

Algorithms

(for Game Design)

Session 1: Introductions, Games, Architecture, Engine

Course Objective

- Gain experience thinking about complex algorithms
- Gain a solid understanding of computer game architectures and algorithms

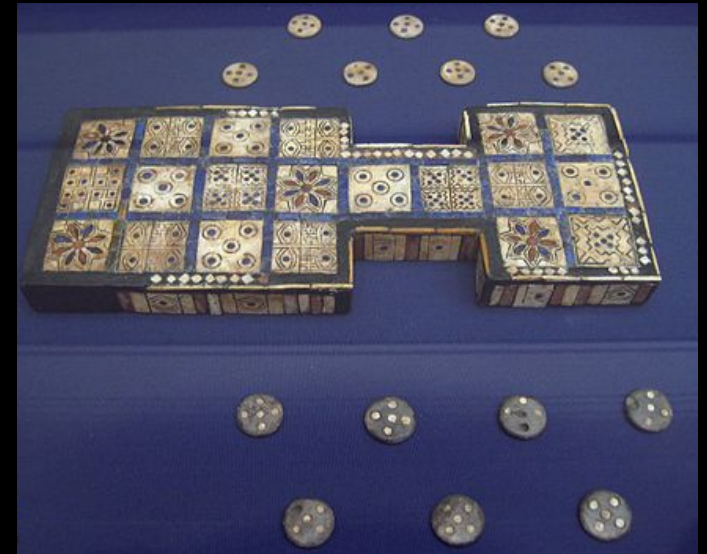
Today

- Introduction to Games & Computer Games
- Introduction to Pygame Engine
 - Game Architectures
 - Game Loops
 - MVC

Textbook: Chapter 0, 1 and 2

Computer Games

- What is a game?
- What is a computer game?
- Why do you play computer games?



Royal Game of Ur
c. 2600 BC

Genre

- Genre = classification based on the type of challenge a game offers
- Can we make a list?

Other Taxonomies

- By platform
 - PC, console, browser, mobile, VR, AR
- By purpose
 - Serious, educational, casual
- By controller
 - Touchscreen, motion capture, dance pad

Why do we build games?

pygame



- Python for writing video games
- Wraps SDL: Simple DirectMedia Layer
- Low-level access to framebuffer, audio, mouse, keyboard and joystick
- Portable to many platforms
- In active development since 2000



Modules

- Modules for controlling different aspects of the game environment

cdrom	playback
cursors	load cursor images
display	control display window
draw	simple shapes
event	manages event queue
font	render text
image	save and load images

joystick	device
key	keyboard device
mouse	device
sndarray	manipulate sounds with
surfarray	manipulate images with
time	control timing
transform	scale/rotate/flip images

Installation

- Use python version 3.5 or greater
- Install using pip

```
$ pip3 install pygame
```

- Check installation with

```
$ python3 -m pygame.examples.aliens
```

- Details at pygame.org/wiki/GettingStarted

Warning: Code coming

- A very simple example
 - Doesn't look simple → 33 lines long
- But, it does relatively amazing stuff
 - Manages a window, draws an image from a file, etc
- This code in intro1.py ← download it and run

```
#!/usr/bin/env python3
''' A simple pygame intro example '''
import pygame
```

```
pygame.init()
```

```
size = width, height = 1024, 768
speed = [3,2]
black = (0, 0, 0)
```

```
screen = pygame.display.set_mode(size)
logo = pygame.image.load('pygame_logo.gif')
logo_width, logo_height = logo.get_size()
logo_x = logo_y = 0
```

```
running = True
while running:
```

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

```
    logo_x += speed[0]
    logo_y += speed[1]
```

```
    if logo_x < 0 or logo_x + logo_width > width:
        speed[0] = -speed[0]
    if logo_y < 0 or logo_y + logo_height > height:
        speed[1] = -speed[1]
```

```
    screen.fill(black)
    screen.blit(logo, (logo_x, logo_y))
    pygame.display.flip()
```

screen is a "surface" object

so is logo

surface methods let you get the size (width, height)

check if the user closed the window

move the logo position

check if going off screen

draw all black

now copy/paste in the logo at its (x, y) location

game loop

Game Loop

- Notice the structure of intro1.py
 - some initialization stuff
 - a loop (while running)
 - Game ends when loop is exited
- This is a very common structure for games

Game Loop (2)

- Each time around the loop:
 1. Check for inputs (from user, network, ...)
 2. Update game state, based on those inputs
 3. Draw the next frame, based on that state

Game Loop: intro1.py

- Our loop had this structure
 1. Did the user close the window?
 2. Add the speed variables to position
 - Check if the logo would be outside the window
 3. Draw the new frame:
 - fill with black
 - paste in the logo image in the new location

Philosophy

- Games need to be responsive
- Often needs to check user inputs
 - and respond to them

Frame

- One picture is rendered (calculated and shown to user) each time around the loop
 - This is a **frame**
- You know this from game ads. "FPS" -- frames per second
 - If lower than 30fps, user perceives lag
 - Often want 60+fps
- Imagine if game state was so complex that updating it took 250ms
 - Maximum of 4fps (and only if rendering was quick)
- Thus, FPS is usually an important design metric/goal

Normal structure of a game

```
initialize_game()
```

```
while not done:
```

```
    *user_inputs, done = get_inputs()
```

```
    game_state = update_game_state(user_inputs)
```

```
    render_game(game_state)
```

BTW: Some languages and systems are built around this structure -- Arduino, Processing

