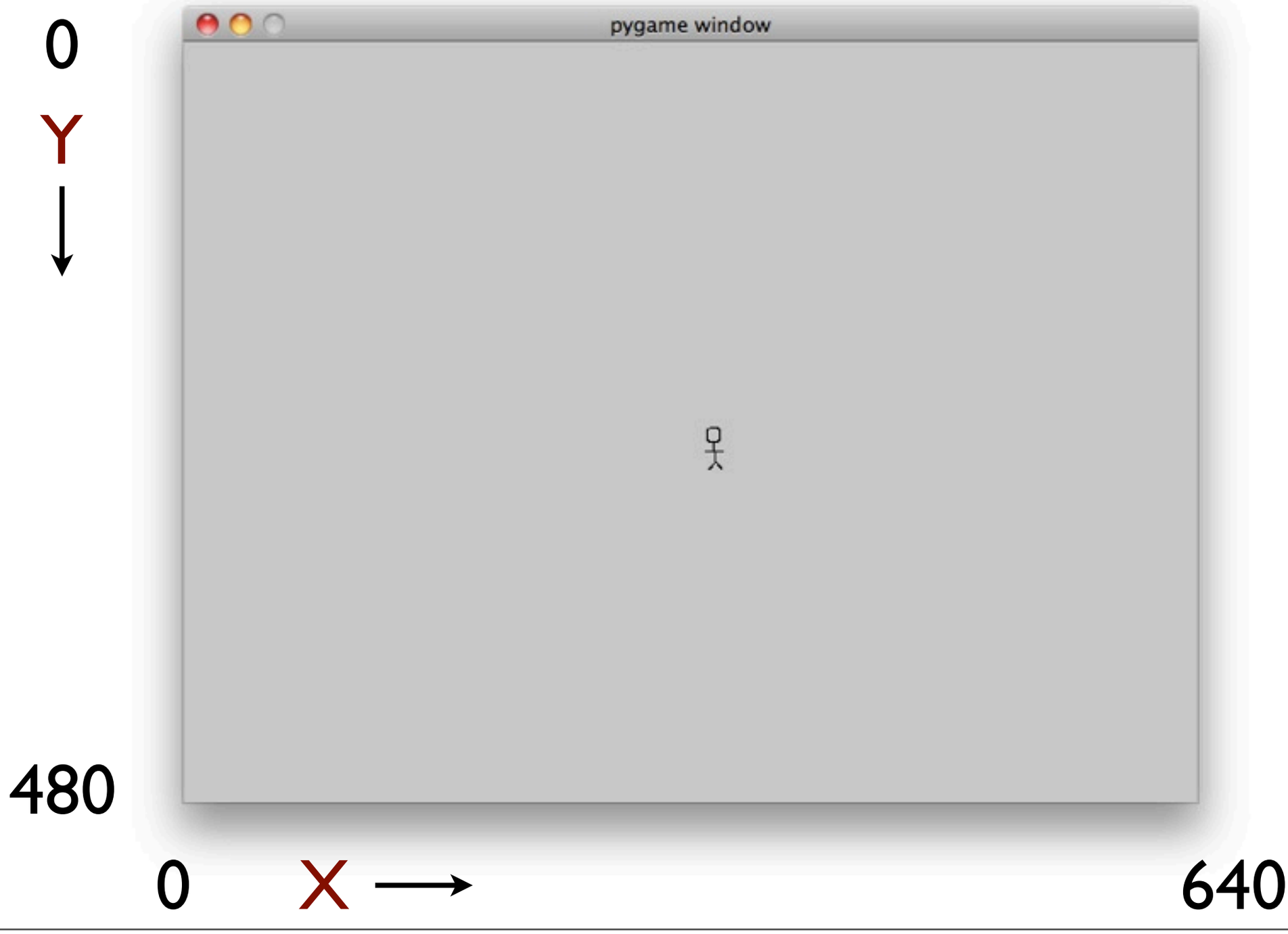

... now we have code that's stand-alone, re-enterable and may contain its own state (which is shareable.) We don't have any yet, but we will. The "game" is a good place to store a reference to the player, playing field, enemies, bullets, etc. Things that various bits of code need access to without resorting to globals. If we want to re-start the game all we have to do is throw out the old game instance, create a new one and off we go.
The "game" could be called "level" or "world" or whatever one chunk of your gameplay is.

Drawing



The final step to getting something basic going is being able to draw images to the screen. Once you've achieved this there's a stack of games that you're able to write (adventure games, tetris, ...)

Coordinate System



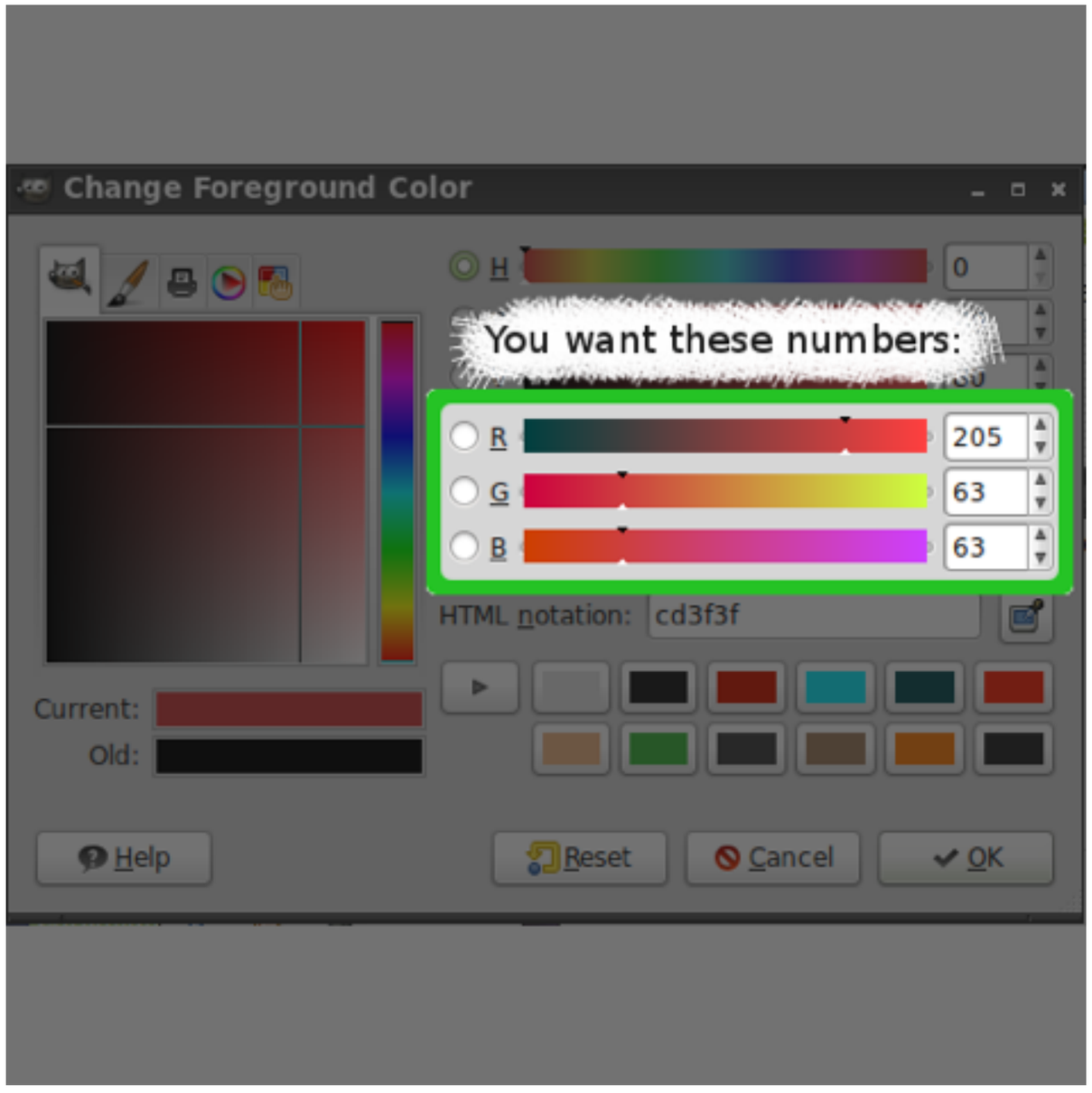
The pygame coordinate system is old-school Y-down because of how screens were refreshed from the early days (serially from top down) and that programs were writing directly to video RAM. The refresh mechanism is the same but OpenGL and other systems use a more sensible Y-up system separated from the VRAM.

Drawing

```
class Game(object):
    def main(self, screen):
+       image = pygame.image.load('player.png')

        while 1:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    return
                if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                    return

+           screen.fill((200, 200, 200))
+           screen.blit(image, (320, 240))
+           pygame.display.flip()
```


04-drawing.py There's four things going on here which I'll cover in turn. First we load an image from disk. It's a PNG but pygame supports most common image formats. If the PNG has transparent bits then they will be honored when drawing the image. Next we clear the screen using a light gray. I'll cover colors on the next slide. Then we're copying (blitting) the image to the screen. Finally flipping the display buffers (we'll come back to that).

Colours in pygame



Colours in pygame are specified in red, green and blue components as a 3-tuple. The values of those components range from 0 (dark) to 255 (full color). You can get those values from most drawing programs.

WARNING: do NOT run the next examples if you've a problem with strobing.
See 04-tearing.py and 04-tearing-fixed.py -- note that pygame uses double-buffering by default so we have to work to make it not use it (by forcing a hardware surface, and not asking for double-buffering). This is also why, after writing your first program, you might be staring at a blank screen.

Screen Buffers

- Drawing directly to the screen is possible
- Screens render in scanlines though (yes, even LCDs - the signal to the LCD is still serial)
- Thus if we draw while the screen refreshes we get “tearing” as you see part old, part new imagery

WARNING: do NOT run the next examples if you’ve a problem with strobing.
See 04-tearing.py and 04-tearing-fixed.py -- note that pygame uses double-buffering by default so we have to work to make it not use it (by forcing a hardware surface, and not asking for double-buffering). This is also why, after writing your first program, you might be staring at a blank screen.

Screen Buffers

- The solution to this is to draw to a buffer (the "drawing" buffer) that's not visible
- When drawing is complete we flip() to ask the display to start showing our new buffer
- The previously-displayed buffer now becomes our drawing buffer

flip() swaps the buffer being used for drawing with the buffer being displayed. Now we're displaying the old draw buffer and drawing on the old display buffer.

Using Less CPU

```
class Game(object):
    def main(self, screen):
+       clock = pygame.time.Clock()

        image = pygame.image.load('player.png')

        while 1:
+           clock.tick(30)

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


Currently our program uses a lot of CPU just looping very very quickly. Here we use a clock to limit the rate we're doing it all to around 30 times (frames) per second. 05-less-cpu.py

Animation

```
        image = pygame.image.load('player.png')
+       image_x = 320
+       image_y = 240

        while 1:
            clock.tick(30)

...
                if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                    return

+               image_x += 10

                screen.fill((200, 200, 200))
-               screen.blit(image, (320, 240))
+               screen.blit(image, (image_x, image_y))
                pygame.display.flip()
```


Now we're drawing something, how about we move it around the screen?
To move the image around the screen we simply make the blitting position a couple of variables. Incrementing the X position once per loop moves it (slowly) across the screen.
06-animation.py
Try moving the fill() to outside the loop so we don't clear the flipped buffer.

User Input

- Getting events
- Distinct keyboard events vs. keyboard state

Events for the keyboard come in two types: key down and key up. Pygame will send you all key down/up events generated. These can be tricky to manage, so there's also a global that keeps track of the pressed state of keys. This means you might miss key presses (eg. impulse actions for "jump") but makes movement handling easier (move while the key is held down.)

User Input

```
            if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                return
```

```
-           image_x += 1
+           key = pygame.key.get_pressed()
+           if key[pygame.K_LEFT]:
+               image_x -= 10
+           if key[pygame.K_RIGHT]:
+               image_x += 10
+           if key[pygame.K_UP]:
+               image_y -= 10
+           if key[pygame.K_DOWN]:
+               image_y += 10
```

```
            screen.fill((200, 200, 200))
            screen.blit(image, (image_x, image_y))
```


So now instead of a fixed increment to the X position we modify the variables in response to key presses. Where previously we detected the ESCAPE key using the KEYDOWN event we're now using pygame.key.get_pressed(). This is so we don't have to manage the up/down state of the keys ourselves. We wouldn't want to miss an ESCAPE impulse though so we still use the KEYDOWN for that. 07-user-input.py

User Input

- Mouse events:
  - MOUSEBUTTONDOWN
  - MOUSEBUTTONUP
  - MOUSEMOTION

Events for the mouse come in three types. All have a Pygame will send you all events generated. These can be tricky to manage, so there's also a global that keeps track of the position of the mouse and the state of its buttons. The buttons are just values 1, 2, 3 for left, middle and right. There are also buttons 4 and 5 for the scroll wheel up and down.

Drawing

```
 import pygame
```

```
+class Player(pygame.sprite.Sprite):
+    def __init__(self, *groups):
+        super(Player, self).__init__(*groups)
+        self.image = pygame.image.load('player.png')
+        self.rect = pygame.rect.Rect((320, 240), self.image.get_size())
+
+    def update(self):
+        key = pygame.key.get_pressed()
+        if key[pygame.K_LEFT]:
+            self.rect.x -= 10
+        if key[pygame.K_RIGHT]:
+            self.rect.x += 10
+        if key[pygame.K_UP]:
+            self.rect.y -= 10
+        if key[pygame.K_DOWN]:
+            self.rect.y += 10
```


While that code achieves the animation goal, it’s not going to make writing a more complete game very achievable, so we’re going to organise our animated image into a Sprite. Images and their location on screen are common partners; a "sprite" is often used to combine the two. Even better, the sprite stores the image *rect* so we know the full space on screen that it occupies. Note the sprite-specific event handling is now part of the sprite itself. 08-sprite.py

Drawing

```
class Game(object):
    def main(self, screen):
        clock = pygame.time.Clock()

-       image = pygame.image.load('player.png')
+       sprites = pygame.sprite.Group()
+       self.player = Player(sprites)

        while 1:
            clock.tick(30)
...
                if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                    return

+           sprites.update()
            screen.fill((200, 200, 200))
-           screen.blit(image, (320, 240))
+           sprites.draw(screen)
            pygame.display.flip()
```


pygame uses "groups" to collect sprites for rendering: you can render them yourself but you usually end up with a bunch and it's good to have them organised.

Smooth Timing

```
        while 1:
-           clock.tick(30)
+           dt = clock.tick(30)

            for event in pygame.event.get():
                if event.type == pygame.QUIT:
...
                if event.type == pygame.KEYDOWN and event.key == pygame.K_ESCAPE:
                    return

-           sprites.update()
+           sprites.update(dt / 1000.)
            screen.fill((200, 200, 200))
            sprites.draw(screen)
            pygame.display.flip()
```


Here we handle the differences between computer capabilities by incorporating the amount of time passed into the movement calculations. Here we handle the differences between computer capabilities by incorporating the amount of time passed into the movement calculations. The sprite's update() method is passed the change in time (the "delta t") since the last call. Note that pygame sprite update() methods will accept and pass on any arguments.
09-smooth.py
Note that we divide the time by a floating-point number because the result will almost certainly be less than 1.

Smooth Timing

```
        self.image = pygame.image.load('player.png')
        self.rect = pygame.rect.Rect((320, 240), self.image.get_size())
```

