# Physics Simulator Lab - Parts 1-4

Welcome to the physics simulator project! Over the next few labs, you'll build a physics engine that can simulate realistic motion. But before we get to bouncing balls and gravity, we need to understand how animation works.

## Part 1: The Stationary Circle

### From Static Pictures to Moving Images

You've been using StdDraw to create static images - you draw some shapes, call `StdDraw.show()`, and you're done. Animation is different. An animation is really just a sequence of static images shown quickly one after another, like a flipbook. Our program will create these images in a loop, displaying 30 frames per second.

## The Animation Loop

The heart of any animation is the **animation loop** - a while loop that runs forever, creating each frame. Every time through the loop represents one frame of your animation. Here's what needs to happen in each frame:

1. **Clear the screen** - Erase the previous frame so you can draw the new one
2. **Draw everything** - Draw all the objects in their current positions
3. **Show the frame** - Display what you just drew
4. **Pause** - Wait a tiny bit so the animation runs at the right speed

Think of it like drawing a flipbook: you need to erase the old drawing, draw the new one, and then flip to show it.

## Understanding the Variables

Let's look at the important variables in the starter code:

- `fps` - Frames per second. We're using 30, which means 30 images per second, a broadcast standard
- `delta_t` - The time between frames in seconds (1/fps second)
- `width` and `height` - The size of our canvas in pixels. The coordinate system goes from (0,0) in the bottom-left to (640, 480) in the top-right
- `frame` - Counts which frame we're on (0, 1, 2, 3, ...)
- `time` - The total elapsed time in seconds

**Important note about `delta_t`:** Notice we wrote `1.0 / fps` instead of `1 / fps`. Why? Because `1 / fps` uses integer division in Java, which would give us `0` (Java throws away the remainder). By writing `1.0` with a decimal point, we're telling Java "this is a double," so it performs decimal division and gives us `0.0333...` like we want. This is one of those quirks of Java you need to watch out for!

For now, you won't need to use `frame` or `time` - but they'll be important soon!

## Part 1: Your Task

You're going to complete the animation loop. The starter code has everything set up except for the body of the while loop. Your job is to fill in the loop to:

1. Clear the background (use white or your favorite color)
2. Draw a filled circle on the left side of the screen, somewhere near the middle vertically
3. Choose any radius you like for your circle
4. Show the frame you just drew
5. Pause for the right amount of time - notice that `StdDraw.pause()` requires an integer number of milliseconds, so we use `1000 / fps` (not `1000.0 / fps`). Since both 1000 and fps are integers, Java does integer division here, which is what we want. (Does this integer division introduce any approximation error in our timing?)

The circle should be stationary for now - it will appear in the same place every frame. Even though it's not moving, you'll see the power of the animation loop structure.

## Part 1: Starter Code

```
public class JustACircle {

    public static final int fps = 30;
    public static final double delta_t = 1.0 / fps;
    public static final int width = 640;
    public static final int height = 480;

    public static void run() {
        StdDraw.setXscale(0, width);
        StdDraw.setYscale(0, height);
        StdDraw.setCanvasSize(width, height);
        StdDraw.enableDoubleBuffering();

        int frame = 0;
        double time = 0;

        // main animation loop
        while (true) {
            frame++;
            time += delta_t;

            // YOUR CODE GOES HERE

            StdDraw.pause(1000 / fps);
        }
    }

    public static void main(String[] args) {
        run();
    }
}
```

## Part 1: Testing Your Code

When you run your program, you should see a window open with a stationary circle. It might seem boring - after all, the circle isn't moving! But you've just created your first animation loop. The circle is being redrawn 30 times per second, which is the foundation for all the physics simulations to come.

**Note**: To stop your program, you'll need to close the StdDraw window or stop it from your IDE, since the while loop runs forever.