Historical Example

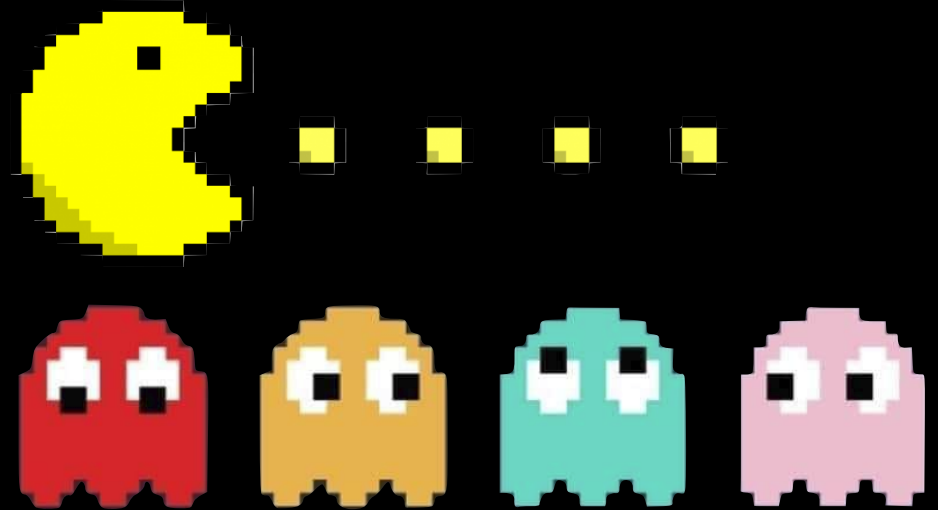
- Pac-Man, released in 1980
- Player guides Pac-Man through the maze, eating dots, avoiding ghosts
 - Also, eating fruit bonuses
 - Eating power pellets, which let him eat the ghosts
- If all dots are eaten → next level



```
# Simplified game loop for Pac-Man
while player.lives > 0:
    inputs = get_joystick_data()
    player.move(inputs)
    for g in ghosts:
        if player.collides(g):
            player.kill() # or g.kill() if power-up
        else:
            g.move(player.position)
# Pac-man eats pellets, fruit, power-ups...
....
# Generate outputs
graphics.draw()
audio.play(sounds)
```

Pac-Man

- You can see the basic structure of the game in the code
- Simplified
 - Attract mode
 - Multiple levels
 - Generation of fruits
 - Regeneration of killed ghosts
 -



Event Queue

- Pygame manages user inputs (and a few other features) with an event queue
 - Queue: a list. First-in-First-Out
 - Event: an object representing something happening in the system
 - Key presses, window resize, mouse clicks,...

Event Queue

- Each time around the game loop, you must handle all the events in the queue
- You are responding to a user action (mostly) → event driven code
- Pygame's event module has very extensive features for handling the event queue

Event Objects

- The event queue contains event objects
- Each event object has a **type** attribute
 - Then other attributes based on type
- type=KEYDOWN has
 - key (what's been pressed), mod (are shift, control keys also pressed), unicode, scancode
- type=MOUSEBUTTONDOWN has
 - pos (x,y of where the cursor was), button (which was clicked)
- type=VIDEORESIZE has
 - size (width, height of the new window)

```
# Example event processing
def get_inputs():
    done = left = right = False
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            done = True
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_ESCAPE:
                done = True
            elif event.key == pygame.K_RIGHT:
                right = True
            elif event.key == pygame.K_LEFT:
                left = True
    return (left, right, done)
```

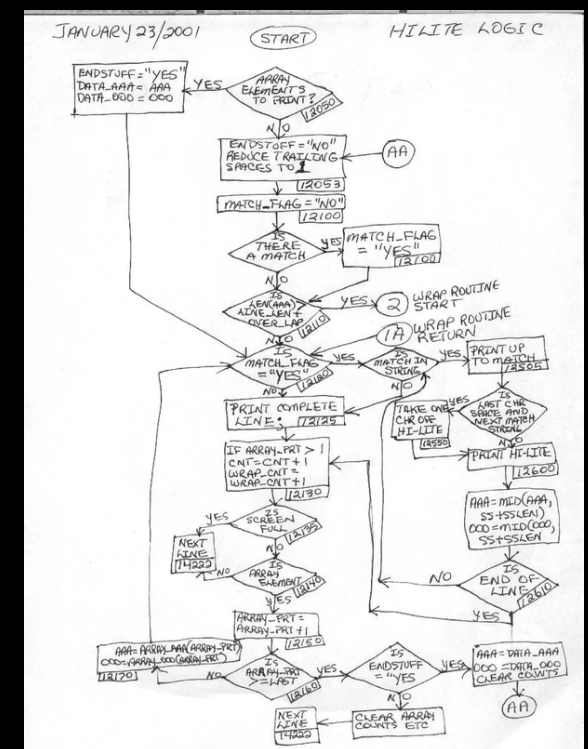
game architecture

Architecture

- **Architecture** is all about the structure of our code
 - What components you build (objects, functions, modules)...
 - And how they interact (what are the interfaces, what parameters get passed)

Spaghetti Code

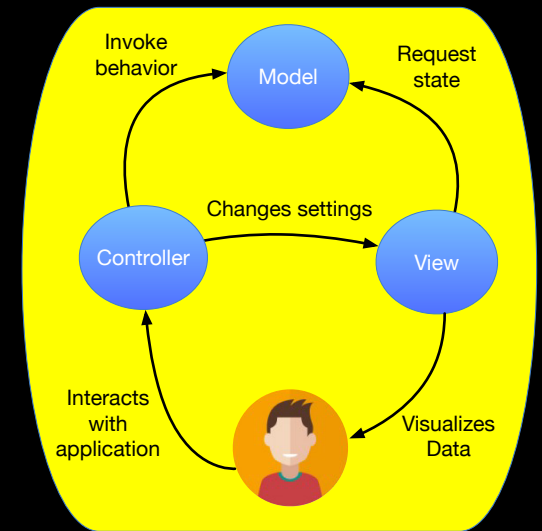
- Architecture matters as your code gets bigger
- Even a bad architecture isn't a problem when it is easy for you to see and understand and remember each bit of code
- Spaghetti code has a complex and tangled control structure
- Difficult to add to or change without error