```
-     def update(self):
+     def update(self, dt):
          key = pygame.key.get_pressed()
          if key[pygame.K_LEFT]:
-             self.rect.x -= 10
+             self.rect.x -= 300 * dt
          if key[pygame.K_RIGHT]:
-             self.rect.x += 10
+             self.rect.x += 300 * dt
          if key[pygame.K_UP]:
-             self.rect.y -= 10
+             self.rect.y -= 300 * dt
          if key[pygame.K_DOWN]:
-             self.rect.y += 10
+             self.rect.y += 300 * dt
```


Now we modify the sprite update to use the delta t. It will typically be a fraction of a second - hopefully about 1/30. or 0.03s so this modification should retain the same speed we had before, assuming the computer was running at the full 30 FPS before.

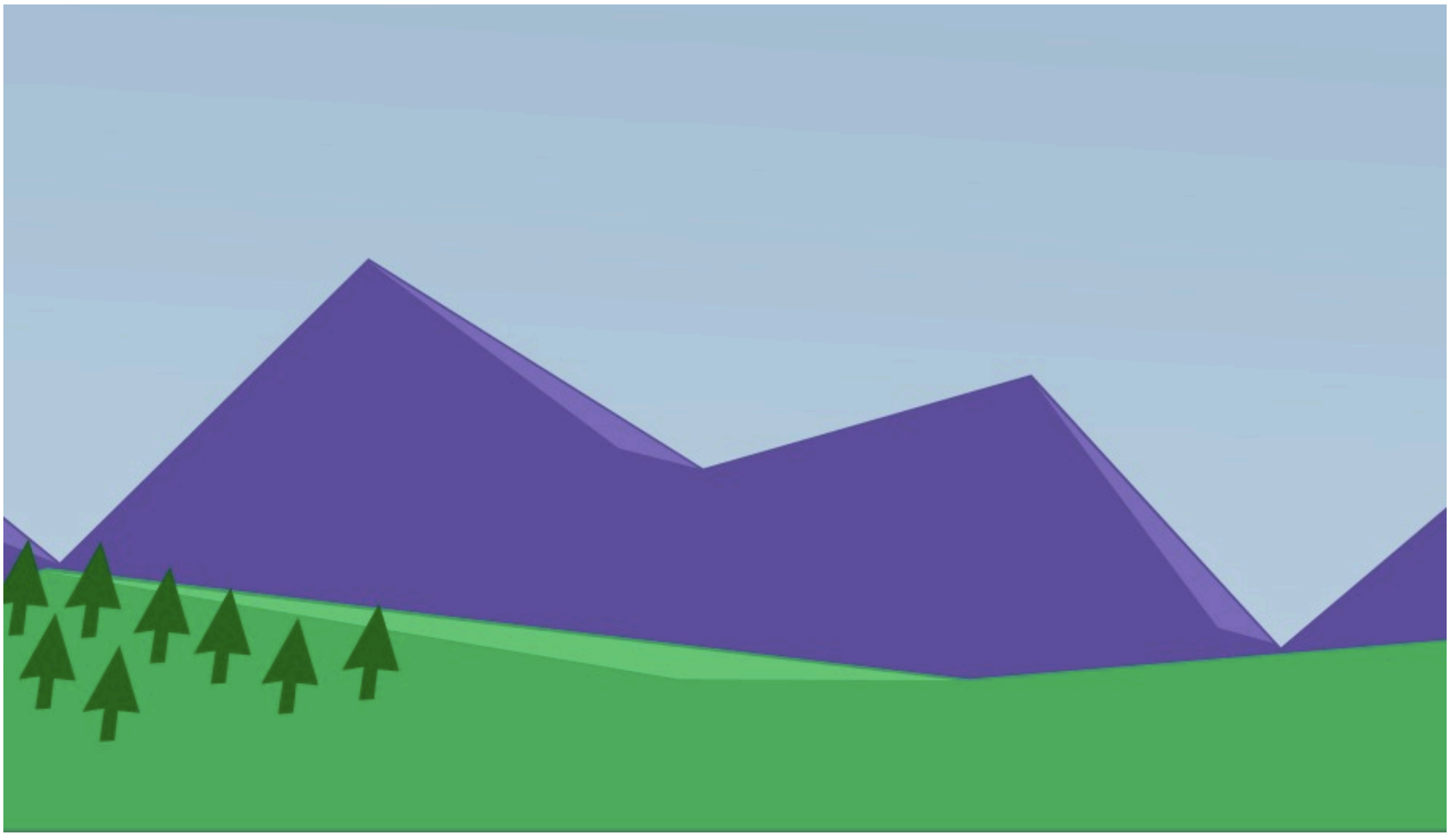
Scene construction

- Scenes are comprised of layers
- These are draw from back to front

Sky

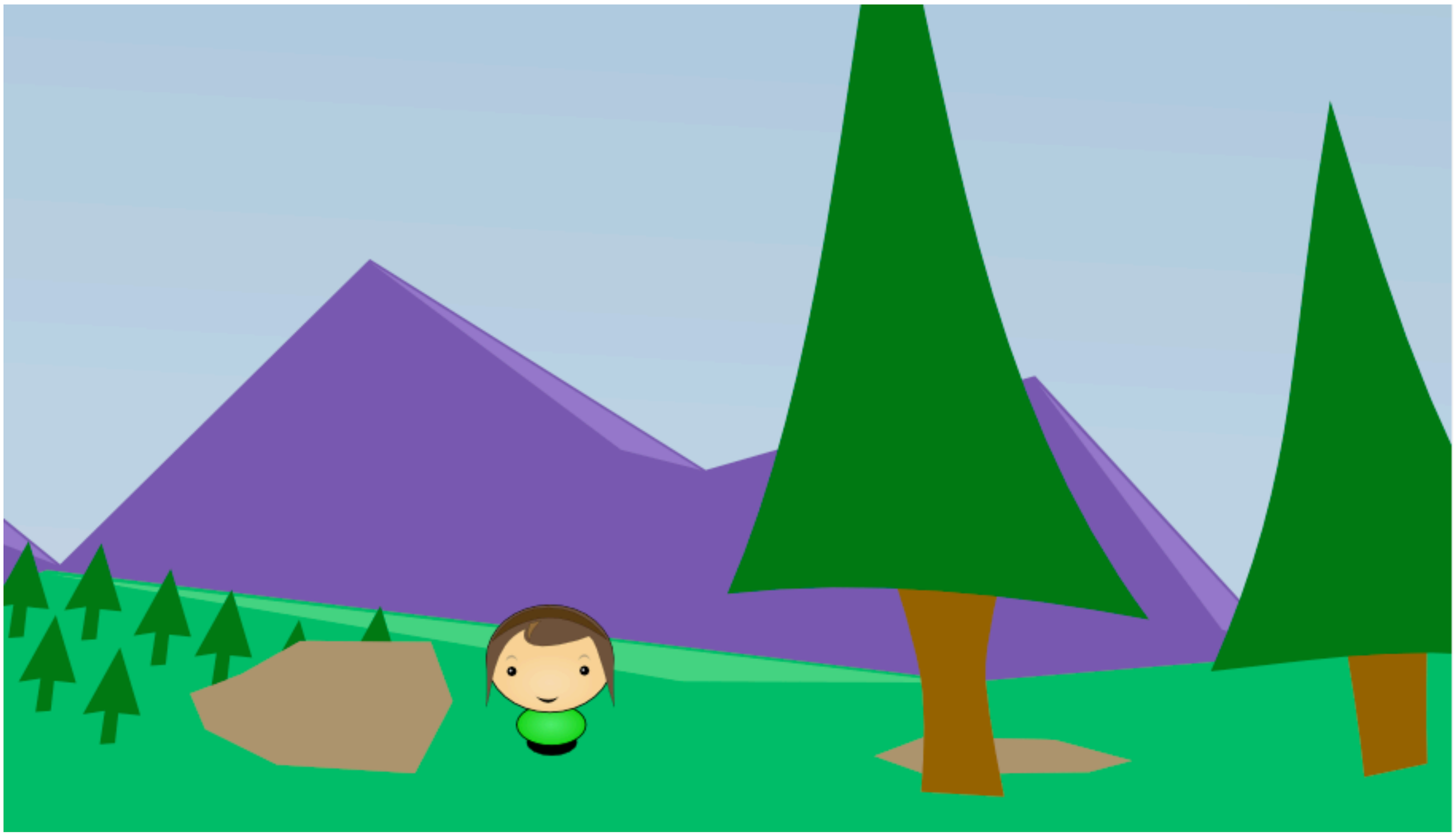
fixed

Background

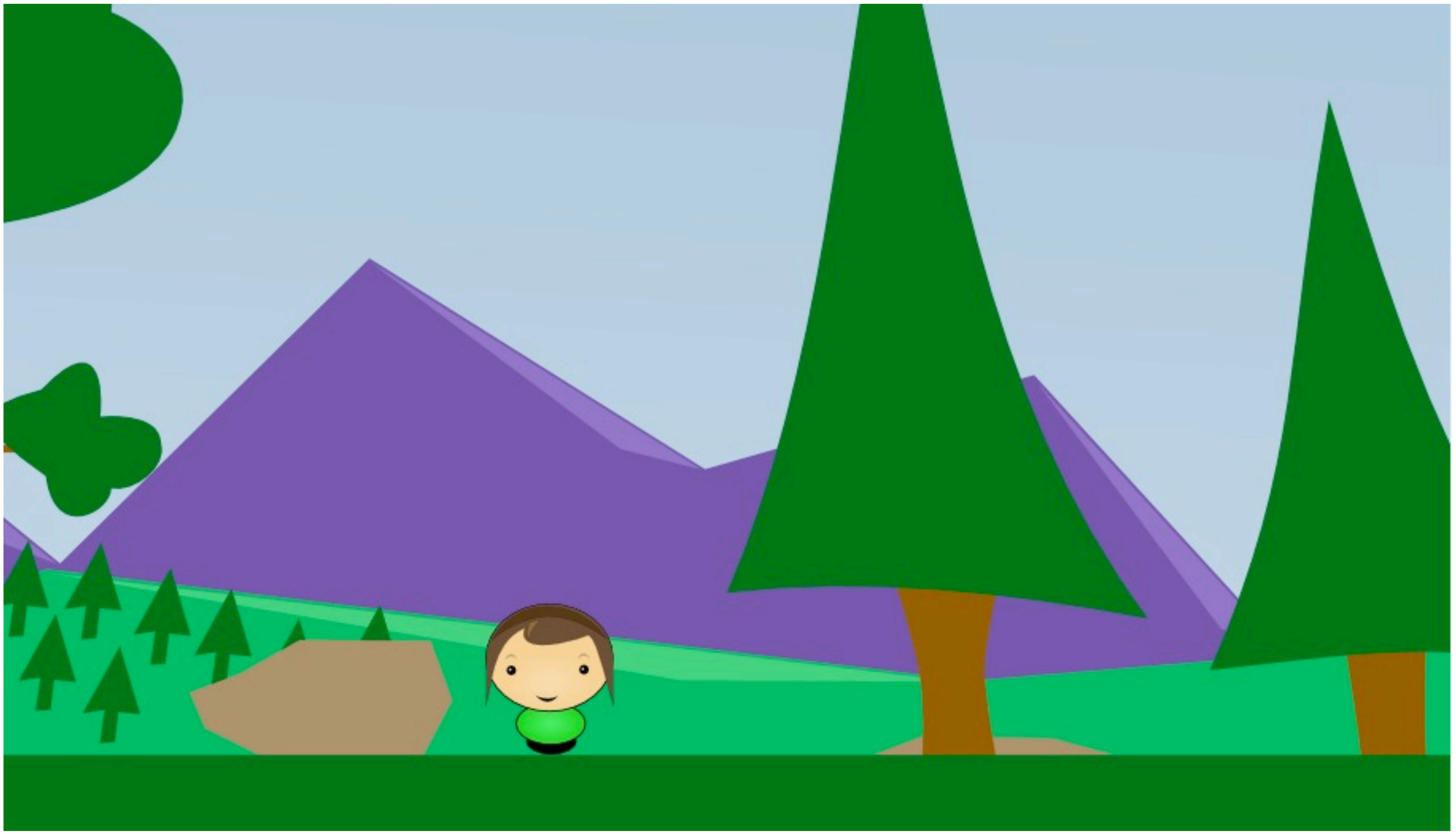


We separate the distance and midfield so we might display them with some parallax.

Foreground 1



Foreground 2



Scene Construction

```
    def main(self, screen):
        clock = pygame.time.Clock()

+       background = pygame.image.load('background.png')
        sprites = pygame.sprite.Group()
        self.player = Player(sprites)
...

            sprites.update(dt / 1000.)
-           screen.fill((200, 200, 200))
+           screen.blit(background, (0, 0))
            sprites.draw(screen)
            pygame.display.flip()
```


Instead of the plain background we add a nice coloured one. We construct our simple scene by drawing a background image first, then the foreground sprites. 10-background.py

Gameplay Mechanics

- Player Controls
- Timed rules (when objects appear; moving the play field)
- Interaction of game objects
- Detecting important events (game won or game over)

Adding obstacles

So now we introduce some basic game mechanics - a limit to where the player can move in the game world.

Detecting Collisions

4 common approaches

There are four basic collision detection methods used in most 2d games (and extrapolated to many 3d games.)

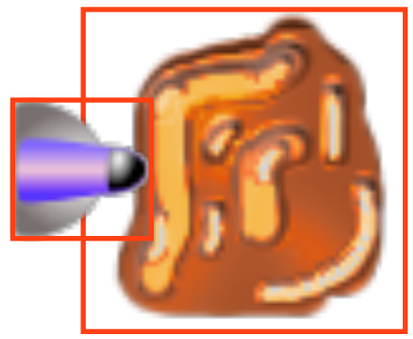
Detecting Collisions

Axis-Aligned Bounding Box

This is by far the most common method. We define a box around each image. This is easy, it uses the dimensions of the original image.

Detecting Collisions

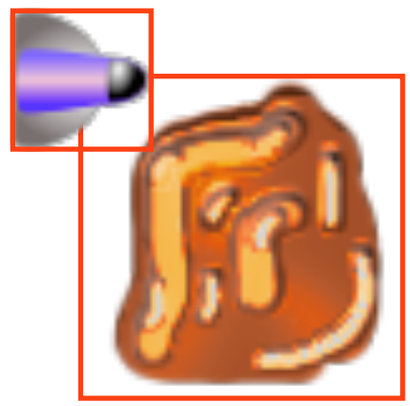
Axis-Aligned Bounding Box



The boxes overlap: collision!

Detecting Collisions

Axis-Aligned Bounding Box



Here you can see collision error where the boxes say we're colliding but the images are not. In fast-moving games the player is unlikely to notice your collision errors. A slight variation is to reduce the size of the box a few pixels - this often produces more fun results.

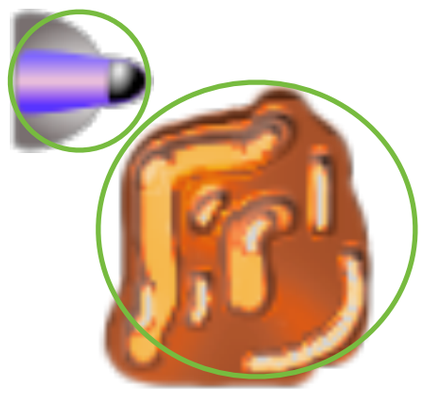
Detecting Collisions

Circle-Circle

An alternative to the box method is to define a circle using the width or height of the image as the diameter of the circle. Then we just have to compare the distance between the mid-points and see whether that's less than the combined radiuses.

Detecting Collisions

Circle-Circle



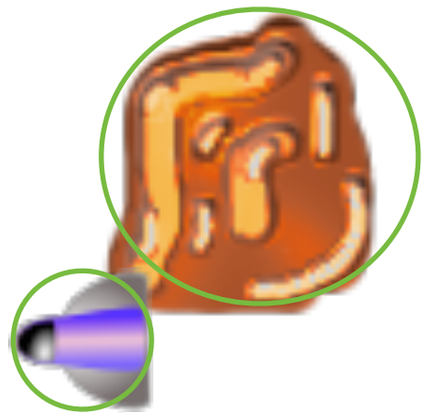
We do better in this case because the images are actually kinda circular.

Detecting Collisions

Circle-Circle

Detecting Collisions

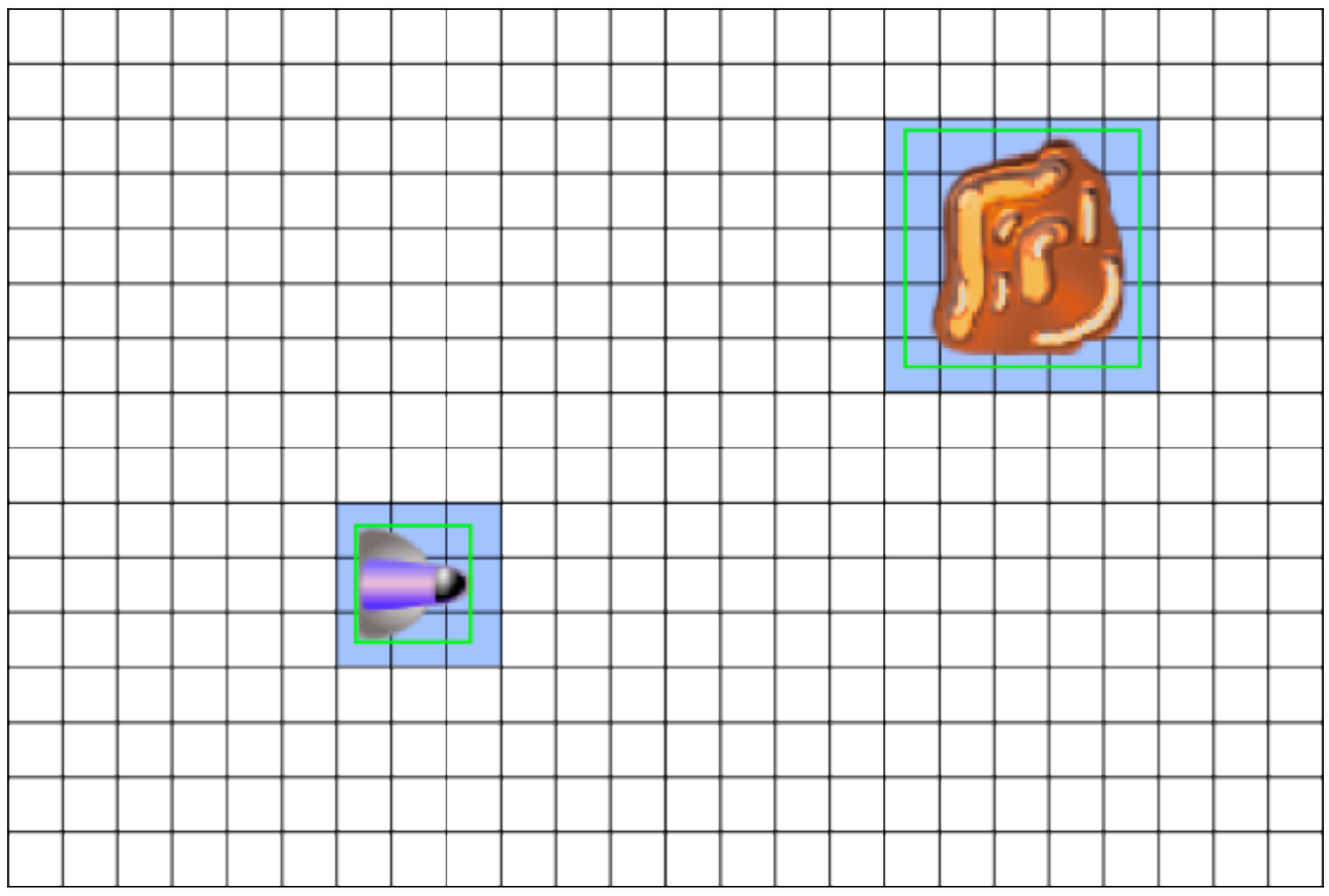
Circle-Circle



But we fail (in the player's benefit) here by missing this collision. Again, the player is not likely to notice in a fast-moving game.

Detecting Collisions

Hash Map



