Physics Simulator Lab - Day 2: Projectile Motion

Yesterday you built a simulator for a ball dropping under gravity. Today we'll extend it to handle full 2D projectile motion - launching a ball with both horizontal and vertical velocity components.

Part 1: Launch the Projectile

Getting Started

Copy your working code from Day 1 to a new file (maybe call it `ProjectileLauncher.java`). You're going to modify it to simulate launching a projectile from the ground. (You can keep this file in the same IntelliJ project; just be sure to select "Run Current File" from the Run options).

Step 1: Set Up Initial Conditions

Before your animation loop, set up your initial conditions. In addition to changing y and its velocity and acceleration, you now have to keep up with a changing x, and its velocity v_x. (There is no x acceleration since there is no force but gravity.)

Position: - Start with `x = 0` (left edge of the screen) - What should `y` be if the ball is sitting on the ground? Think about the ball's radius!

Velocity: - Give the ball an initial horizontal velocity `v_x` (try something like 200 pixels/second) - Give the ball an initial vertical velocity `v_y` (try something like 400 pixels/second to launch it upward) - Experiment with different values to see different trajectories!

Acceleration: - You already have `a_y = g` for gravity - Create `a_x = 0` (no horizontal acceleration for now)

Step 2: Update the Animation Loop

Modify your animation loop to update BOTH x and y positions each frame. You'll need to apply the Euler integration equations to both dimensions:

For x-direction: 1. $\delta x = v_x \delta t$ 2. $x = x + \delta x$

For y-direction: (You already have this from Day 1)

Step 3: Fix the Landing Detection

Your landing detection from Day 1 probably stops the simulation when `y <= radius` or similar. But now you're starting on the ground! You need to make sure the simulation doesn't stop immediately.

Add a check that the ball is moving downward before stopping the simulation. Something like: only stop when `y <= radius` **AND** `v_y < 0`. This ensures the ball has gone up and come back down before we detect the landing.

Step 4: Enhanced Output

After your animation loop ends (when the ball lands), print out all the important information about the flight:

Print: - Time in air (total elapsed time) - Horizontal distance traveled (final x - initial x) - Initial position: x, y - Initial velocity: v__x, v__y - Final position: x, y - Final velocity: v__x, v__y

This data will help you analyze whether your simulation is working correctly!

Let’s Test Part 1

Run your program with different initial velocities. You should see nice parabolic arcs! The ball should launch upward, reach a peak, and come back down to land further along the ground.

Part 2: Track Maximum Altitude

Now let’s find the highest point of the trajectory.

Step 1: Track the Maximum

Create a variable `max_y` before your animation loop and initialize it to your starting y value. Inside your loop, update `max_y` whenever the ball reaches a new height:

```
if (y > max_y) {
    max_y = y;
}
```

Step 2: Report It

After the animation loop ends, add the maximum altitude to your output. Print it along with the other flight data.

Let’s Test Part 2

The maximum altitude should occur somewhere in the middle of the flight. Does it make sense given your initial v__y?

Part 3: Analysis - Energy Conservation

Now for the interesting part: analyzing whether your simulation conserves energy correctly.

The Question

Look at your output, specifically at the initial and final `v_y` values. In a perfect world with no air resistance, what should be the relationship between initial `v_y` and final `v_y`?

Think about it: the ball starts on the ground going up with some `v_y`, and lands back on the ground going down. What does energy conservation tell you?

Check Your Results

Are they exactly equal (ignoring the sign)? Probably not! There’s likely a small difference. Why?

Try this experiment: Increase `fps` to 60, or even 120. Run your simulation again. Does the difference between initial and final `v_y` get smaller?

What’s Happening?

There are two sources of error in your simulation:

1. **Frame rate discretization**: With 30 fps, you're taking relatively large time steps (1/30 second). The simulation has to approximate what happens during that time. Higher frame rates mean smaller time steps and better approximation.
2. **Linear approximation in Euler integration**: When you calculate $\delta y =$ ‘v_y‘Öδt, you're assuming the velocity is constant during that time step. But velocity is actually changing continuously due to acceleration! This introduces a small error each frame.

These errors accumulate over the flight, which is why your final `v_y` might not exactly match your initial `v_y`.

Discussion question: Is the final `v_y` larger or smaller in magnitude than the initial `v_y`? What does this tell you about whether your simulation is gaining or losing energy?

We'll come back to this issue later and learn about better integration methods that reduce these errors!

What's Next?

You now have a working 2D projectile simulator! Continue on to explore some interesting physics questions below.

Part 4: Physics Experiments

Now that your simulator is working, let's use it to explore some classic physics problems!

Experiment 1: Launch Angle and Speed

So far, you've been setting `v_x` and `v_y` directly. Let's think about launch parameters the way physicists do: as a **magnitude** (speed) and **angle**.

Before your animation loop, define: - `speed` - the initial speed in pixels/second (try 500) - `theta` - the launch angle in degrees (try 60)

Then compute the velocity components: - `v_x = speed * cos(theta)` - `v_y = speed * sin(theta)`

Important: Java's trig functions use radians, not degrees! You'll need to convert theta to radians first using `Math.toRadians(theta)`.

Test it out - does a 45-degree launch give you the classic parabolic arc?

Experiment 2: Optimal Angle for Maximum Range

Here's a classic physics question: **What launch angle gives you the maximum horizontal distance?**

Keep your `speed` constant (like 500) and try different values of `theta`. Record the horizontal distance traveled for each angle.

What angle gives you the farthest distance? Does this match what you learned in physics class?

Now try a different speed (like 300 or 700). Does the optimal angle change, or is it always the same?

Experiment 3: Launching from a Height

Real projectiles often launch from above ground level. Modify your initial `y` position to start higher up (try `y = 100` or `y = 200`).

Now what angle gives you the maximum range? Is it still the same as when launching from the ground?

Try different heights. Does the optimal angle depend on the launch height? Why or why not?

Experiment 4: Precision Challenge

Make your canvas wider using `StdDraw.setCanvasSize(1200, 480)` and `width = 1200`.

Now the challenge: **Can you get the ball to land exactly in the opposite corner?** - Start at (0, 0) (or (0, radius) to be precise) - Try to land at (1200, 0)

You'll need to find the right combination of speed and angle. How close can you get?

Experiment 5: Target Practice

Here's a fun challenge: hit a specific target point in mid-air!

Setup: 1. Choose a target point somewhere above the ground - maybe (400, 200) or wherever you like 2. Draw the target as a small filled circle (use a different color so you can see it) 3. Draw it every frame so it stays visible

Challenge: Find a launch angle and speed that makes your ball hit the target! You'll need to experiment with different combinations.

Extension - Automatic Hit Detection: Right now you have to watch and see if you hit the target. Let's make the computer check!

Inside your animation loop, check each frame whether the ball has contacted the target point. Think about what "contact" means geometrically - the ball is a circle with radius `r`, and the target is a point. When are they touching? (Hint: the distance formula might be helpful!)

When a hit is detected: - Print a victory message ("Target hit!") - Maybe print the time of impact and the coordinates - Optionally: stop the simulation or change the ball's color

Can you write code that automatically detects when you've successfully hit the target?

Bonus exploration: Try different target positions. Do you notice that some targets are impossible to hit no matter what angle and speed you try? What determines whether a target is reachable?

Reflection

Through these experiments, you've explored: - Converting between Cartesian and polar representations of velocity - Optimization problems (finding the angle that maximizes range) - How initial conditions affect trajectories - Geometric collision detection

These are the kinds of problems that come up in everything from video games to rocket science. You're doing real physics and real computational problem-solving!