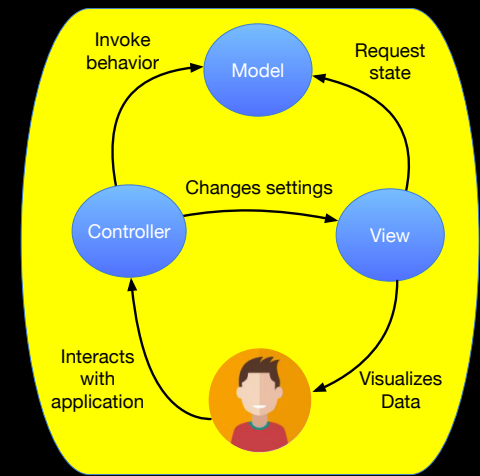
MVC: model-view-controller

MVC: Model-View-Controller

- MVC is a common architectural style for GUI, web applications and games
- I'm not saying your games MUST be MVC, many aren't
- It's a suggestion that you, as the designer, should consider
 - There may be better structures for your game



Model

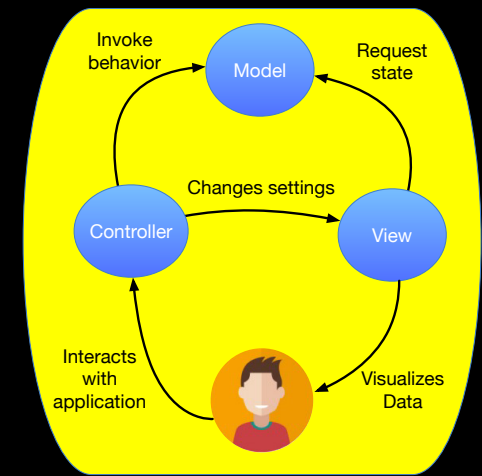


- **Manages the data and operations on the data**
 - data: stuff like, where are the ghosts, how many pellets are left on the level, score
- Provides a well-defined interface to the data
 - Changes to the data structures don't require changes to the rest of the code
- In web and many GUI apps, the model is a database

Model (2)

- Example: Chess Game
 - Data: The model knows where all the pieces are
 - Which pieces have been captured
 - Operations: initializing the board, moving pieces, weird moves (en passant, castling), determining if checkmate exists, scoring of captured pieces, etc.

View

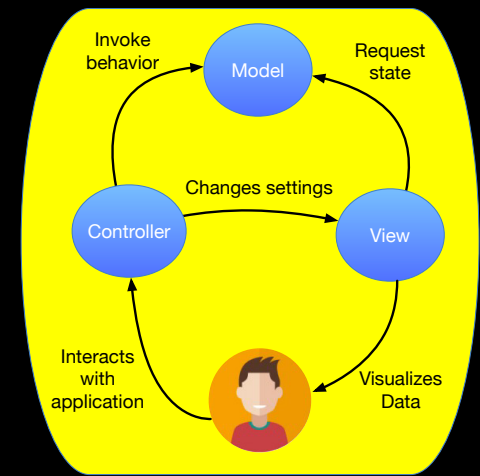


- **Provides a way to visualize the data**
 - Think: Draw the frame
 - But also: make the music, rumble, etc
- In theory: different view components with the same data from the model
 - Often: you want a text-view to aid development
 - Or, a special view to help debug

View

- Example: Chess Game
 - May be rendered in 2D or 3D
 - Perhaps VR
 - iPhone vs laptop vs console views

Controller



- Handles input
 - Not just from the keyboard/mouse/joystick
 - Also, network inputs for multi-player games
 - GPS and accelerometer for mobile games
- **Maps input actions to model or view actions**

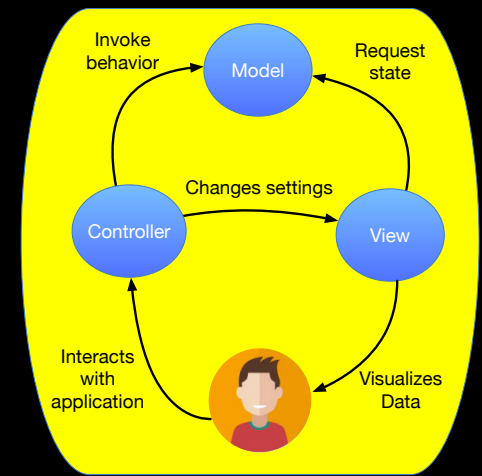
Controller (2)

- **Maps input actions to model or view actions**
- Input actions are things like: mouse clicked, key pressed, network message about a shot fired...
- Model actions are things like: avatar moved left, opponent forfeits, spaceship explodes...
- View actions: window resized, instant replay requested, menu button pressed...

