Physics

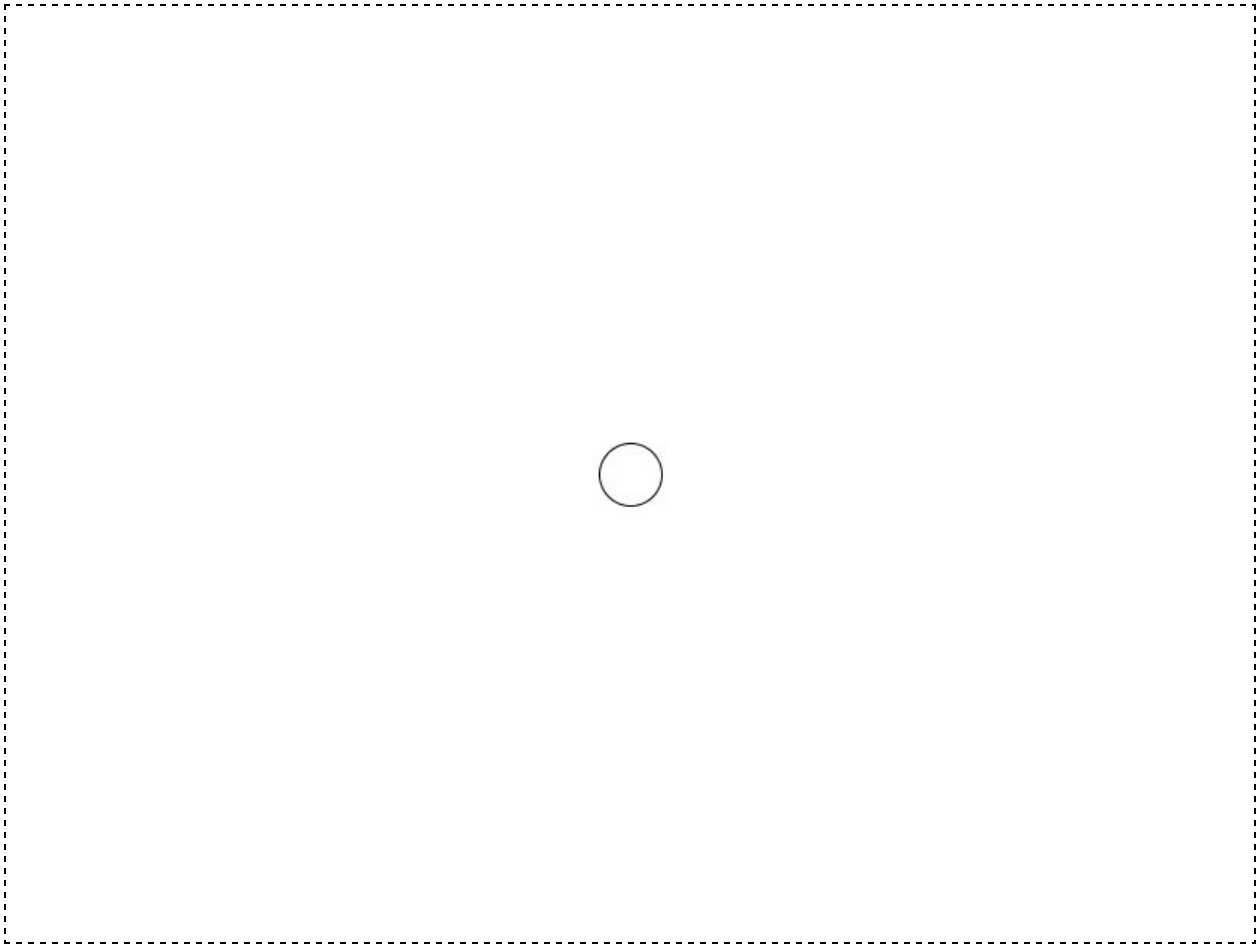
Bouncing Ball

Overview

This project is intended as an introduction to game physics. In it, you'll create a ball that will bounce around the screen. It may seem like a simple outcome, but the concepts introduced here are applicable to a wide range of game mechanics.

Getting Started

To get started, download the **BouncingBall** project from GitHub and open it with Processing. It's pretty simple to start out with! If you hit play, you should see something like



Awesome, right? Game of the year material right there. Anyway, let's look at the code we have so far...