---

## Part 2: Motion

Now let's make the circle actually move! Right now, you probably have the x and y coordinates of your circle hard-coded as numbers in your `filledCircle()` call. To create motion, we need to make those positions change over time.

### Step 1: Create Position Variables

Before the animation loop (but after the setup code), create variables to store your circle's position. Give them meaningful names like `ball_x` and `ball_y`, or `circle_x` and `circle_y`, or whatever makes sense to you. Initialize them to wherever you want your circle to start.

### Step 2: Use Variables in Your Drawing

Update your `filledCircle()` call to use these variables instead of hard-coded numbers.

### Step 3: Update Position Each Frame

Inside your animation loop, update the x position each frame to make the ball move horizontally. You'll need to figure out how to change x based on the `frame` variable. Experiment with different approaches - you'll discover that you can control the speed of motion by how you update x.

**A thought about boundaries:** What happens when your circle reaches the edge of the screen? Does it disappear off into the void? You might want to use an `if` statement to detect when the circle goes past `width`, or perhaps the modulo operator `%` could create some interesting wrapping behavior.

### Step 4: Creative Motion

Now it's time to experiment! Try modifying both x and y to create different types of motion. Here are some ideas to get you started:

- Make the circle move diagonally
- Make it speed up or slow down over time
- Create a parabolic arc (like a thrown ball)
- Make it wrap around the screen (hint: use `%` to create repeating patterns)
- Try using `Math.sin()` or `Math.cos()` with the frame number to create smooth curves or circular paths
- Combine multiple functions with x and y to create complex patterns

The key insight: by using mathematical functions of `frame`, you can create all sorts of motion patterns. Play around and see what you can discover.

### Testing Part 2

When you're done, you should see your circle moving around the screen in whatever pattern you programmed. Try tweaking the numbers in your motion equations to see how they affect the animation.

### What's Next?

In the next part, we'll start thinking about physics - using velocity and acceleration instead of directly controlling position. That's when things get really interesting.

---

## Part 3: Real Physics - Velocity!

Now we're going to transition from frame-based motion to **physics-based motion**. Instead of directly changing the position based on frame numbers, we'll use velocity - the rate at which position changes over time. This is how real physics works.

### Understanding Units

Before we start, let's talk about units. In our simulation: - **Distance**: 1 pixel = 1 cm - **Time**: We already have `delta_t` which is 1/30 second per frame - **Velocity**: pixels per second (which is the same as cm/s in our physics model)

So if a ball has a velocity of 60 pixels/second, it will move 60 pixels down in one second, or 2 pixels per frame (since each frame is 1/30 second).

### Step 1: Start Fresh

Comment out or remove any motion code from Part 2. We're starting over with a physics approach. Set your x position to a constant value near the middle of the screen - the ball will only move vertically now. Also position your ball at the top of the screen.

### Step 2: Create Velocity Variables

Before your animation loop, create a `double` variable `v_y` for the vertical velocity. Initialize it to a negative value (like -60 or -100) since negative y velocity means moving downward.

### Step 3: Update Position Using Kinematics

Inside your animation loop, you'll use the kinematic equation: **displacement = velocity × time**

Each frame, you need to: 1. Calculate the displacement: $\Delta y = v\Delta t$ 2. Update the position: $y_t = y_{t-1} + \Delta_y$ 3. Make sure you convert these math equations to proper code.

This is the fundamental equation of motion under zero force. The ball's position changes based on only its velocity and the elapsed time.

### Step 4: Experiment with Velocity

Run your program and watch the ball fall! Try different values of `v_y`: - What happens with `v_y = -30`? - What about `v_y = -200`? - What if you make `v_y` positive?

Notice how the speed is constant - the ball doesn't speed up as it falls. That's because we haven't added acceleration (yet)

### Step 5: Stop at the Floor

You'll notice the ball falls right off the bottom of the screen. Add an `if` statement in your loop to check if the ball has reached the floor (y <= 0 or some radius threshold). When it hits the floor, we'll exit the `while` loop, at least for now.