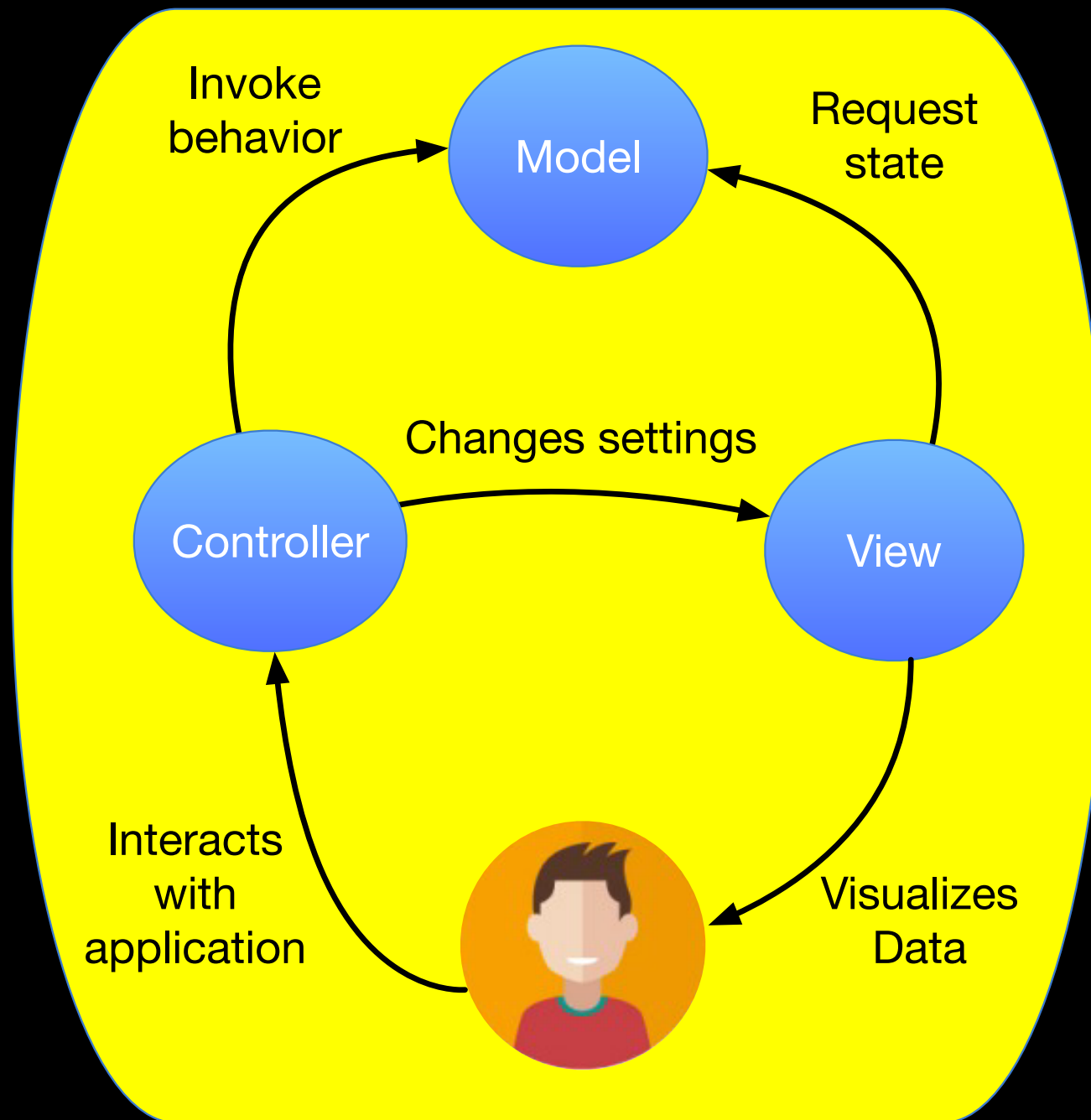
Controller (3)

- Game customizations often result in different mappings for the controller
 - If arrow keys are normal for movement...
 - ... but a left handed player prefers WASD,
 - that's a different mapping (or perhaps a different controller component)

MVC Advice



- Advice: When programming, start with the model
- Figure out how to store data and what data is needed



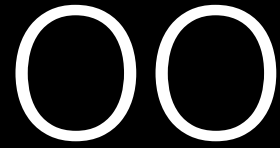
object-oriented model/view

OO

- Another common style is to have objects for the items in your game
 - Pac-Man, Ghosts, fruit, the maze, ...
- Each object knows its own state, so is part of the model
 - Ghost knows its position, how to update (move, turn to "frightened", chase mode, etc), ...
 - A model component may still keep track of all objects and tell them when to update state



- Each object knows how to draw itself, so is part of the view
- A **draw()** method gets called by the view module
- Image is drawn on the screen, purple if frightened, animated correctly, ...



- You still have a model
 - It has references to objects and calls their **update()** method
- You still have a view
 - It has references to objects and calls their **draw()** method...
 - in the correct order so the rendering works properly

graphic primitives

Graphics

- Most games rely heavily on computer graphics
 - Provides visual feedback for players
 - Provides guidance to players

History

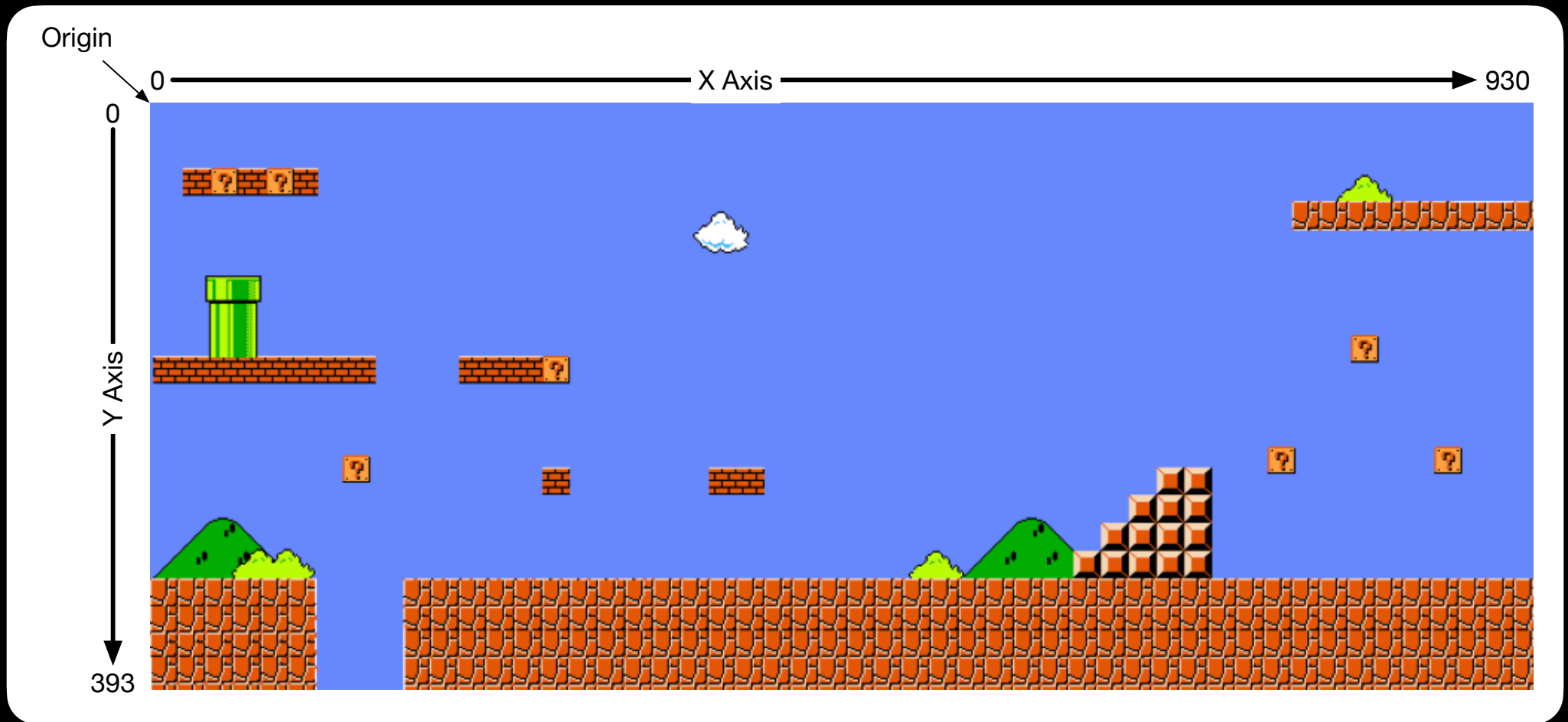
- Development of algorithms for computer graphics has been driven by the gaming industry
- Today's tech for VR / ultrarealistic 3D renders are a result of decades of improvements
- Starting with very first video game ever
 - Spacewar!
 - 1962, PDP-1 minicomputer



