# Animation & Interactivity

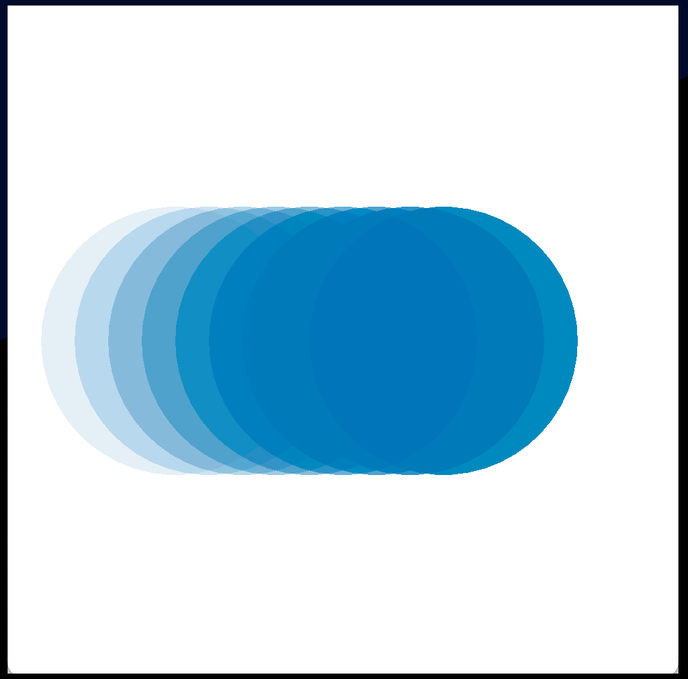
# Learning Objectives

- Write PennDraw programs that create moving images
- Get introduced to a basic form of iteration with the "infinite while loop"
- Write programs that react in real time to user inputs
  - Understand how to check for & use mouse input with PennDraw
  - Understand how to check for & use keyboard input with PennDraw

# Animation in PennDraw

# Animation as a Model

- For static images, we picked the positions, sizes, & colors of our shapes **once**.
- Smooth animation is achieved by showing a lot of similar images very quickly
  - A "flipbook" model
  - Screen is updated ~30 times per second
  - Shapes will change position, size, color slightly with each update



# Frames

- Animations as "flipbooks" require that we draw many images per second
  - Call these images **frames**
  - More frames per second *("higher FPS")* ➡️ smoother animation
- A frame consists of a set of shapes rendered at a specific moment in time
  - Different frames typically have the same shapes but drawn in different ways

# Drawing Frames in PennDraw

Animations usually include a *setup* and an *animation loop.*

```
import penndraw as pd
# SETUP: This code is run just one time!
pd.set_canvas_size(500, 500)
x_center = 0.5
y_center = 0.5
half_side = 0.1
pd.set_pen_color(pd.HSS_BLUE)
# ANIMATION LOOP: This code is run many times per second,
# over and over and over and over again.
while True:
    pd.clear()
    pd.filled_square(x_center, y_center, half_side)
    x_center += 0.01
    if x_center > 1 + half_side:
        x_center = -half_side
    pd.advance() # Necessary at the end of the loop
```

# Animating: the Setup

Animated programs usually start with a "setup" block, where we:

- choose settings, like canvas size
- declare variables that we will use to draw each frame of the animation
  - variables will vary, but we can pick their initial values (deciding how the animation starts)
- do anything that needs to happen **only one time.**

```
import penndraw as pd
pd.set_canvas_size(500, 500)
x_center = 0.5
y_center = 0.5
half_side = 0.1
pd.set_pen_color(pd.HSS_BLUE)
```

# Animating: the `while True` Loop

`while`, like `if`, is a keyword that allows us to control the flow of a program.

```
while expression:
  do_this()
  do_that()
```

When we reach the `while`, we test its condition. If `True`, we execute the statements in the body. Then, we **test the condition again**.

- Different from a conditional (`if`), where we only test ONCE!
- If the `expression` is literally `True`, we will loop here forever...

# Animating: the `while True` Loop

The body of the `while` loop is our *animation loop*:

- runs many times per second
- runs indefinitely until the program is manually stopped
- ➡️ allows us to draw many frames per second, doing something slightly different each time.

# Animating: Animation Loop Recipe

For each frame,

1. decide whether to clear the screen
    i. Clearing the screen ➡️ all previous shapes disappear, only most recent shape is visible
    ii. Not clearing ➡️ most recent frame is drawn on top of other frames, which might still be visible
2. draw the next frame based on current properties of the shapes
    i. "properties of the shapes" usually stored in variables
3. update the properties of the shapes for the next frame
4. `pd.advance()` ➡️ make everything show up on screen
    i. Always need this at the end of the loop.