The first three lines tell us that our program needs to remember three numbers in order to run. The ball on the screen has an X position (how far to the right it is), a Y position (how far down it is), and a radius (how big it is).

```
float ballPositionX;
float ballPositionY;
float ballRadius;
```

The next three lines define the Processing **setup** function, which is run once as soon as the sketch starts. All we do in setup right now is set the size of the sketch window to be 800 pixels wide and 600 pixels tall, and initialize the values of the three variables.

```
void setup() {
  size(800, 600);
  ballPositionX = 400;
  ballPositionY = 300;
  ballRadius = 20;
}
```

The next three lines define the Processing **draw** function, which is run every frame. In this function, we first draw a white background, and then draw the ball using the Processing ellipse function.

```
void draw() {
  background(255);
  ellipse(ballPositionX, ballPositionY, ballRadius * 2, ballRadius * 2);
}
```

Nothing too crazy happening here. If you want, play around with changing the values of **ballPositionX**, **ballPositionY**, or **ballRadius** and see how it affects the sketch.

Velocity

The first step towards getting the ball bouncing is getting the ball moving. In physics, the motion of an object is called its *velocity*. Velocity describes how much an object's position changes in a period of time. In the real world, we often measure velocity in units like *miles per hour* or *feet per second*. In Processing, we'll be measuring velocity in units of *pixels per frame*. Because our Processing sketch is two-dimensional, velocity has two components. They describe how much an object's position changes left-to-right, as well as up-and-down. Declare two **float** variables to hold onto that information.

```
float ballVelocityX;
float ballVelocityY;
```

In **setup**, initialize these variables to some values, say **5** and **-2** for example.

Now that the variables for velocity have been declared and initialized, it's time to actually use them! Each frame (every time **draw** is run), we want the ball's position in each axis to change by its velocity in that axis. This can be accomplished with two lines of code! Add the following to the beginning of the **draw** function.

```
ballPositionX = ballPositionX + ballVelocityX;
ballPositionY = ballPositionY + ballVelocityY;
```

Each frame, we set the value of **ballPositionX** to be the same as its value last frame, but with **ballVelocityX** added to it. Same for **ballPositionY** and **ballVelocityY**. Try running your sketch now! If you've done it correctly, the ball should move! Unfortunately, it probably quickly moves right off the edge of the screen...

Bouncing Off The Walls

What should have happened? If we imagine the world of the Processing sketch as a room with walls at the edges of the window, you might expect that the ball would bounce off of them. But what exactly do we mean when we say *bounce off*. Specifically, we mean the ball's *velocity* should change when it's *position* is at the edge of the screen.

This could happen at any time, so we'll have to check for it every single frame. In order to have something happen only sometimes, we can use an **if** statement. After the ball's position is updated, we need to check and see *if* it has moved out of the window. There are four different ways this could happen (too far left, too far right, too far up, and too far down), and we'll need to check for all of them.

Once we've found out if the ball needs to bounce this frame by checking its position, how should its velocity be updated? If it has gone off the left or right edge, we should reverse its **x** velocity. If it has gone off the top or bottom edge, we should reverse its **y** velocity.

```
if (ballPositionX > width) {
  ballVelocityX = -ballVelocityX;
}

if (ballPositionX < 0) {
  ballVelocityX = -ballVelocityX;
}

if (ballPositionY > height) {
  ballVelocityY = -ballVelocityY;
}

if (ballPositionY < 0) {
  ballVelocityY = -ballVelocityY;
}
```

Once you've added this code to the draw method after the position update step, hit play again. If you've done it correctly, the ball should now deflect off the edges of the window! But, something might look a little off. The ball seems to bounce not when it first hits the edge, but after it's already slid off the screen somewhat.

Our math for checking collision with the walls is not quite correct. The **ballPositionX** and **ballPositionY** variables describe the position of the *center* of the ball. We want to check when the *edge* of the ball has hit a wall. We have to include the *size* of the ball in our calculations! A smaller ball's center position will be closer to the wall when it bounces than a larger ball.

A better block of code for collision checking is the following. Notice that we *add* the radius when checking the "too far right" and "too far down" conditions (positive **x** and **y** directions), but *subtract* it when checking the "too far left" or "too far up" conditions (negative **x** and **y** directions).

```
if (ballPositionX + ballRadius > width) {
  ballVelocityX = -ballVelocityX;
}

if (ballPositionX - ballRadius < 0) {
  ballVelocityX = -ballVelocityX;
}

if (ballPositionY + ballRadius > height) {
  ballVelocityY = -ballVelocityY;
}

if (ballPositionY - ballRadius < 0) {
  ballVelocityY = -ballVelocityY;
}
```

Run the sketch again. If everything is correct, the ball should now bounce against the edge of the screen properly! Pretty cool, right? Try changing the initial velocities, and see how that affects the sketch.

Gravity

You've gotten pretty far now. At this point you've got the groundwork written for quite a few games whose driving mechanic is a bouncing ball. Think arcade games like Breakout or Pong (look them up if you haven't heard of them!) If you want, you could leave this project at this point and start turning this sketch into something like that.