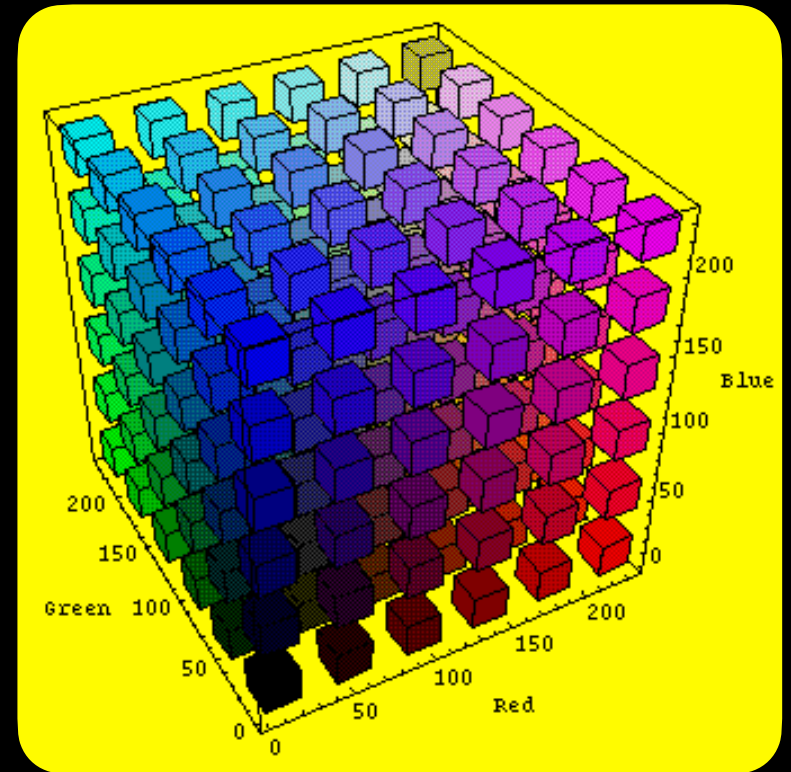
Engine

- An engine provides some level of graphic algorithms to be embedded in your code
 - No need for every game dev to write code for how to draw a line
- Pygame is an engine for 2D games
 - 3D and VR with other engines (Unreal, Unity...)

A Digital Image

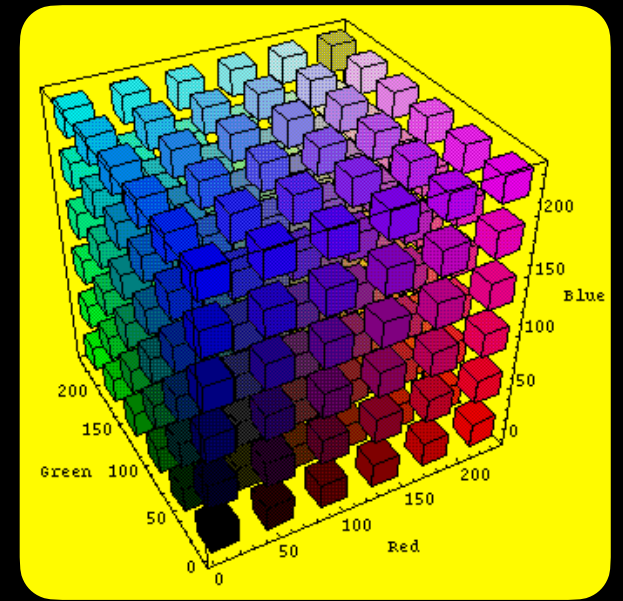


- An image is just an array of pixels
 - Rows and columns
- Each pixel has a color
 - Specified by a number
- Usually formed by mixing 8-bit values (thus 0-255) for red, green, blue and alpha
 - Alpha is often neglected or ignored
 - A system with 8-bits for RGB, known as "24-bit color"

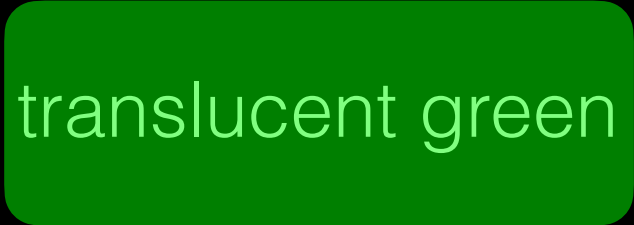


Colors

- RGB with 8-bit values
- In Python and Pygame, the color is a tuple
 - Red = (255, 0, 0)
 - Mario background blue = (112, 136, 242)
- There is also a **Color** class with extra operations
 - Can be specified as 6 hex digits (as a string)
 - Red = '0xFF0000', Mario = '0x7088F2'