# Example Animation Loop: Sliding Square

Produces a square that slides left-to-right across the canvas.

```
import penndraw as pd
x_center = 0.5 # SETUP
while True:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)        # 2. draw this frame
    x_center += 0.01                            # 3. update shapes for next frame
    pd.advance()                                # 4. pd.advance()
```

# Example Animation Loop: Sliding Square

Modify the program to include print statements, time tracking:

```
import penndraw as pd
x_center = 0.5 # SETUP
loop_num = 1
while True:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)  # 2. draw this frame
    print(f"In Loop #{loop_num}, square is at x={x_center}")
    x_center += 0.01                            # 3. update shapes for next frame
    pd.advance()                                # 4. pd.advance()
```

# Example Animation Loop: Sliding Square

Modify the program to include print statements, time tracking:

```
import penndraw as pd
x_center = 0.5 # SETUP
loop_num = 1
while True:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)  # 2. draw this frame
    print(f"In Loop #{loop_num}, square is at x={x_center}")
    x_center += 0.01                            # 3. update shapes for next frame
    loop_num += 1
    pd.advance()                                # 4. pd.advance()
```

🖨️👇

```
In Loop #1, square is at x=0.5
In Loop #2, square is at x=0.51
In Loop #3, square is at x=0.52
...
```

# Controlling the Slide

```
import penndraw as pd
x_center = 0.5 # SETUP
while True:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)        # 2. draw this frame
    x_center += 0.01                            # 3. update shapes for next frame
    pd.advance()                                # 4. pd.advance()
```

After a while, `x_center` will be very big.

- If `x_center` is much bigger than `1.0`, the square won't be visible at all!
- Want to add some logic to make sure the square resets after a while

# Controlling the Slide

What does "off the screen" mean?

- Happens when a shape is all the way off the top, bottom, left, or right sides of the screen.
- If the square is always heading to the right, it'll fall off the right side
- Since the square has a `half_length` of `0.1`, the coordinate of its *left side* is always `x_center - 0.1`.
- ➡️ the square is offscreen when `x_center - 0.1 > 1.0`

# Controlling the Slide

Making the square reset to the left once it disappears:

- After we update the square, check if it's offscreen
- If the square is offscreen, move it all the way to the left of the screen
- If the square is not offscreen, don't do anything else extra

```
import penndraw as pd
x_center = 0.5 # SETUP
while True:
    pd.clear()                                  # 1. clear the screen
    pd.filled_square(x_center, 0.5, 0.1)        # 2. draw this frame
    x_center += 0.01                            # 3. update shapes for next frame
    if x_center - 0.1 > 1.0:
      x_center = -0.1
    pd.advance()                                # 4. pd.advance()
```

# Animation: Advancing

You always need a call to `pd.advance()` at the end of your animation loop. Otherwise nothing shows up.

# Mouse Input

# Clicking into Place

PennDraw includes a few tools useful for handling cursor position & clicking:

| Function | Return Type | Description |
|---|---|---|
| `pd.mouse_pressed()` | `bool` | Returns `True` if the mouse is being held this frame. |
| `pd.mouse_x()` | `float` | Returns the **x coordinate** of the mouse's current location, e.g. `0.9` or `0.1443` |
| `pd.mouse_y()` | `float` | Returns the **y coordinate** of the mouse's current location, e.g. `0.9` or `0.1443` |

# Click Counter

```
import penndraw as pd
num_clicks = 0;
while True:
  pd.clear()
  pd.text(0.5, 0.5, f"Number of Clicks: {num_clicks}")
  if pd.mouse_pressed():
    num_clicks = num_clicks + 1
  pd.advance()
```

*Each frame, if we click the mouse, increment `num_clicks`.*

# Following Square

```
import penndraw as pd

pd.set_canvas_size(500, 500)
x_center = 0.5
y_center = 0.5
half_side = 0.1
pd.set_pen_color(pd.HSS_BLUE)

while True:
  pd.clear()
  x_center = pd.mouse_x()   # Ask for the x-coordinate of the cursor
  y_center = pd.mouse_y()   # Ask for the y-coordinate of the cursor
  if pd.mouse_pressed():    # Ask whether the mouse is being clicked
    pd.set_pen_color(pd.HSS_RED)
  else:
    pd.set_pen_color(pd.HSS_BLUE)
  pd.filled_square(x_center, y_center, half_side)
  pd.advance()
```

# Keyboard Input

# "Keys" to Success

User key presses can also be registered!

| Function | Return Type | Description |
| --- | --- | --- |
| `pd.has_next_key_typed()` | `bool` | Returns `True` if a key is currently being held |
| `pd.next_key_typed()` | `str` | Returns the key currently being held down. |

🚨 Don't use `next_key_typed()` without checking `has_next_key_typed()` first! 🚨

# Checking Keys

The value produced by `pd.next_key_typed()` is a string with a length of one: just a single character.

```
key = pd.next_key_typed()
```

- To see if a specific key was pressed:
  - `if key == "x": ...` or `if key == "!": ...` e.g.
- To see if the key was a lowercase letter:
  - `if "a" <= key <= "z": ...`
- To see if the key was a digit:
  - `if "0" <= key <= "9": ...`

# Light Switch

```
import penndraw as pd
on = False
while True:
  if on:
    pd.clear(pd.BLACK)
  else:
    pd.clear(pd.YELLOW)

  if pd.has_next_key_typed():
    key = pd.next_key_typed()
    if key == "x":
      on = not on
  pd.advance()
```