```
d = {(42,42): [ship], (43, 42): [ship], (44, 42): ship, ...
     (52, 45): [asteroid], (53, 45): [asteroid], ...}
```


Here we take the axis-aligned bounding boxes and map them to cells several pixels wide (about 10 in the image above). We can then quickly check whether the something is colliding with anything by looking up its cells.

Detecting Collisions

Pixel-Perfect

In this method we compare each pixel in one image against the pixels in the other image. If any overlap then we have a collision. This is horrendously expensive and almost always overkill, and very few games use this method.

Adding obstacles

```
        sprites.add(background)
        self.player = Player(sprites)
+       self.walls = pygame.sprite.Group()
+       block = pygame.image.load('block.png')
+       for x in range(0, 640, 32):
+           for y in range(0, 480, 32):
+               if x in (0, 640-32) or y in (0, 480-32):
+                   wall = pygame.sprite.Sprite(self.walls)
+                   wall.image = block
+                   wall.rect = pygame.rect.Rect((x, y), block.get_size())
+       sprites.add(self.walls)
```


Adding some walls to the rendering, retaining a reference to the walls on the game object ...

Adding obstacles

```
-               sprites.update(dt / 1000.)
+               sprites.update(dt / 1000., self)
                sprites.draw(screen)
                pygame.display.flip()
```


... and passing the Game object to the sprite update ...

Adding obstacles

```
        self.image = pygame.image.load('player.png')
        self.rect = pygame.rect.Rect((320, 240), self.image.get_size())

-    def update(self, dt):
+    def update(self, dt, game):
+        last = self.rect.copy()

        key = pygame.key.get_pressed()
        if key[pygame.K_LEFT]:
            self.rect.x -= 300 * dt
...
        if key[pygame.K_DOWN]:
            self.rect.y += 300 * dt

+       for cell in pygame.sprite.spritecollide(self, game.walls, False):
+           self.rect = last
```


... so that we can collide the player with the walls. Basic rectangular collisions. Describe other forms of collision detection. 11-walls.py

Conforming Collisions

```
+        new = self.rect
         for cell in pygame.sprite.spritecollide(self, game.walls, False):
-            self.rect = last
+            cell = cell.rect
+            if last.right <= cell.left and new.right > cell.left:
+                new.right = cell.left
+            if last.left >= cell.right and new.left < cell.right:
+                new.left = cell.right
+            if last.bottom <= cell.top and new.bottom > cell.top:
+                new.bottom = cell.top
+            if last.top >= cell.bottom and new.top < cell.bottom:
+                new.top = cell.bottom
```


You may have noticed that the collision detection stops the player some random distance away from the wall. We modify the collision detection to determine which side of the blocker the player hit and therefore align (conform) the player's side with the blocker's side. This code will be handy for some other things later on... 12-conforming.py
Note how we can't slide along the walls.

Gravity

```
        super(Player, self).__init__(*groups)
        self.image = pygame.image.load('player.png')
        self.rect = pygame.rect.Rect((320, 240), self.image.get_size())
+       self.resting = False
+       self.dy = 0

    def update(self, dt, game):
        last = self.rect.copy()
```


Gravity always improves a game.

We need a couple of extra player variables - vertical speed and a flag to indicate whether the player is resting on a surface.

Gravity