Alpha Channels

- An extra 8-bits specifies the amount of transparency
 - Technically, amount of **opaqueness**
- 0 = transparent
- 255 = opaque
- (0, 255, 0, 128) is a translucent green 
- In Pygame, some special handling is required for alpha-channel objects

The Surface

- In Pygame, a **surface** is a place to draw
 - Think of it as a canvas on which an artist places paint
 - Or, as a bank of memory where all those pixels are stored
- Special surface: the **display surface** -- will be visible to the user in the game window

Returned from set_mode method as:

```
display_surface = pygame.display.set_mode((1024, 768))
```

Surface (2)

- Other surfaces are possible and quite useful
- Create a surface to draw a picture
 - Later, or often, can copy the image to the display
- Commonly, have a background surface

```
background = pygame.Surface(display_surface.get_size())
```

- Start every frame render by copying the background into the display surface

Surface Operations

- The Surface object has many possible methods
- Only a few are related to drawing
- The **get_at** and **set_at** methods let you get or set the color of a pixel at a specific location

```
background.set_at((x,y), color)  
color = background.get_at((x,y))
```

- Notice that the location is a tuple


```
# draw_notepaper.py
def draw(surface, size_x, size_y):
    blue = (0, 0, 200)
    red = (200, 0, 0)
    white = (255,255,255)

    for x in range(size_x):
        for y in range(size_y):
            surface.set_at((x,y), white)

    for y in range(60, size_y, 20):
        for x in range(size_x):
            surface.set_at((x, y), blue)

    x = 25
    for y in range(0, size_y):
        surface.set_at((x, y), red)
```

background of the page is white



draw horizontal blue lines

draw one vertical red line

There Must Be A Better Way (TMBABW)

- `draw_notepaper` is PAINFUL, though it works
- `set_at` gets called 251,400 times for a 400x600 sized window
- An entire surface can be set to a single color with the fill function

```
surface.fill( color )
```

- And a line can be drawn between two points with

```
pygame.draw.line(surface, color, (x0, y0), (x1, y1))
```

- Note: endpoints don't have to be contained in the surface dimensions. They will be automatically clipped to only draw the parts inside the surface

A better draw_notepaper.py -- now only 30 method calls!!!

```
def draw(surface, size_x, size_y):
```

```
    blue = (0, 0, 200)
```

```
    red = (200, 0, 0)
```

```
    white = (255, 255, 255)
```

background of the page is white



```
    surface.fill(white)
```

```
    for y in range(60, size_y, 20):
```

draw horizontal blue lines

```
        left_side = (0, y)
```

```
        right_side = (size_x, y)
```

```
        pygame.draw.line(surface, blue, left_side, right_side)
```

draw one vertical red line

```
    pygame.draw.line(surface, red, (25, 0), (25, size_y))
```

Draw module

- In addition to the **line** method, other drawing tools are available in the **draw** module
 - All take a **surface** as a parameter

```
pygame.draw.arc(surface, color, rect,  
                start_angle, stop_angle, width=1)
```

- **arc** will draw an elliptical or circular arc from `start_angle` to `stop_angle` (both in radians 😞)
- Position and dimensions are specified by the `rect` parameter, a bounding box
 - More details on Rect objects coming

More Draw module

```
pygame.draw.lines(surface, color, closed, points, width=1)
```

- **lines** will draw a series of lines, consecutively connecting (x,y) tuples in the **points** list
- If **closed** is **True**, will also connect the last point back to the first

```
pygame.draw.polygon(surface, color, points, width=1)
```

- **polygon** acts similarly, connecting points from a list, but then it fills in the enclosed space

```
pygame.draw.circle(surface, color, center, radius)
```

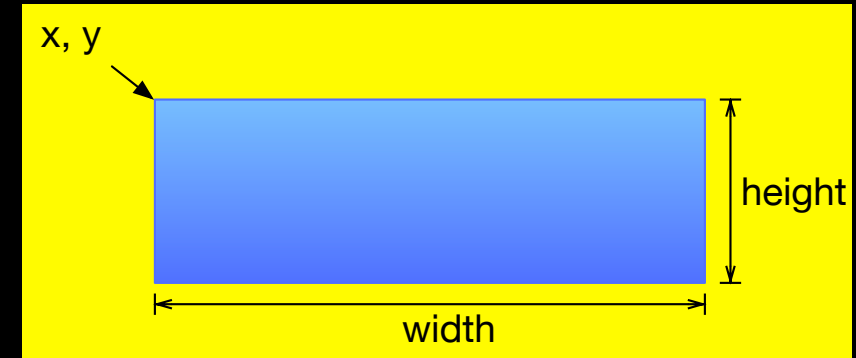
- Drawing a circle is pretty obvious

```
# draw_python.py -- Draw the Python logo  
surface.fill(white)
```

```
points = [(27, 50), (76, 50), (76, 45), (45, 45), (45, 26)]  
pygame.draw.lines(surface, blue, False, points)  
pygame.draw.arc(surface, blue, (45, 15, 61, 22),  
                 0*radians, 180*radians)  
pygame.draw.line(surface, blue, (106, 26), (106, 62))  
pygame.draw.arc(surface, blue, (76, 47, 30, 30),  
                 270*radians, 0*radians)  
pygame.draw.line(surface, blue, (91, 76), (57, 76))  
pygame.draw.arc(surface, blue, (41, 76, 32, 32),  
                 90*radians, 180*radians)  
pygame.draw.line(surface, blue, (41, 92), (41, 110))  
pygame.draw.line(surface, blue, (41, 110), (27, 110))  
pygame.draw.arc(surface, blue, (12, 50, 30, 60),  
                 90*radians, 270*radians)  
.... many more lines of annoyingly measured points
```



Rect



- Pygame very, very commonly uses an object called a **Rect** to describe a box
 - The **arc** function used it as a bounding box
- Rects are defined by four values
 - **Rect(x, y, width, height)**
 - or less commonly, by two pairs of tuples
 - **Rect((x, y), (width, height))**