Here's how to modify your animation loop: - define a `boolean done = false;` - Change the while loop condition from `while (true)` to `while (!done)` - When the ball hits the floor, set `done = true` instead of trying to stop the ball's motion - This will make the loop end cleanly when the ball hits the ground.

### Testing Part 3

When you're done, you should see your ball fall from the top of the screen at constant velocity and stop when it hits the bottom.

**Experimental verification:** Let's check if your simulation is actually correct! Add some print statements to verify the physics: - Print the `time` and `y` position at the very beginning (frame 0) - Print the `time` and `y` position after the ball hits the ground (after the `while` loop) - Use these values to compute the actual speed your ball traveled (hint: speed = distance / time) - Print both the computed speed and the velocity you programmed (`v_y`) - Do they match? If not, why might they be different?

This is an important habit in scientific computing - always verify your simulation matches what you expect!

## What's Next?

In Part 4, we'll add **acceleration** - specifically gravity. That's when the ball will speed up as it falls, just like in the real world. We'll implement a full Euler integration step to update both velocity and position.

---

## Part 4: Acceleration and Gravity

Welcome to the exciting part - **real falling motion**! In Part 3, your ball fell at constant velocity. But real objects don't fall at constant speed - they accelerate due to gravity. Now we'll add acceleration to make the simulation realistic.

### Understanding Acceleration

Acceleration is the rate at which velocity changes over time. Earth's gravity causes objects to accelerate downward at about 980 cm/s². Since we're measuring distances in pixels (and treating 1 pixel = 1 cm), gravity will be 980 pixels/s².

### Step 1: Add Gravity

At the top of your program with the other constants, add:

```
public static final double g = 980.66;  // acceleration due to gravity (pixels/s²)
```

Notice it's negative because gravity pulls downward (toward y = 0).

### Step 2: Set Up Initial Conditions

Before your animation loop: - Create a variable `a_y` and set it equal to `-g` (negative!) - Change your initial `v_y` to `0` - the ball starts at rest this time - Position the ball at the top of the screen (like before) - Keep the boolean variable called `done` and initialize it to `false` - we'll use this to end the simulation when the ball hits the ground

### Step 3: The Euler Integration Step

This is the heart of physics simulation. Each frame, you need to update BOTH velocity and position using these equations:

1. $\Delta v = a\Delta t$
2. $v_t = v_{t-1} + \Delta$
3. $\Delta y = v_t \Delta t$
4. $y_t = y_{t-1} + \Delta y$

This is called **Euler integration** - we're approximating continuous motion by taking small discrete time steps. Notice that we update velocity first (using acceleration), then update position (using the new velocity).

Translate these equations into code in your animation loop.

#### Step 4: Watch It Fall!

Run your program. You should see the ball start slowly and then speed up as it falls - just like a real falling object! The motion should look much more natural than the constant-velocity fall from Part 3.

#### Step 5: Experimental Verification

Time to check if your simulation matches the physics. You'll verify one of the fundamental kinematic equations that you're learning in physics class.

**After your animation loop ends** (not inside it), add code to: - Print `time` and `y` at the very beginning (frame 0) - Print `time` and `y` when the ball hits the ground - Use these values to compute the actual distance the ball fell - Use the time and the value of `g` to compute what the distance SHOULD be according to the kinematic equation for distance under constant acceleration (look this up in your physics notes!) - Print both the actual distance and the theoretical distance - How close are they? What might cause any differences?

This verification is crucial - it shows whether your simulation is physically accurate.

**Extra experiment:** Try changing `fps` to different values (like 10, 60, or 120). Does the frame rate affect the accuracy of your simulation? Why or why not?

## What's Next?

Congratulations! You've built a basic physics simulator with proper acceleration. In future parts, we could add bouncing, multiple objects, different forces, or even 2D motion. You now have the foundation for simulating all kinds of physical systems!