```
             self.rect.x -= 300 * dt
         if key[pygame.K_RIGHT]:
             self.rect.x += 300 * dt
-        if key[pygame.K_UP]:
-            self.rect.y -= 300 * dt
-        if key[pygame.K_DOWN]:
-            self.rect.y += 300 * dt
+
+        if self.resting and key[pygame.K_SPACE]:
+            self.dy = -500
+        self.dy = min(400, self.dy + 40)
+
+        self.rect.y += self.dy * dt
```


We remove the up and down key handlers - that would just be cheating at this point. Now we detect the space bar for jumping if the player is resting on a surface. If the player jumps we give them an impulse of 500 y per second, otherwise we accelerate them down by 40 y per second squared.

Gravity

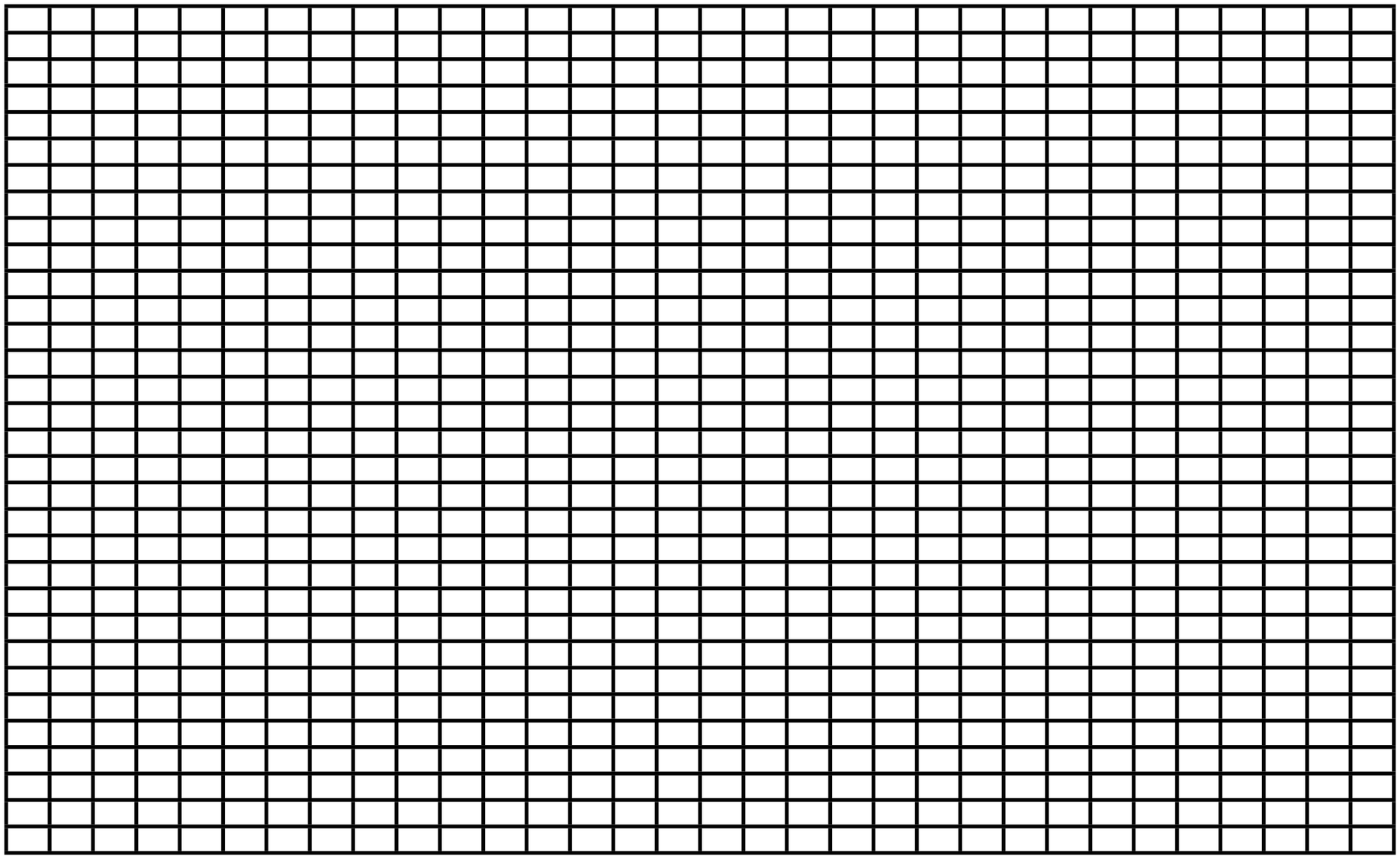
```
            new = self.rect
+           self.resting = False
            for cell in pygame.sprite.spritecollide(self, game.walls, False):
                cell = cell.rect
                if last.right <= cell.left and new.right > cell.left:
...
                if last.left >= cell.right and new.left < cell.right:
                    new.left = cell.right
                if last.bottom <= cell.top and new.bottom > cell.top:
+                   self.resting = True
                    new.bottom = cell.top
+                   self.dy = 0
                if last.top >= cell.bottom and new.top < cell.bottom:
                    new.top = cell.bottom
+                   self.dy = 0
```


Now it's important to keep track of the resting flag, but also to zero out the vertical speed if we hit something upwards or downwards. The reason for that will be demonstrated later. 13-gravity.py

Proper Map To Play In

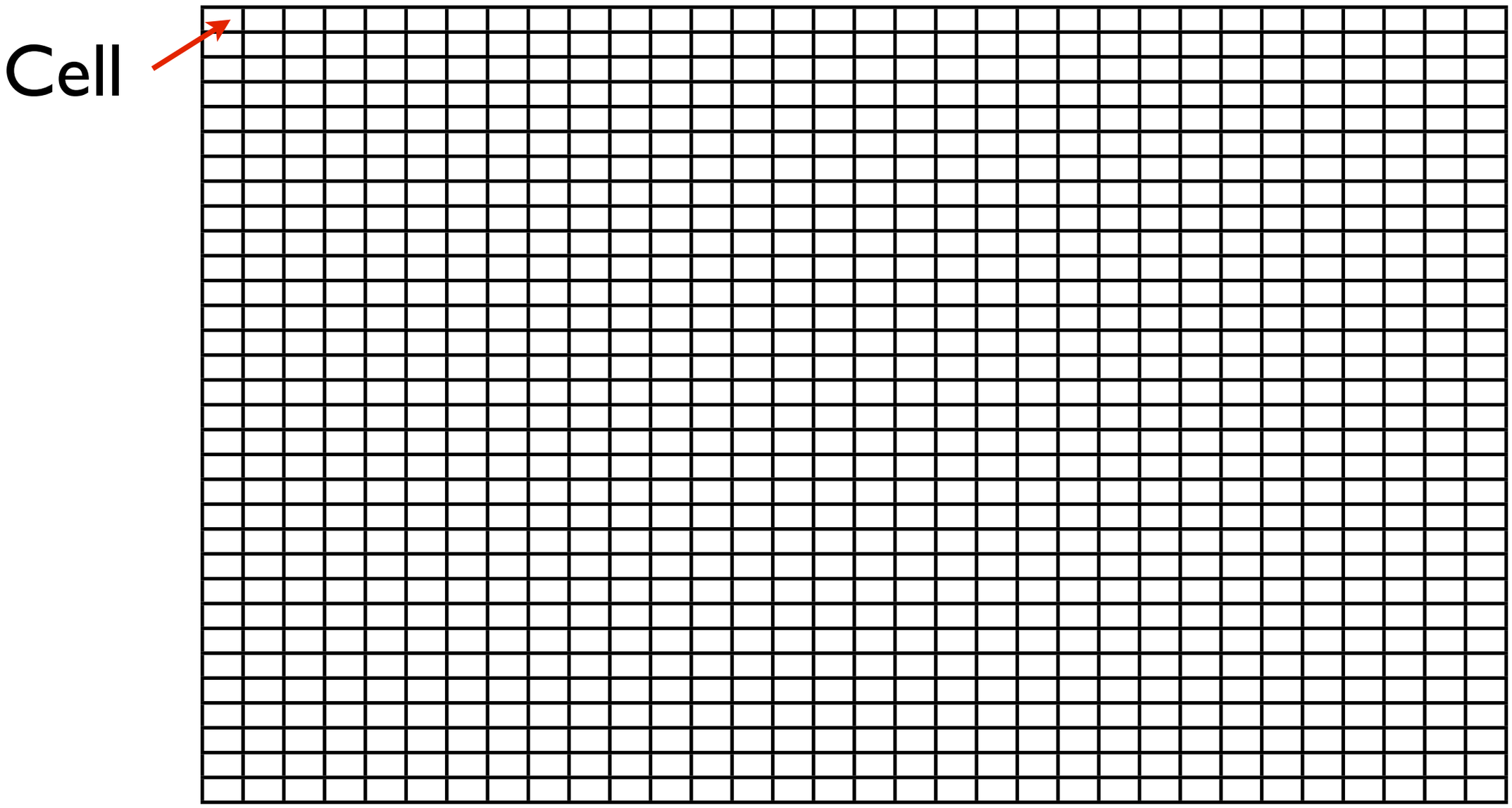
Now we are going to introduce a real tile map loaded from the common TMX format. Using a library, because we're not crazy. There's a bunch of TMX libraries. And TMX tools.

Tile Maps



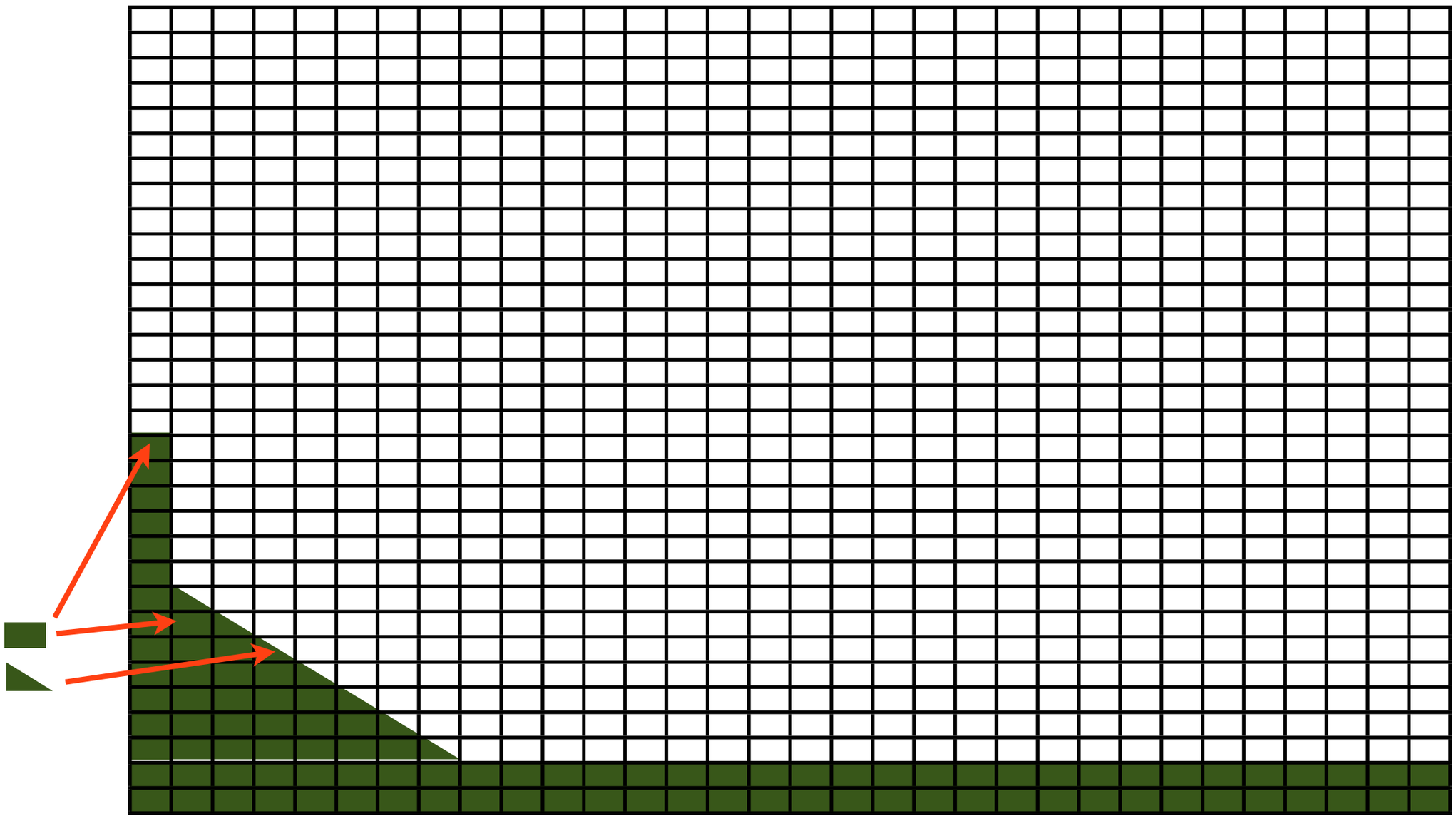
This is a tile map. At the moment it’s blank.

Tile Maps



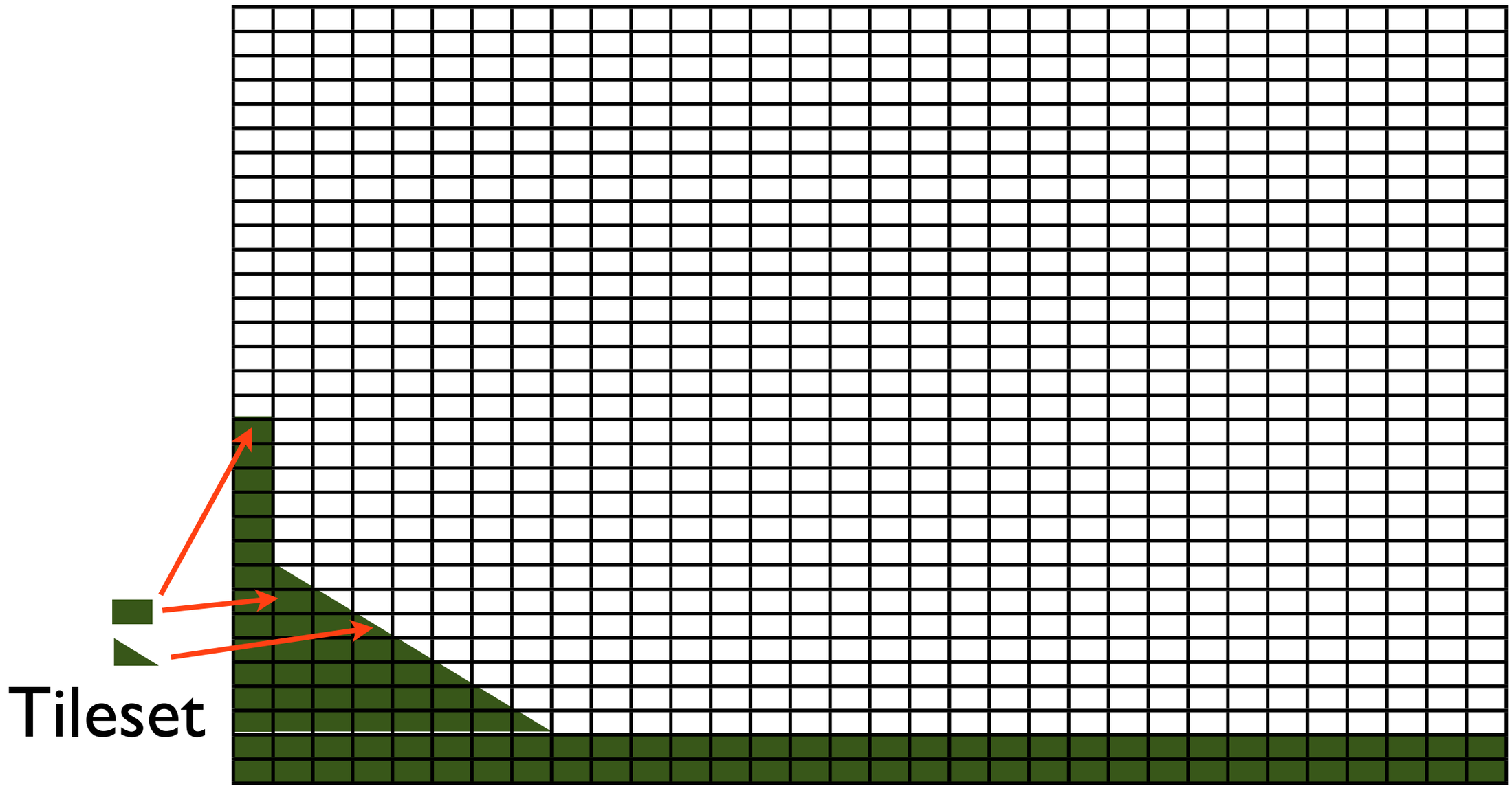
Tile maps are made up of cells. This map is 32 by 32 cells. Cells may be empty.

Tile Maps



Or cells might have contents, or tiles. Here we are using two tiles – a square and a triangle – to draw a part of a basic map.

Tile Maps

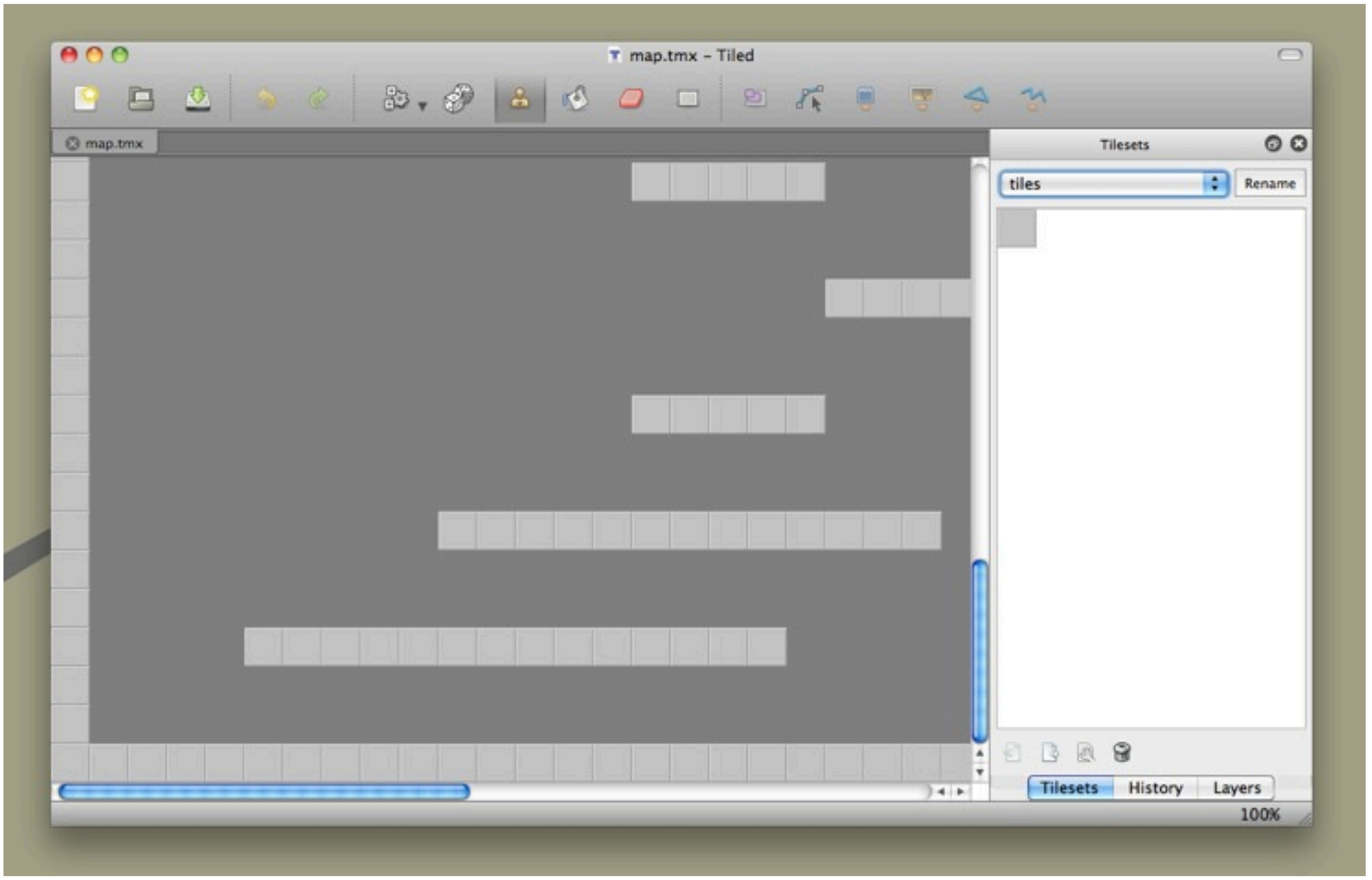


Tiles come from tile sets, which are just collections of tile images which may be placed around tile maps.

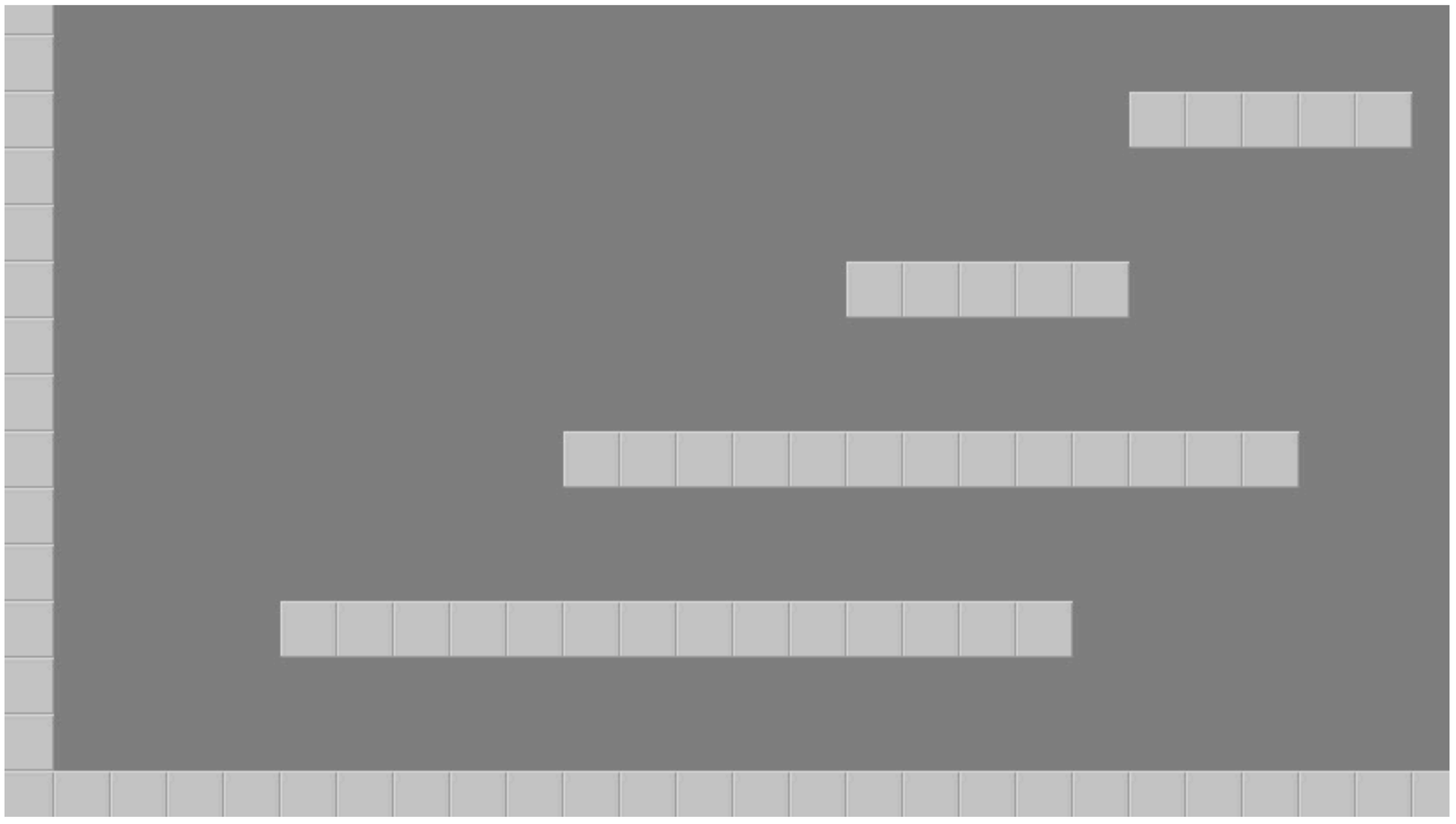
Tiled

http://mapeditor.org/

Tiled

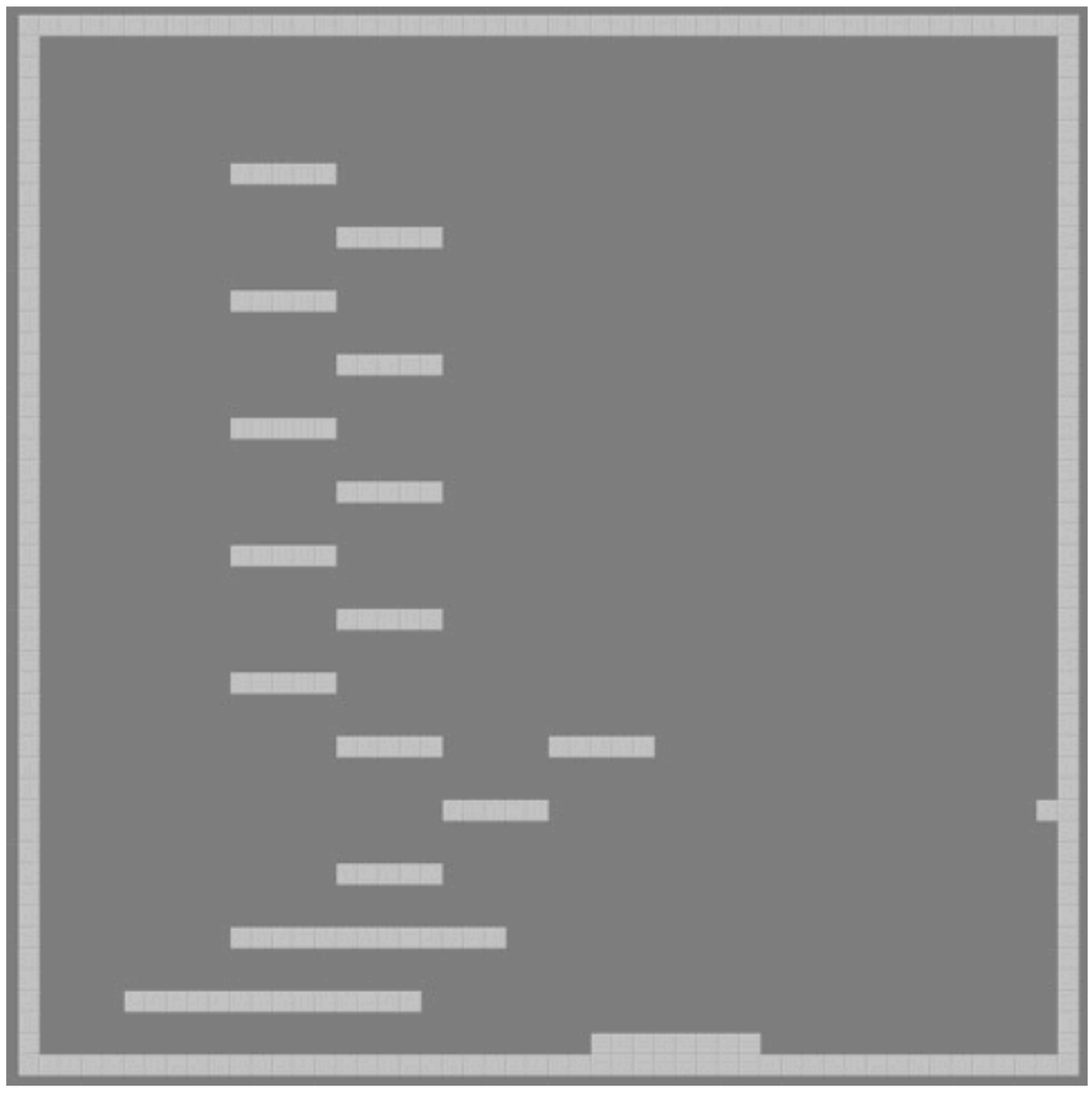

map.tmx – Tiled
map.tmx
Tilesets
tiles
Rename
Tilesets
History
Layers
100%

Tile Mapping



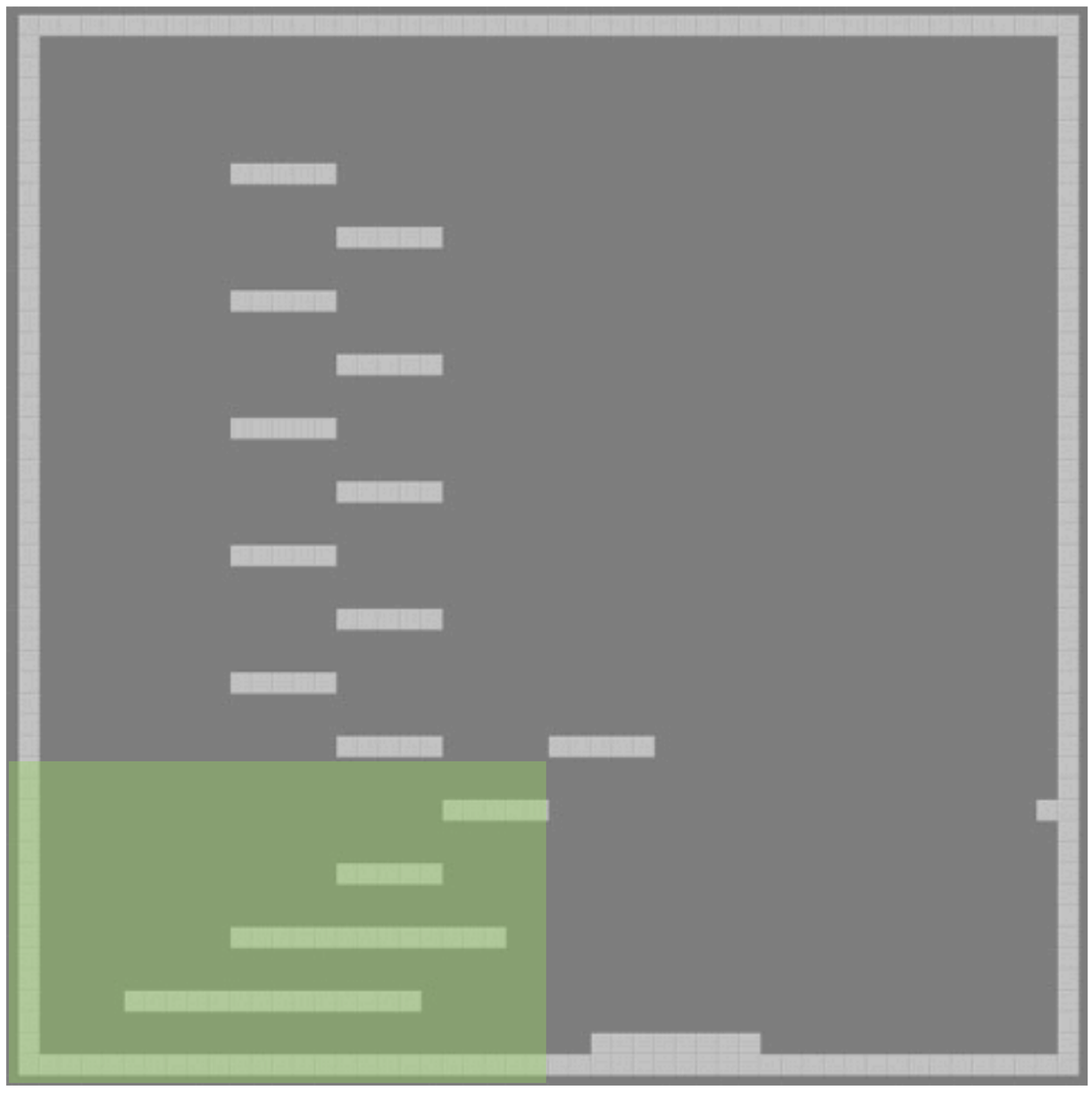
Out map is made of cells - each cell has a tile assigned to it.

Tile Mapping



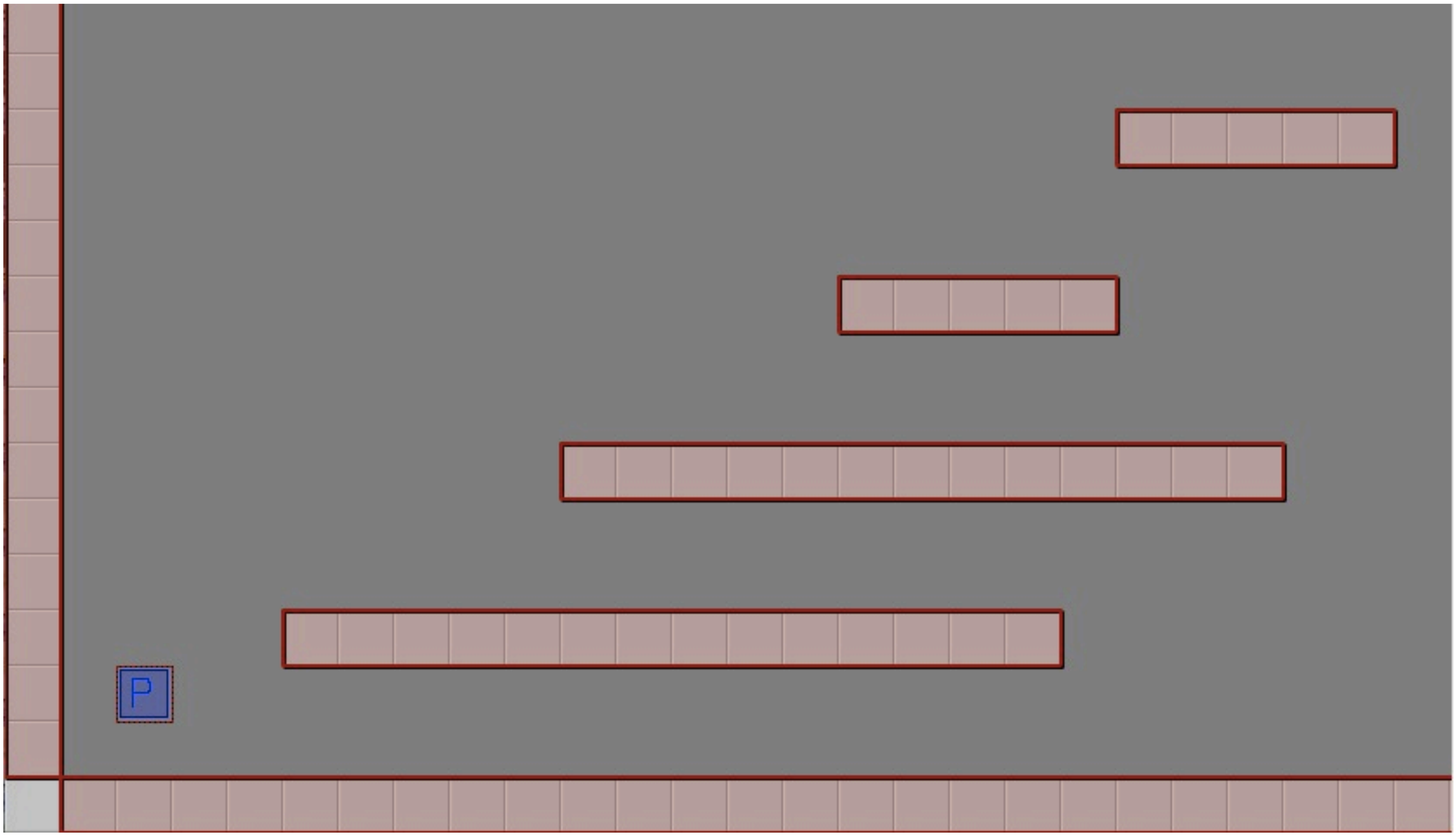
This is the entire map, 50x50 cells. The player only sees a limited view on the map. We call that a viewport.

Tile Mapping



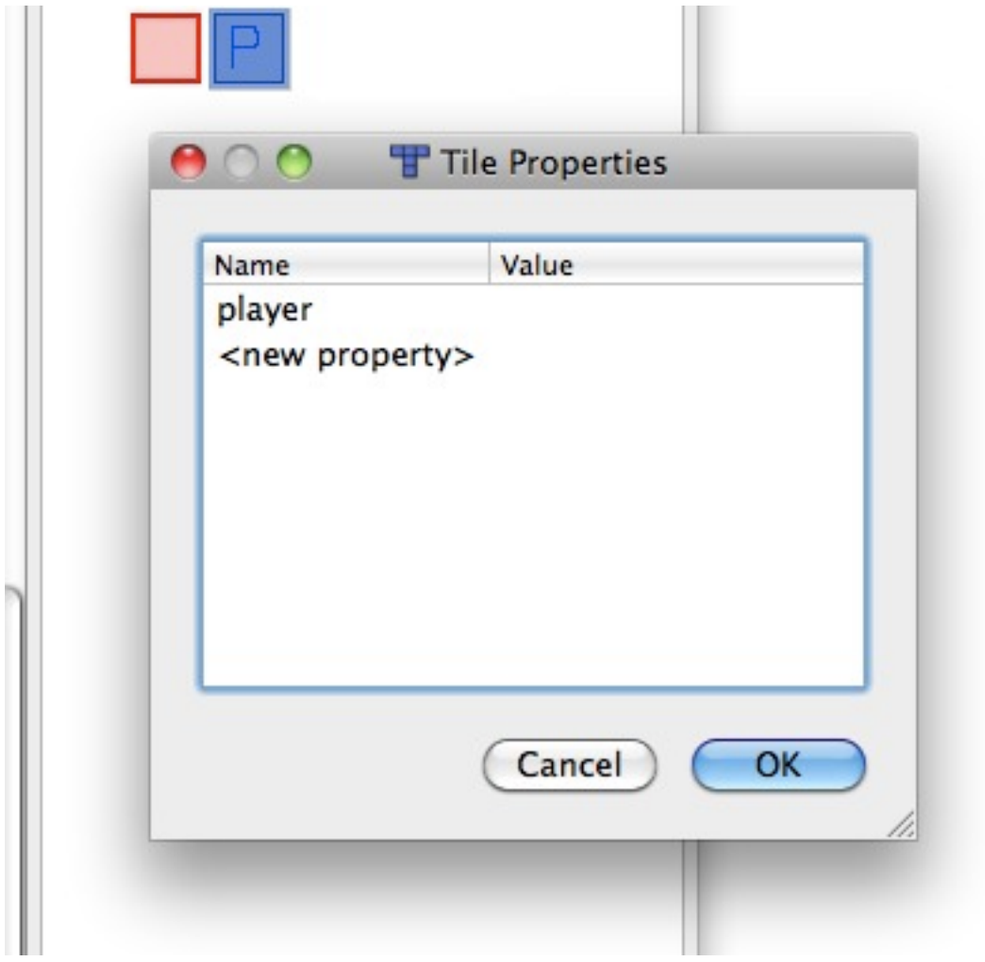
Here's an example viewport.

Tile Mapping



Our tile map has a second invisible layer which has trigger tiles. These triggers may be looked up by our program to see whether something interesting should happen. Above you can see a cell marked as the player spawn location and many other cells marked as “blocker”, meaning the player should not be able to pass.

Tile Mapping



This is the view in Tiled showing the tile properties for the P tile.

Tile Mapping