Rect Methods

<code>pygame.Rect.copy</code>	Copy the rectangle
<code>pygame.Rect.move</code>	Returns a moved rectangle
<code>pygame.Rect.move_ip</code>	Moves the rectangle, in place
<code>pygame.Rect.inflate</code> / <code>_ip</code>	grow/shrink the rectangle size (and an in place version)
<code>pygame.Rect.clip</code>	Crops a rectangle inside another rectangle
<code>pygame.Rect.clipline</code>	Crops a line inside the rectangle
<code>pygame.Rect.union</code> / <code>_ip</code>	Joins two rectangles into one
<code>pygame.Rect.contains</code>	Test if one rectangle is inside another
<code>pygame.Rect.collidepoint</code>	Test if a point is inside a rectangle
<code>pygame.Rect.colliderect</code>	Test if two rectangles overlap
<code>pygame.Rect.collidelist</code>	Test if the rectangle overlaps with any in a sequence of Rects
many others	

Using Rects with Draw

- Back in the Draw module
- We can draw a rectangle, defined by a Rect

```
pygame.draw.rect(surface, color, rect, width=0)
```

- We can draw an ellipse, bounded by a Rect

```
pygame.draw.ellipse(surface, color, rect, width=0)
```

```
# Use Rects for boxes, circles in draw_checkerboard.py
rect = pygame.Rect(strip_size, strip_size, box_size, box_size)
color = black
for row in range(8):
    for col in range(8):
        pygame.draw.rect(surface, color, rect)
        if row in rows_with_pieces and color == black:
            circle_rect = rect.inflate(deflate, deflate)
            pygame.draw.ellipse(surface, white, circle_rect)
        rect.move_ip(box_size + strip_size, 0)
        if color == black:
            color = red
        else:
            color = black
    rect.move_ip(-8*(box_size + strip_size), box_size + strip_size)
    if color == black:
        color = red
    else:
        color = black
```

Make rect for the checkerboard box

Draw the box

Draw ellipse in smaller box

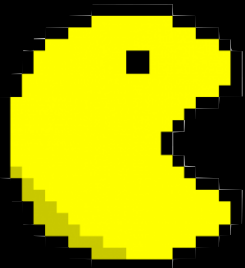
Move the rect to the next place in the row

Move the rect to the first place in the next row

images and blit

Pygame Images

- Much of our game graphics will not be created by calling line, circle, arc, etc
- Instead, you hire a graphic designer to draw game assets as images
- Then we will copy those images to various places on the screen



Optimization



- Think back to Pac-Man
- Pac-Man's movement is very simple
 - Just a circle with a wedge cut out
- But to draw it requires computation (including trig functions!)
- Performance optimization: Do the computation once and use the resulting images for the animation



Image Formats

- There are many, many image formats
 - You've probably seen: JPG, PNG, GIF
- Each standardizes how the color values of the pixel array should be written into a file
- Pygame knows how to load and save
 - JPG, PNG, GIF + BMP, PCX, TGA, TIF, LBM, PBM, PPM and XPM formats

Image Load / Save

- Relatively straightforward in Pygame

```
surface = pygame.image.load(filename)  
pygame.image.save(surface, filename)
```

- You can pass in a string as the filename, or any Python file-like object (ex: **Path** object)

```
pygame.surface.convert()
```

- **convert** will change the format of a surface to match the display surface
 - A good idea for any loaded image, as otherwise it will need to be converted each time it is used

BLIT

- Each image loaded results in a separate surface
 - But, at the end of the game loop, you need a single surface for the display
 - How do you combine surfaces?
- BLIT (or BITBLIT) is an operation in many computing systems that copies a block of memory from one location to another
 - For graphical operations, BLIT "pastes" one image into another

Pygame Blit

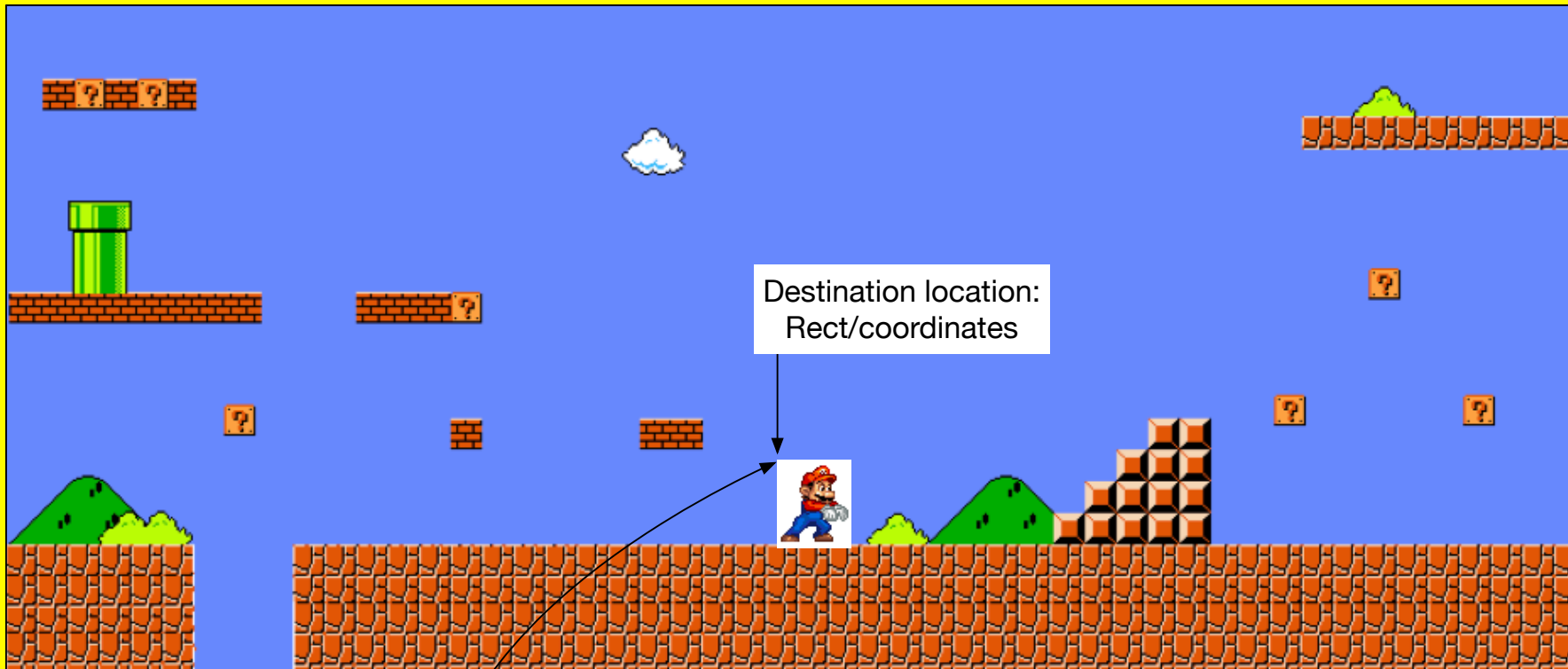
- The **Surface** object has a **blit** method for pasting another image (or portion) into the image

```
surface.blit(source, dest, area=None, special_flags=0)
```

