# Advanced Programming #3: Animation

CS SCHOLARS – PROGRAMMING

# Learning Goals

Create **time-based animations** within the interaction framework

# Event-Based Interaction

In class we discussed **event-based interaction**, where we built programs that responded to user mouse and keyboard events and modified a Tkinter view to produce interactive behavior.

We can also build programs that changes graphics **over time**. In other words, we can build programs that produce moving graphics, or animation!

# New Controller: timerFired

We'll add one new controller to our interaction framework – a **time** controller. This controller will repeatedly call a rule function (`timerFired`) at regular intervals.

If we change the state of the model in `timerFired`, that will cause the model to constantly change as time passes. This can lead to fluid movement!

# Example: Moving Circle

To create a simple animation, we use the same approach as in `mousePressed` and `keyPressed` – just change a value in the model and use that changed value in `makeView`.

For example, to make a circle move constantly to the right, put this in `timerFired`:, then use `data.x` in `makeView`:

```
data.x = data.x + 5 # move the circle to the right
```

# Example: Bouncing Circle

Eventually the circle will move all the way off the screen. By making `timerFired` more complex, we can change the animation to make the circle bounce instead!

Store **two** pieces of information in the model – the current x position *and* the current direction. When the ball would overlap with the edge of the screen, reverse the direction.

Note that we compare the left side of the ball to the left edge of the screen, and the right side of the ball to the right side of the screen

```
def init(data):
    data.x = 200
    data.dx = 10 # delta x

def redrawAll(canvas, data):
    canvas.create_oval(data.x - 20, 200 - 20,
                       data.x + 20, 200 + 20,
                       fill="red")

def timerFired(data):
    # if new position would be offscreen
    if (data.x + data.dx - 20) < 0 or \
       (data.x + data.dx + 20) > 400:
        # reverse direction
        data.dx = - data.dx
    data.x = data.x + data.dx
```

# Activity: Bounce Vertically

**You do:** try adding some code so that the ball bounces vertically as well as horizontally. To make the code extra interesting, make the vertical movement rate different from the horizontal rate!

# Activity Answer

```
def init(data):
    data.x = 200
    data.dx = 10
    data.y = 200
    data.dy = 15

def redrawAll(canvas, data):
    canvas.create_oval(data.x - 20, data.y - 20, data.x + 20, data.y + 20, fill="red")

def timerFired(data):
    if (data.x + data.dx - 20) < 0 or (data.x + data.dx + 20) > 400:
        data.dx = - data.dx
    data.x = data.x + data.dx

    if (data.y + data.dy - 20) < 0 or (data.y + data.dy + 20) > 400:
        data.dy = - data.dy
    data.y = data.y + data.dy
```

# Rate Changes Update Speed

The current ball movement might not feel very smooth. How can we make it look better?

You can change the **update rate** in the original call to `runSimulation`. This rate is the length in seconds between calls to `timerFired`.

By default we set the rate to 0.1 (10 times per second). If you make the number smaller, the function will be called more times per second (for example, 0.05 would be called every 0.05 seconds, 20 times per second); this makes the animation smoother. If you make the number larger, it will be called less often and the animation will move slower.

In Tkinter, you can't go any faster than 1ms (1/1000 second) between calls. Realistically, you won't see much improvement beyond 10ms (1/100 second) either.

# How the Time Loop Works

How does this work, anyway?

The **time** controller in the function `timeLoop` calls our function `timerFired`, then calls `redrawAll` to update the view. It simulates a time loop with the built-in function `canvas.after`. This function calls `timeLoop` again (like an infinite loop) but pauses before making the call. That lets us repeat infinitely without freezing the window.

The function `runSimulation(width, height, rate)` sets up the initial time loop. It runs itself behind the scenes after that.

# Example: Bubble Popping Game

Let's put animation together with user interaction to make a game! Specifically, we want to make a **bubble-popping game**, where new bubbles appear on the screen constantly and the user tries to pop them by clicking on them.

First, we need to determine what must be stored in the model:

- Information about the bubbles – position, size, color
- Score – how many bubbles have been popped?

# Step 1: Generate Bubbles

First, let's just make an animation that makes bubbles appear on the screen at regular intervals.

Each bubble will be represented as a `[x, y, size, color]` list, like we did in the activity in class.

The default rate is too fast! Change it to `0.5` instead of `0.1`.

```
def init(data):
    data.bubbles = []

def redrawAll(canvas, data):
    for bubble in data.bubbles:
        [x, y, size, color] = bubble
        canvas.create_oval(x - size, y - size,
                           x + size, y + size,
                           fill = color)

def timerFired(data):
    # Make a new bubble
    size = random.randint(10, 25)
    # Make sure bubbles are not offscreen
    x = random.randint(size, 400-size)
    y = random.randint(size, 400-size)
    color = random.choice(["red", "green", "blue"])
    data.bubbles.append([x, y, size, color])
```

# Step 2: Click on Bubbles

To have the user click on the bubbles, we need to detect which bubble (if any) was clicked.

Iterate through the list of bubbles and use the **distance formula** to check whether the distance between the clicked location and the center of the circle is within the radius (size) of the circle. If it is, remove that circle from the list and `return` to exit the loop.

Removing the circle from the list also removes it from the screen!

```
def mousePressed(event, data):
    for bubble in data.bubbles:
        [x, y, size, color] = bubble
        dist = ((x - event.x)**2 + \
                (y - event.y)**2)**0.5
        if dist <= size:
            data.bubbles.remove(bubble)
            return
```

# Step 3: Keep Track of Score

Finally, store a score in the model, display it in `makeView`, and update it when a bubble is removed in `mousePressed`.

Now we have a proper game!

```
def init(data):
    data.bubbles = []
    data.score = 0

def redrawAll(canvas, data):
    for bubble in data.bubbles:
        [x, y, size, color] = bubble
        canvas.create_oval(x - size, y - size,
                           x + size, y + size,
                           fill = color)
    canvas.create_text(200, 20,
                       text="Score: " + str(data.score),
                       font="Arial 20 bold")

def mousePressed(event, data):
    for bubble in data.bubbles:
        [x, y, size, color] = bubble
        dist = ((x - event.x)**2 + \
                (y - event.y)**2)**0.5
        if dist <= size:
            data.bubbles.remove(bubble)
            data.score += 1
            return
```

# Learning Goals

Create **time-based animations** within the interaction framework