```
 import pygame
+import tmx
```


So, using a tmx library...

Tile Mapping

```
-        sprites = pygame.sprite.Group()
-        self.player = Player(sprites)

-        self.walls = pygame.sprite.Group()
-        block = pygame.image.load('block.png')
-        for x in range(0, 640, 32):
-            for y in range(0, 480, 32):
-                if x in (0, 640-32) or y in (0, 480-32):
-                    wall = pygame.sprite.Sprite(self.walls)
-                    wall.image = block
-                    wall.rect = pygame.rect.Rect((x, y), block.get_size())
-        sprites.add(self.walls)
```


Now we have to rewrite how our scene is created and managed.

Tile Mapping

```
+        self.tilemap = tmx.load('map.tmx', screen.get_size())
+
+        self.sprites = tmx.SpriteLayer()
+        start_cell = self.tilemap.layers['triggers'].find('player')[0]
+        self.player = Player((start_cell.px, start_cell.py), self.sprites)
+        self.tilemap.layers.append(self.sprites)
```


We now load the tile map from the file and we use the tmx library to manage the viewport, correctly repositioning any sprites it manages so they're rendered in the scrolled position on screen (or off-screen.) Now we're getting the tilemap to manage the updating of out map and sprites, and also the drawing of the map and sprites. The background is static (not scrolling) so it's just blitted separately.

Tile Mapping

```
-               sprites.update(dt / 1000., self)
+               self.tilemap.update(dt / 1000., self)
                screen.blit(background, (0, 0))
-               sprites.draw(screen)
+               self.tilemap.draw(screen)
                pygame.display.flip()
```


The tilemap now manages updates and drawing.

Tile Mapping

```
 class Player(pygame.sprite.Sprite):
-    def __init__(self, *groups):
+    def __init__(self, location, *groups):
         super(Player, self).__init__(*groups)
         self.image = pygame.image.load('player.png')
-        self.rect = pygame.rect.Rect((320, 240), self.image.get_size())
+        self.rect = pygame.rect.Rect(location, self.image.get_size())
         self.resting = False
         self.dy = 0
```


The player start position is now supplied at creation time, taken from the tile map.

Tile Mapping

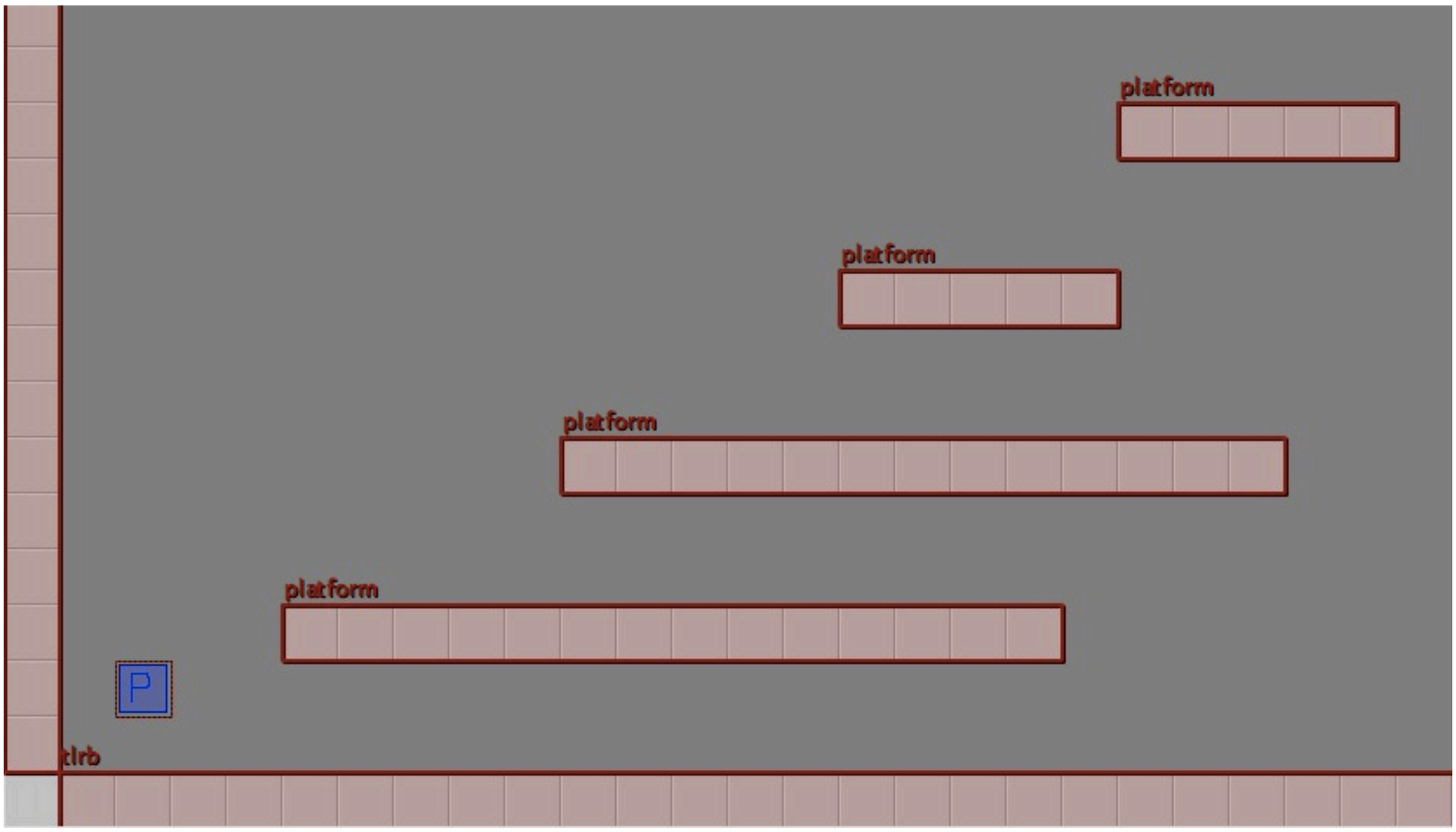
```
            new = self.rect
            self.resting = False
-           for cell in pygame.sprite.spritecollide(self, game.walls, False):
-               cell = cell.rect
+           for cell in game.tilemap.layers['triggers'].collide(new, 'blockers'):
                if last.right <= cell.left and new.right > cell.left:
                    new.right = cell.left
                if last.left >= cell.right and new.left < cell.right:
...
                    new.top = cell.bottom
                    self.dy = 0