- The **source** parameter is another surface
- **dest** is a location: (x, y) tuple or Rect with the top-left location you want
- **area** can be a Rect that will specify a smaller portion of the source surface to be pasted
 - If **None**, then the entire source surface is pasted

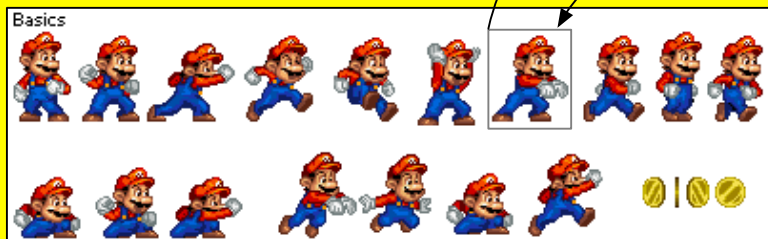
Blit example

Destination surface: background



Source surface: mario

Source Area (a Rect)



```
background.blit(mario, dest_rect, area_rect)
```

```
# render mario
```

```
display_surface = pygame.display.set_mode(1200,622)
```

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

```
mario = pygame.image.load('mario_sprites.png').convert()
```

```
...
```

```
while not done:
```

```
    *user_inputs, done = get_inputs()
```

```
    ...
```

```
    jump_r = pygame.Rect(254, 13, 42, 49)
```

```
    display_surface.blit(background, (0,0))
```

```
    display_surface.blit(mario, (390, 510), jump_r)
```

```
    pygame.display.flip()
```

Blit Optimization

- Graphics hardware often handles the entire BLIT, thus relieving the CPU of lots of byte copies
- Often the blit includes other forms of raster operations
- **special_flags** lets you do more than just paste: ADD, SUB, MULT, MIN, MAX

Colorkey

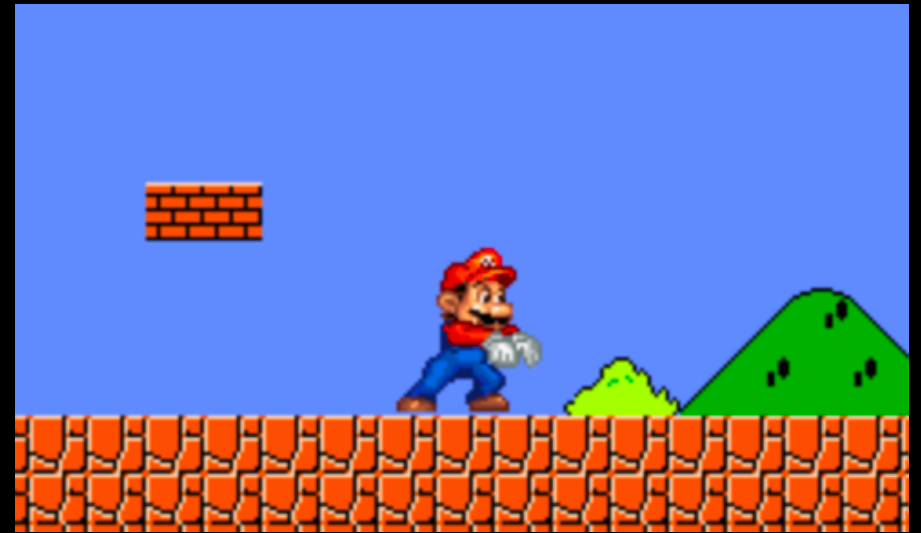
- Other common blit operations include a form of masking to allow "background" pixels not to be overwritten
- To get rid of the white box around Mario, we can set the colorkey for the mario surface
 - Specifies a particular color (white in this case) that will be considered to be transparent
 - During a blit, any white pixel will not be copied
- Could also solve this with alpha channels, but assets must be created with alpha values ahead of time

```
# render mario
```

```
display_surface = pygame.display.set_mode(1200,622)
background = pygame.image.load('mario_background.png').convert()
mario = pygame.image.load('mario_sprites.png').convert()
white = mario.get_at((0,0))
mario.set_colorkey(white)
```

```
...
while not done:
    *user_inputs, done = get_inputs()
    ...
```

```
jump_r = pygame.Rect(254, 13, 42, 49)
display_surface.blit(background, (0,0))
display_surface.blit(mario, (390, 510), jump_r)
pygame.display.flip()
```



Surface Transforms

- `pygame.transform` module has methods to manipulate a surface
- You can rotate / scale / chop / greyscale /
- For example:

```
pygame.transform.rotate(source_surface, angle) → Surface
```

What did you learn today?

- Games are a part of human nature
 - No surprise that humans put them on a computer
- Game architectures are important
 - For this class, always watch out for the game loop

What did you learn today?

- Pygame engine is a library with lots of useful routines
 - lines, colors, circles, images, ...
 - events
 - surfaces