+           game.tilemap.set_focus(new.x, new.y)
```


Now we're colliding the player against the tilemap, which means colliding with all the tiles that define the blockers property.
We also control the viewport by setting it to focus (center) on the player. The tmx library is smart enough to restrict the view so we don't try to display cells outside the map. 14-tilemap.py

The platforms...

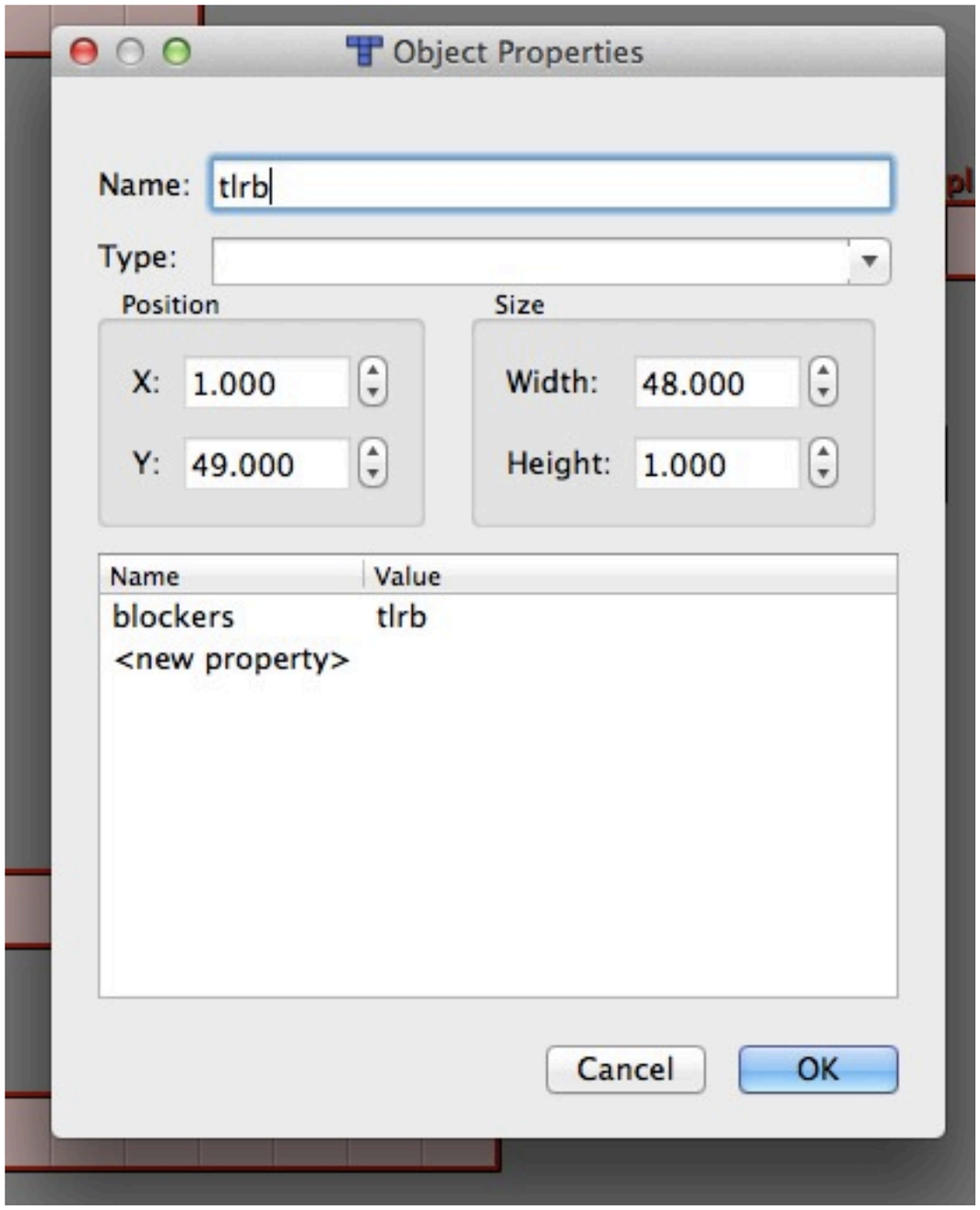
Running the code from the last step you'll notice, as you jump around like a loon, that you can't jump up onto platforms. It's often useful in a platformer to be able to have only some sides of a block that are blocked from passage. To allow us to jump up onto a platform from below (or, perhaps, climb a ladder up onto one) we need to indicate the sides of the blockers that allow passage.

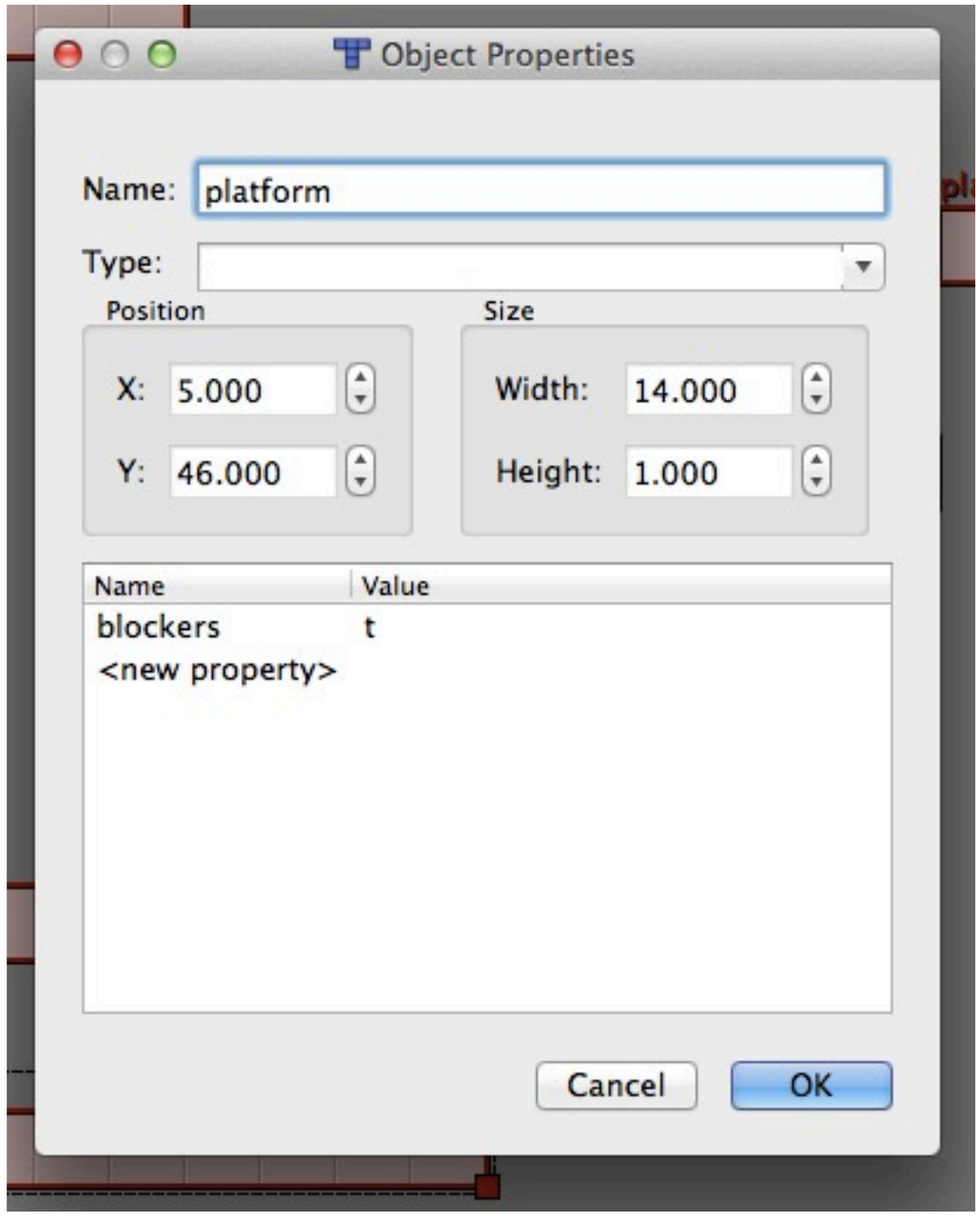
Blocker Sides



The solution is to only mark those sides of the tiles that are going to be collidable - the outside edge of the various shapes the tiles are forming in the level. In this way those pesky incorrect or unwanted collisions are avoided. It does make the code a little more complex though.

Blocker Sides





Tuesday, 26 March 13

The properties of the platform and full blocker shapes.

Blocker Sides

```
        new = self.rect
        self.resting = False
        for cell in game.tilemap.layers['triggers'].collide(new, 'blockers'):
-           if last.right <= cell.left and new.right > cell.left:
+           blockers = cell['blockers']
+           if 'l' in blockers and last.right <= cell.left and new.right > cell.left:
                new.right = cell.left
-           if last.left >= cell.right and new.left < cell.right:
+           if 'r' in blockers and last.left >= cell.right and new.left < cell.right:
                new.left = cell.right
-           if last.bottom <= cell.top and new.bottom > cell.top:
+           if 't' in blockers and last.bottom <= cell.top and new.bottom > cell.top:
                self.resting = True
                new.bottom = cell.top
                self.dy = 0
-           if last.top >= cell.bottom and new.top < cell.bottom:
+           if 'b' in blockers and last.top >= cell.bottom and new.top < cell.bottom:
                new.top = cell.bottom
                self.dy = 0
```

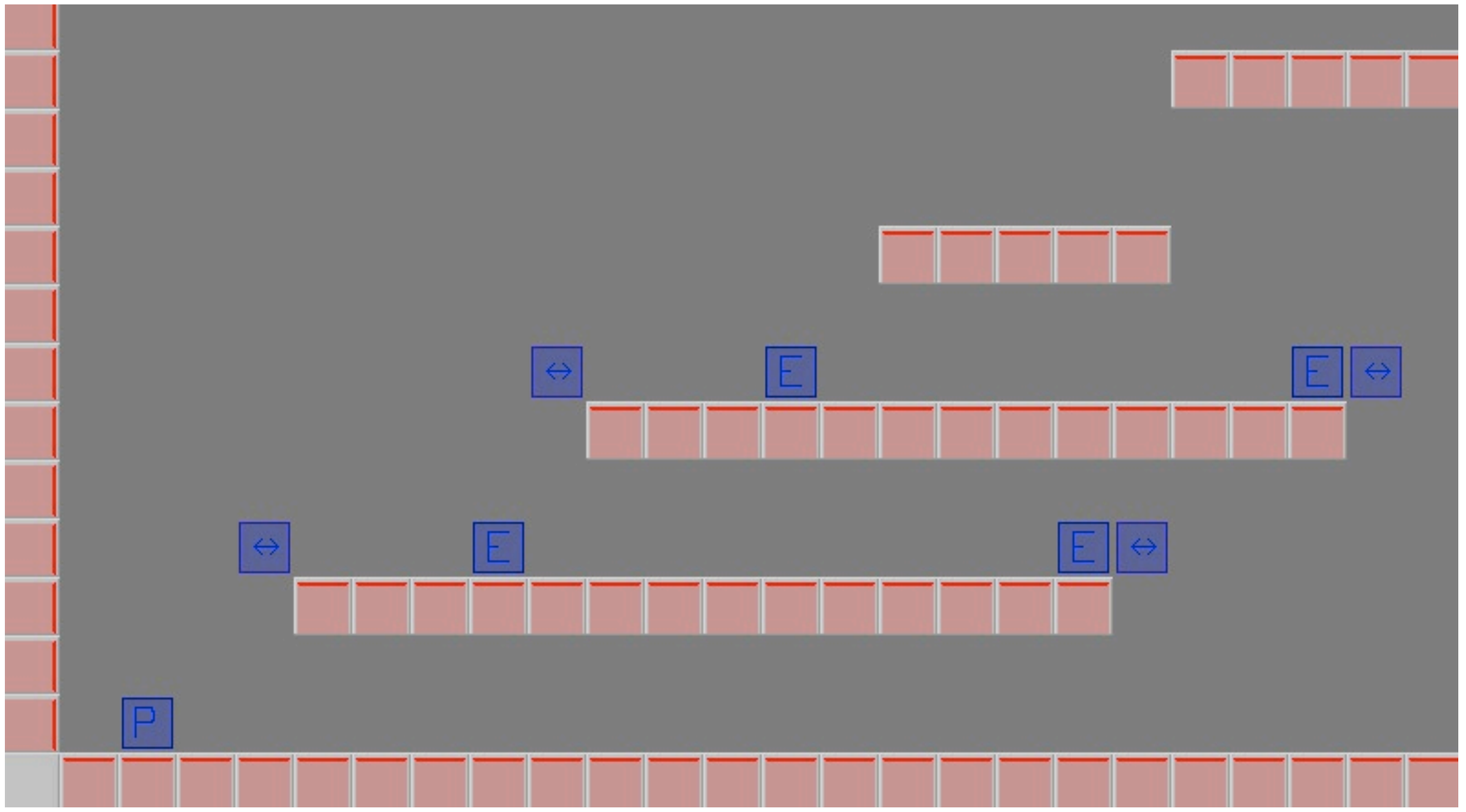

So now the “blocker” property for each tile defines the side (l, r, t, b) that is blocked.
Note that if we wanted platforms the player could jump up onto then we just mark the cell as not blocking from the bottom. 15-blocker-sides.py

Enemies

- Where are the enemies in the level?
- How do they move?
- Do they do anything else?

Anything else: shoot back, alter the level in some way, do special moves?

Enemies



Introducing new map triggers for the enemies.

Enemies

The same approach of triggers in the level works for scrolling shooters too.
This is a side-scrolling shooter game. The red is the screen viewport, the white thing is the player

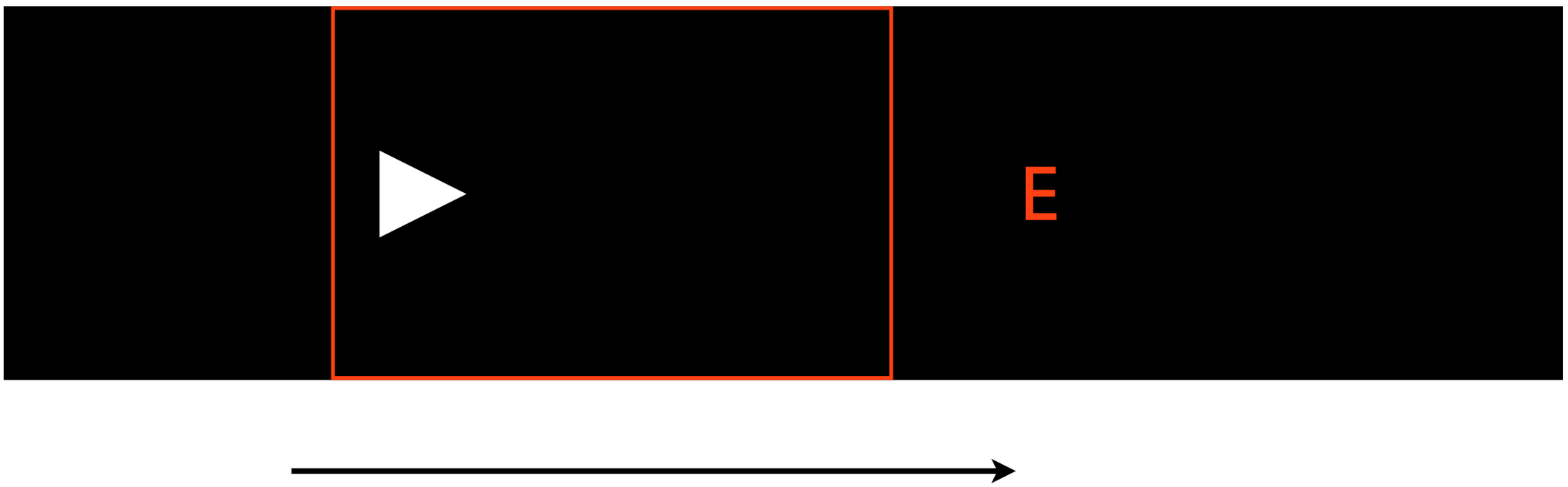
Enemies

player moves this way

Enemies

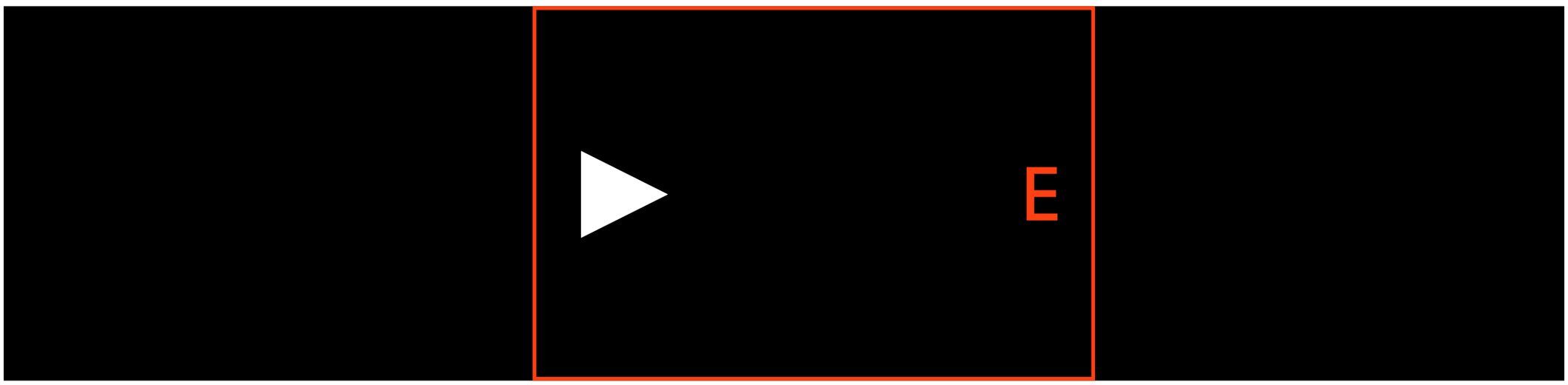
player moves this way

Enemies



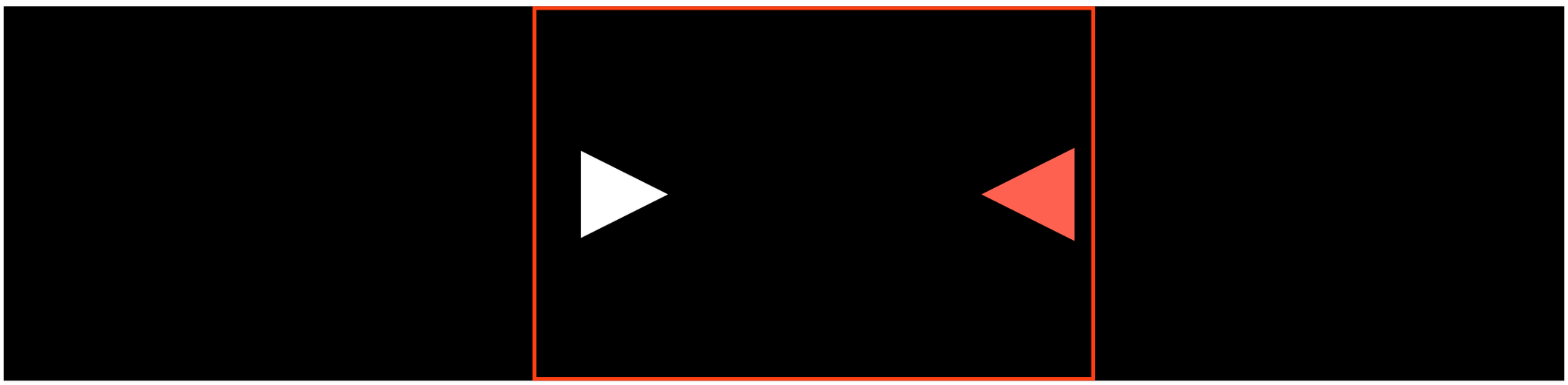
if we introduce triggers into the level map

Enemies



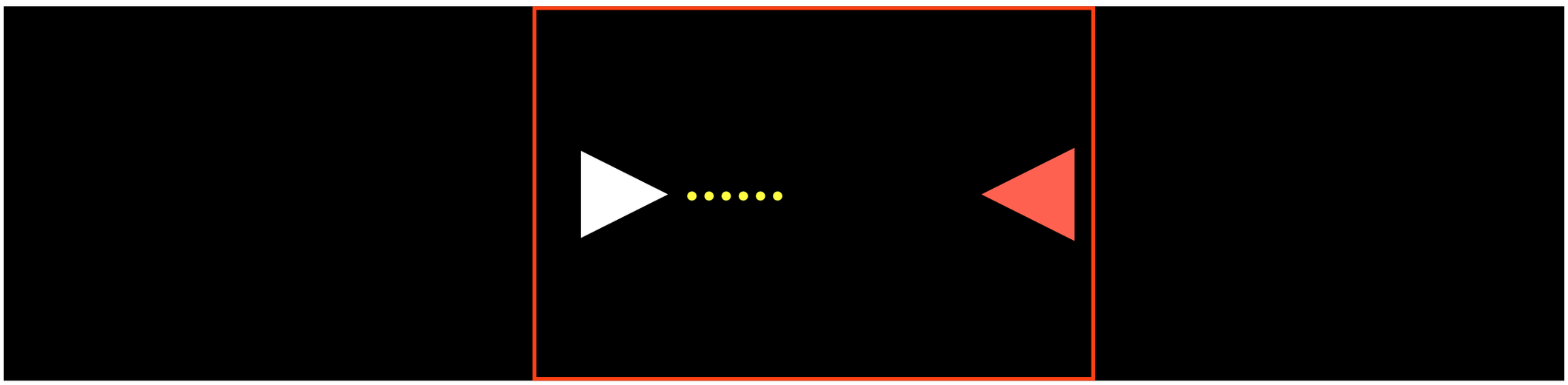
when the player’s view exposes a trigger

Enemies



we create a new enemy!

Enemies



pew pew

Enemies

```
+class Enemy(pygame.sprite.Sprite):
+    image = pygame.image.load('enemy.png')
+    def __init__(self, location, *groups):
+        super(Enemy, self).__init__(*groups)
+        self.rect = pygame.rect.Rect(location, self.image.get_size())
+        self.direction = 1
+
+    def update(self, dt, game):
+        self.rect.x += self.direction * 100 * dt
+        for cell in game.tilemap.layers['triggers'].collide(self.rect, 'reverse'):
+            if self.direction > 0:
+                self.rect.right = cell.left
+            else:
+                self.rect.left = cell.right
+            self.direction *= -1
+            break
+        if self.rect.colliderect(game.player.rect):
+            game.player.is_dead = True
```


A new Sprite subclass for the enemies: simple movement between trigger points on the map. Then simple collision detection with the player to hurt (kill) the player if they contact.

Enemies

```
        self.tilemap.layers.append(sprites)

+       self.enemies = tmx.SpriteLayer()
+       for enemy in self.tilemap.layers['triggers'].find('enemy'):
+           Enemy((enemy.px, enemy.py), self.enemies)
+       self.tilemap.layers.append(self.enemies)

        background = pygame.image.load('background.png')
```


Load up the enemies all located at the “enemy” trigger points in the map.
Enemies are added to a separate sprite group so we can access them later.

Enemies

```
        self.rect = pygame.rect.Rect(location, self.image.get_size())
        self.resting = False
        self.dy = 0
+       self.is_dead = False
```


Keep track of the player's health.

Enemies

```
            self.tilemap.draw(screen)
            pygame.display.flip()

+           if self.player.is_dead:
+               print 'YOU DIED'
+               return
```


Exit the game if the player dies. 16-enemies.py

Player Direction

```
 class Player(pygame.sprite.Sprite):
     def __init__(self, location, *groups):
         super(Player, self).__init__(*groups)
-        self.image = pygame.image.load('player.png')
+        self.image = pygame.image.load('player-right.png')
+        self.right_image = self.image
+        self.left_image = pygame.image.load('player-left.png')
         self.rect = pygame.rect.Rect(location, self.image.get_size())
         self.resting = False
         self.dy = 0
         self.is_dead = False
+        self.direction = 1
```


So far our player has been a single stick figure. Let’s alter the image based on what the player is doing. We need to know which direction the player is facing. Talk about gun cooldown and bullet lifespan.

Player Direction

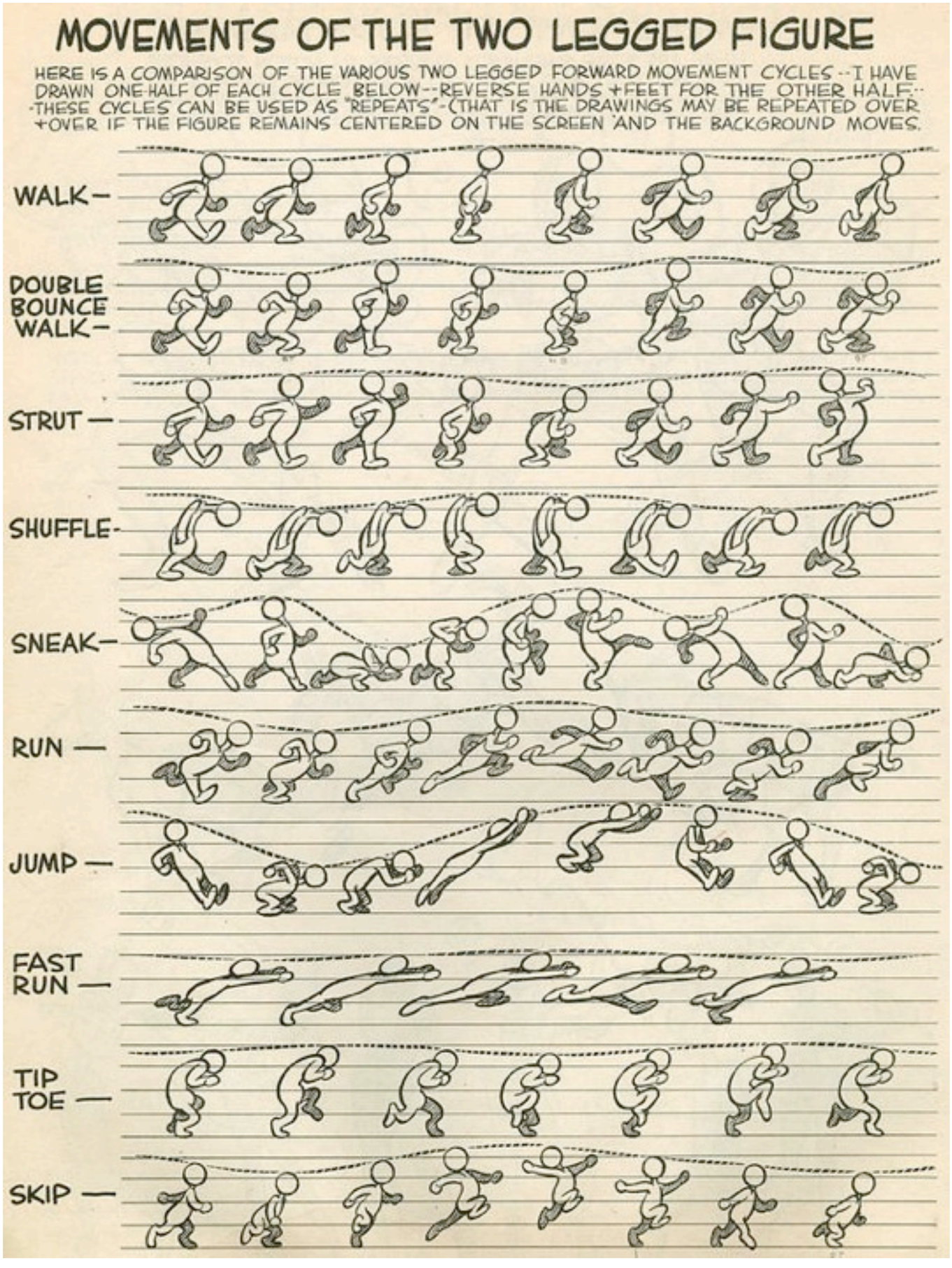
```
    def update(self, dt, game):
        last = self.rect.copy()

        key = pygame.key.get_pressed()
        if key[pygame.K_LEFT]:
            self.rect.x -= 300 * dt
+           self.image = self.left_image
+           self.direction = -1
        if key[pygame.K_RIGHT]:
            self.rect.x += 300 * dt
+           self.image = self.right_image
+           self.direction = 1
```


Note possibility of sprite animation. 17-player-direction.py

Image Animation

- Image position on screen
- Image changes over time

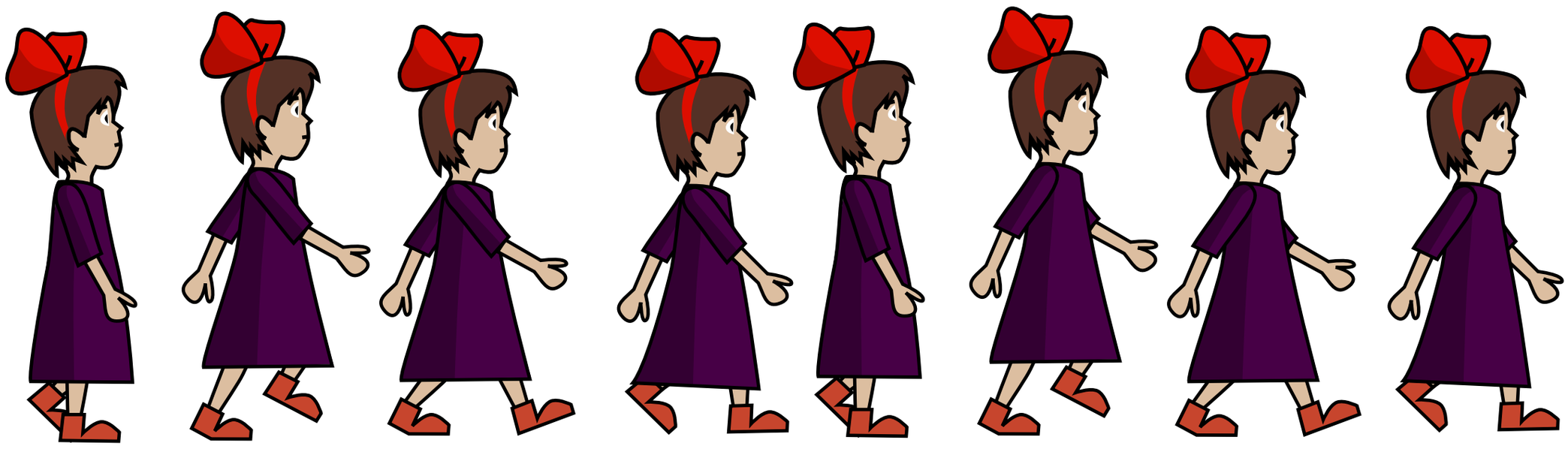
As we've seen we can do animation using a couple of techniques. We're already using movement in our little game. We also have some very, very basic image animation (left or right facing.) We could also introduce running and jumping animations.



Tuesday, 26 March 13

A set of various movements from a classic book “How to Animate” by Preston Blair. It’s trained countless cartoon animators over the years. There’s samples like this and tutorials aplenty out on the interwebs for drawing this kind of thing.

Animation from multiple images

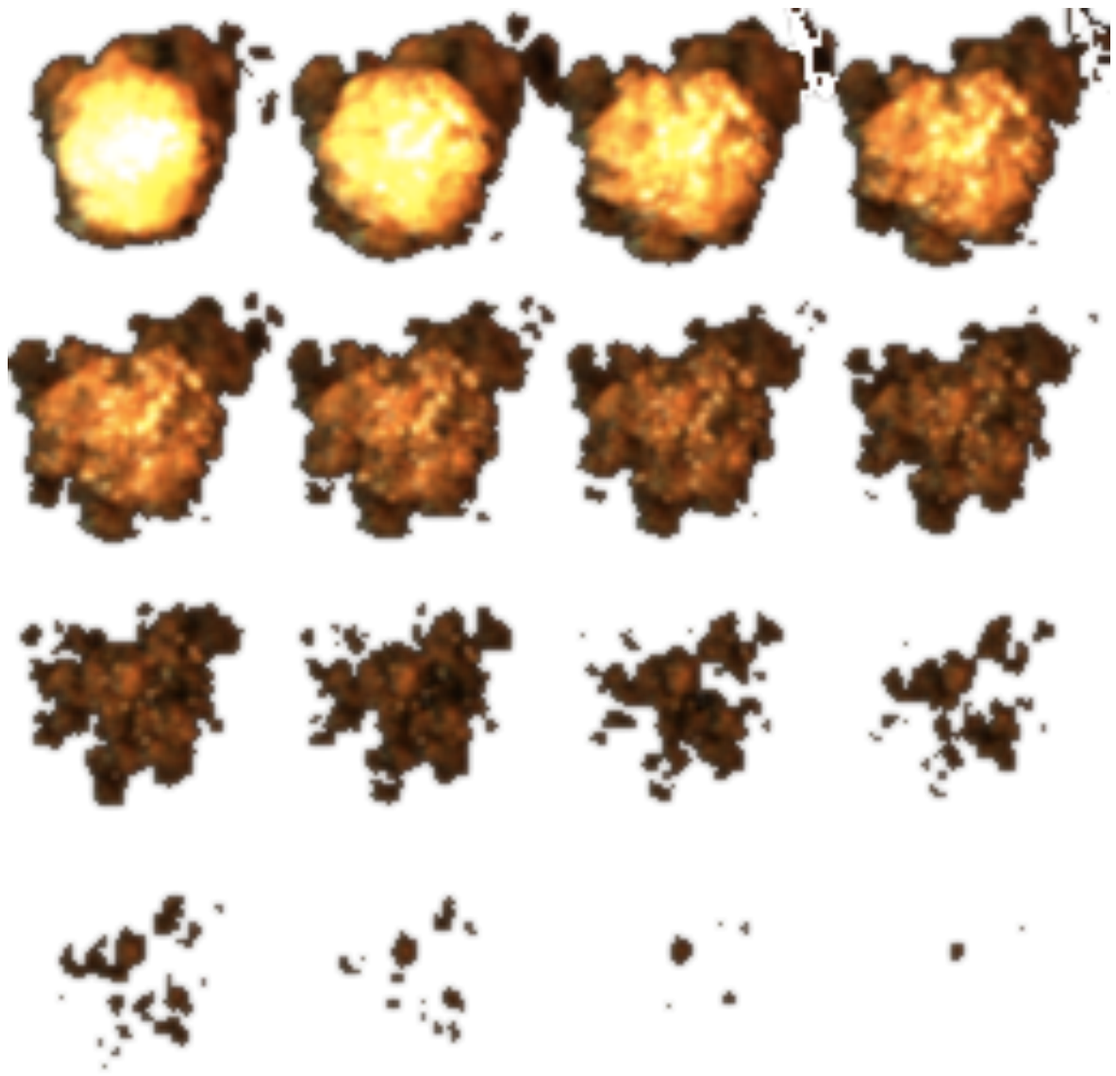


And here's a pretty crappy walk animation I created a couple of years back, before I found those reference images. Still, they worked.

Make life easier for yourself - make all your animation frames the same size!

Also be sure to account for frame rate. Update the animation every N seconds, not every N updates.

Special Effects



Shooting

- initial positioning
- bullet lifespan
- gun cooldown

There's a few things we need to manage when shooting bullets around. The initial position, which should be roughly at the exit point of the weapon being fired - we'll just go with the mid-left or mid-right point of the player sprite. Then there's how long the bullet should live for and how often the player can shoot.

Shooting

```
+class Bullet(pygame.sprite.Sprite):
+    image = pygame.image.load('bullet.png')
+    def __init__(self, location, direction):
+        super(Bullet, self).__init__()
+        self.rect = pygame.rect.Rect(location, self.image.get_size())
+        self.direction = direction
+        self.lifespan = 1
+
+    def update(self, dt, game):
+        self.lifespan -= dt
+        if self.lifespan < 0:
+            self.kill()
+            return
+        self.rect.x += self.direction * 400 * dt
+
+        if pygame.sprite.spritecollide(self, game.enemies, True):
+            self.kill()
```


Note sprite kill()

Shooting

```
        self.dy = 0
        self.is_dead = False
        self.direction = 1
+       self.gun_cooldown = 0
```


so here de define the default value for the gun cooldown - when this is 0 the player may shoot

Shooting

```
            self.image = self.right_image
            self.direction = 1
```

```
+       if key[pygame.K_LSHIFT] and not self.gun_cooldown:
+           if self.direction > 0:
+               Bullet(self.rect.midright, 1, game.sprites)
+           else:
+               Bullet(self.rect.midleft, -1, game.sprites)
+           self.gun_cooldown = 1
+
+       self.gun_cooldown = max(0, self.gun_cooldown - dt)
```

```
            if self.resting and key[pygame.K_SPACE]:
                self.dy = -500
            self.dy = min(400, self.dy + 40)
```


we add a new bullet in the direction the player is facing - at the mid height point on the side of the player sprite. we also need to cool the gun down.
18-shooting.py

Sound

- Reading sound files
- Playing sounds and background music

Sounds

19-sounds.py

Load sounds and .play() sounds when appropriate. Easy! Try not to overload the mixer though!

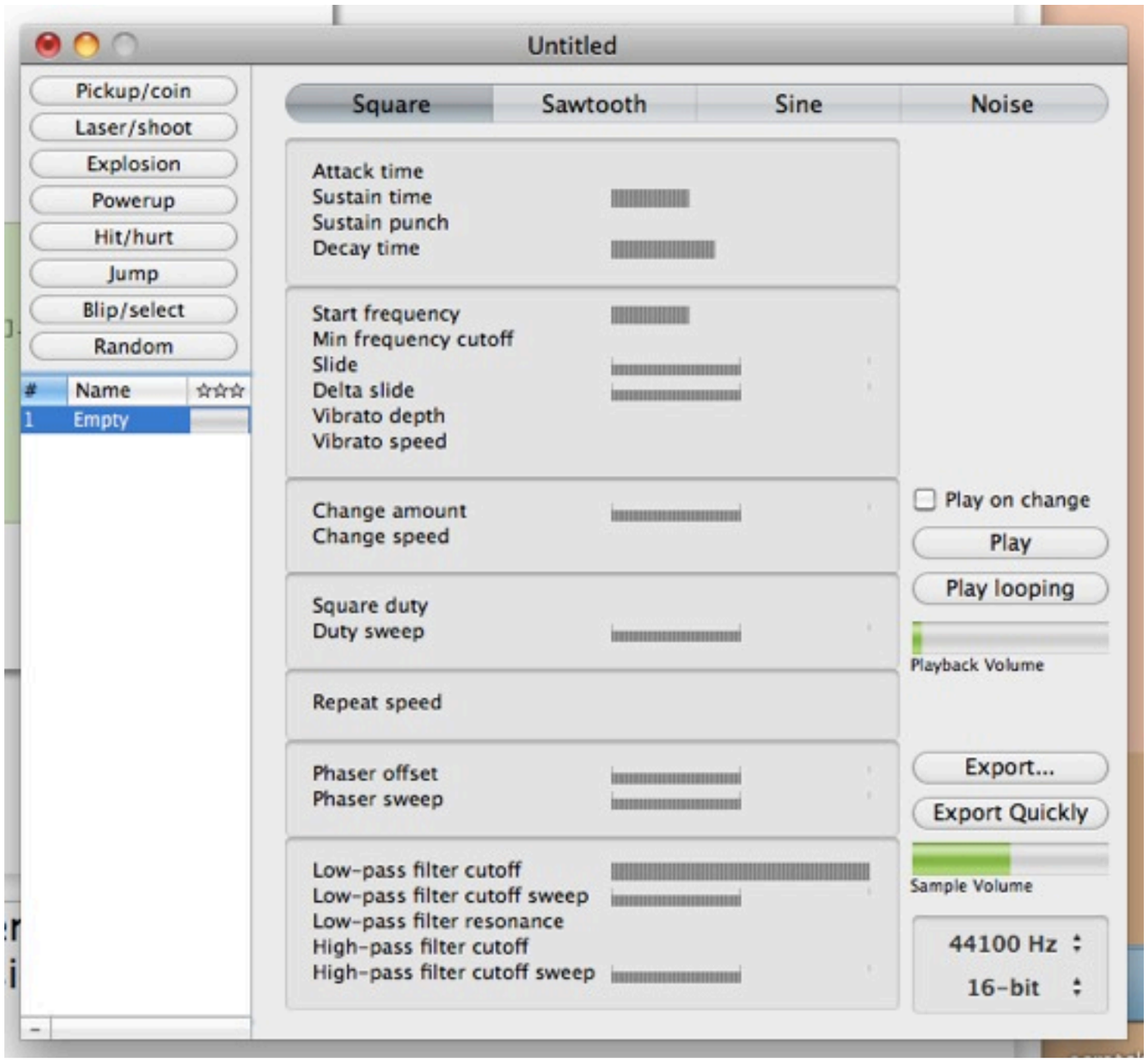
Sounds

```
        background = pygame.image.load('background.png')

+       self.jump = pygame.mixer.Sound('jump.wav')
+       self.shoot = pygame.mixer.Sound('shoot.wav')
+       self.explosion = pygame.mixer.Sound('explosion.wav')
+
        while 1:
            dt = clock.tick(30)
```

Sounds

```
            self.rect.x += self.direction * 400 * dt

            if pygame.sprite.spritecollide(self, game.enemies, True):
+               game.explosion.play()
                self.kill()

...

                else:
                    Bullet(self.rect.midleft, -1, game.sprites)
                self.gun_cooldown = 1
+               game.shoot.play()

            self.gun_cooldown = max(0, self.gun_cooldown - dt)

            if self.resting and key[pygame.K_SPACE]:
+               game.jump.play()
                self.dy = -500
            self.dy = min(400, self.dy + 40)
```

Sounds



All the sound effects were created by this awesome tool cfxr which is a Mac port of the awesome sfxr.

Game Architecture

- Player lives
- Game won / game over screen
- Opening screen
- Options menu

Scenes

(or, adding a menu)

Now we're going to use a third-party library to implement a front menu for the game.

```
import pygame
import kezmenu

from platformer import Game

class Menu(object):
    running = True
    def main(self, screen):
        clock = pygame.time.Clock()
        background = pygame.image.load('background.png')
        menu = kezmenu.KezMenu(
            ['Play!', lambda: Game().main(screen)],
            ['Quit', lambda: setattr(self, 'running', False)],
        )
        menu.x = 200
        menu.y = 100
        menu.enableEffect('raise-col-padding-on-focus', enlarge_time=0.1)

        while self.running:
            menu.update(pygame.event.get(), clock.tick(30)/1000.)
            screen.blit(background, (0, 0))
            menu.draw(screen)
            pygame.display.flip()

if __name__ == '__main__':
    pygame.init()
    screen = pygame.display.set_mode((640, 480))
    Menu().main(screen)
```


20-scenes.py

Special Effects

- Simple image animations
- Juice!
- PARTICLES!!!! - use libraries like lepton

lepton works with pygame or opengl

Other Game Types

- minesweeper
- car driving
- side-scrolling shooter
- tower defence

Minesweeper

- rendering a grid of cells
- detecting mouse clicks on cells
- exposing contents

Car Driving

- different movement model
- detecting correct circuit
- detecting collision with edge of track (pixel-based collision detection)

Side-Scroller

- rendering game layers
- moving the scene sideways (or up)
- handling triggers
- handling many collisions

Tower Defence

- tower placement (selection & position)
- movement solution for creeps

Packaging

- Windows executable
- OS X application
- Android app

Windows executable

- install:
  32-bit Python 2.7 (NOT 64-bit)
  pygame 1.9.1
  py2exe
- still mostly undocumented :(
  (the info is in **GeneralTipsAndTricks**)
- cross fingers

py2exe for match3.py

```
import py2exe
from distutils.core import setup

# override the system DLL detection to allow some pygame font-related DLLs
origIsSystemDLL = py2exe.build_exe.isSystemDLL
def isSystemDLL(pathname):
    if os.path.basename(pathname).lower() in ("sdl_ttf.dll", "libfreetype-6.dll"):
        return False
    return origIsSystemDLL(pathname)
py2exe.build_exe.isSystemDLL = isSystemDLL

dist_dir = os.path.join('dist', 'match3')
data_dir = dist_dir

class Target:
    script = 'main.py'
    dest_base = 'match3'    # set the executable name to 'match3'
setup(
    options={'py2exe': {'dist_dir': dist_dir, 'bundle_files': 1}},
    windows=[Target],
)
```


So py2exe can be a bit fun to figure out. Fortunately there's a bunch of other peoples' experiences out in the Internet. The above is the first half of the setup.py that bundles match3 into an executable. It needs to flag a couple of DLLs that py2exe would otherwise not include. Also because my game source file is called "main.py" I need to go through that odd Target definition to rename the executable (it would otherwise be "main.exe")

py2exe for match3.py

```
import os
import shutil
import glob
import fnmatch

# define our data files
data = []
for dirpath, dirnames, filenames in os.walk('.'):
    data.extend(os.path.join(dirpath, fn) for fn in filenames
        if fnmatch(fn, '*.png'))
data.extend(glob.glob('*.ttf'))
data.extend(glob.glob('*.txt'))

dest = data_dir
for fname in data:
    dname = os.path.join(dest, fname)
    if not os.path.exists(os.path.dirname(dname)):
        os.makedirs(os.path.dirname(dname))
    if not os.path.isdir(fname):
        shutil.copy(fname, dname)
```


Here's the rest of the setup.py file. It copies all the data files my game needs into the directory created by py2exe. We can then zip up that directory and distribute that as my game.

OS X application

- pip install py2app
- py2applet --make-setup main.py
- ... and it broke in many ways

pyInstaller for match3

- Unpack pyinstaller and work it that directory
- python pyinstaller.py -n match3 -w ~/src/match3/main.py
- edit match3/match3.SPEC file
- python pyinstaller.py -y match3/match3.spec
- profit!

Tuesday, 26 March 13

pyInstaller has more documentation than py2app and py2exe but there's still gaps. Edited the spec file to add in the missing data files. In theory I could generate the exe using the same script, but I've not tried that yet.

generated match3.spec

```
a = Analysis(['/Users/richard/src/match3/main.py'],
             pathex=['/Users/richard/Downloads/pyinstaller-2.0'],
             hiddenimports=[], hookspath=None)
pyz = PYZ(a.pure)
exe = EXE(pyz, a.scripts, exclude_binaries=1,
          name=os.path.join('build/pyi.darwin/match3', 'match3'),
          debug=False, strip=None, upx=True, console=False)
coll = COLLECT(exe, a.binaries, a.zipfiles, a.datas, strip=None,
               upx=True, name=os.path.join('dist', 'match3'))
app = BUNDLE(coll, name=os.path.join('dist', 'match3.app'))
```


This is the file as generated by pyInstaller, though reformatted for this slide (their generated file is slightly more readable.)

... modified

```
SOURCE = '/Users/richard/src/match3/'
a = Analysis([SOURCE + 'main.py'],
             pathex=['/Users/richard/Downloads/pyinstaller-2.0'],
             hiddenimports=[], hookspath=None)
pyz = PYZ(a.pure)
exe = EXE(pyz, a.scripts, exclude_binaries=1,
          name=os.path.join('build/pyi.darwin/match3', 'match3'),
          debug=False, strip=None, upx=True, console=False)
```

```
# define our data files
datafiles = TOC()
for dirpath, dirnames, filenames in os.walk(SOURCE + '/data'):
  for fn in filenames:
    path = os.path.join(dirpath, fn)
    bundle_path = os.path.relpath(path, SOURCE)
    datafiles.append((bundle_path, path, 'DATA'))
```

```
coll = COLLECT(exe, a.binaries, a.zipfiles, a.datas, datafiles, strip=None,
               upx=True, name=os.path.join('dist', 'match3'))
app = BUNDLE(coll, name=os.path.join('dist', 'match3.app'))
```


So I just had to add this code to collect up my data files and include the TOC() of them in the COLLECT() thingy.

Android

PyGame Subset 4 Android

android.py build <game> release install

Kivy / python-for-android

python build.py <flags> release

There's two projects that will build Python apps for Android including integration better than the Python scripting for Android project. TODO

pgs4a


```
{"layout": "internal", "orientation": "portrait", "package":
    "net.mechanicalcat.match3", "include_pil": false, "name": "Match 3",
    "icon_name": "Match 3", "version": "1.2.6", "permissions": [],
    "include_sqlite": false, "numeric_version": "126"}
```


You generate a file like this by answering some questions and then editing it. The build process is then quite painless.

Kivy <flags>, oh yes

```
python build.py \
  --package net.mechanicalcat.match3 \
  --name 'Match 3' \
  --version 1.2.6 --numeric-version 126 \
  --private ~/src/match3 \
  --orientation portrait \
  --icon ~/src/match3/data/android-icon.png \
  --presplash ~/src/match3/data/android-presplash.jpg \
  --blacklist blacklist.txt \
  --sdk 14 \
  release
```


Note: --dir vs. --private. I have a PR to fix this; Kivy's moving to a new build system though.

Content Tools

- sfxr
- pyxeledit
- Tiled

Where To From Here?

- http://pygame.org
- http://inventwithpython.com
- http://pyweek.org