If not, let's press forward with more physics!

In the real world, objects don't just fly around rooms, bouncing off the walls. They are affected by the force of *gravity*. Gravity changes the velocity of moving objects, directing it more towards the ground. Similar to how we use the word *velocity* to describe the change in position over a period of time, we use the word *acceleration* to describe the change in velocity over a period of time. You may be familiar with that word already, but you might think of it just as *speeding up*. When an object speeds up, it is *accelerating* because its velocity is increasing. In physics, we also refer to objects which are slowing down or changing direction as *accelerating*, because these are also changes in velocity.

On Earth (or on any planet), gravity is directed towards a single point: The center of the planet itself! In Processing, we'll simplify that a bit and imagine that gravity is always pointing straight down (in the direction of increasing **y** coordinates).

Adding gravity to our sketch is only a few lines of code. First we'll need to remember another number: How strong gravity is in the world of our sketch. Declare another global **float** variable to hold onto this information.

```
float gravity;
```

In the **setup** function, initialize this variable to a small value like **0.2**.

Now that our gravity variable has been declared, and initialized, we have to actually use it to modify the velocity! *Before* the position update, add a single line to change the ball's velocity in the **y** direction.

```
ballVelocityY = ballVelocityY + gravity;
```

Run the sketch again. If everything's correct, the ball should now be pulled towards the bottom of the screen as if by the force of gravity! However depending on how strong gravity is, and how fast the ball is going, you may notice something wrong...

Stuck In The Floor!

If you haven't noticed the bug, try changing the initialization of the following variables in **setup** to these values

```
ballVelocityX = 5;
ballVelocityY = -2;
gravity = 0.5;
```

After five or six bounces, something very weird happens and the ball seems to stick to the bottom of the screen and shake around a lot before sliding out of view. Definitely not what you'd expect to happen if a real ball was dropped! You may have seen bugs like this in real video games, particularly 3D video games where the physical calculations the

computer must do can be very complicated. Sometimes, characters or objects seem to glitch through walls or other objects in an unrealistic manner.

Thankfully, we can fix the problem! But first we need to understand why it's happening. It's kind of subtle, and when I first wrote this assignment I didn't understand what was going on either! Debugging is one of the most valuable (but also one of the most difficult, and often time consuming) skills to develop as a programmer. In order to track down this bug, let's think through one frame of the sketch step by step. We know the problem happens when the ball bounces off the bottom of the screen, so let's start there.

We're in **draw** function. Imagine the ball is traveling downwards. At the beginning of a frame, we apply an acceleration to its velocity in the **y** direction, making the value of that velocity larger. The ball is now traveling downwards *more rapidly* than before. Then, we change the position of the ball by adding its velocities in each direction.

Because we know the bug happens during a bounce, let's assume that this position change put it over the bottom edge. When that happens, we reverse the direction of the **y** velocity. Previously positive (since the ball was moving downwards in the positive-**y** direction), the **y** velocity is now negative. Now at the end of the **draw** function, we set the white background and actually put an ellipse on the screen at the right location and the right size.

Nothing seems wrong yet...

On to the next frame. Back to the beginning of the **draw** function. The velocity in the **y** direction is now negative because the ball is moving upwards. Adding gravity to it makes it *less* negative. In other words, the ball is now moving upwards *more slowly*. Now we update the ball's position by adding its velocities in each direction. Then, check to see if the ball has gone off the edge of the screen.

Oh, wait...

What if, because of the slow-down due to gravity, the ball is not moving fast enough to get back to being fully on the screen during this frame? This means even after moving upwards a little bit, the bottom edge of the ball is still too far down. Then, our **if** statement gets triggered again! The **y** velocity gets flipped for a second frame in a row and now the ball is moving downwards again - even *further* off the screen.

And there it is. The source of the problem. If the ball is slowed enough by gravity that it's still off the edge even after "bouncing," it will keep having its **y** velocity flip-flopped every single frame thereafter, producing the weird stuck-in-the-floor jittering effect that we've observed. The first step of debugging is done now because we know exactly what is causing the problem; but how to fix it?

In the real world, objects never overlap at all -- reality doesn't have *frames* like a Processing sketch does. We need to make sure that the ball can never be off the edge of the screen two frames in a row, which means we need to cheat physics a little bit. One solution to our bug is to move the ball *just* back into the window every time a collision is detected.

Exactly how far should we move it? The same distance it's overlapping the wall. It took me some pencil-and-paper sketching to understand exactly how to calculate this distance. Draw it out yourself if you don't understand. Then modify the **if** statements to the following

```
if (ballPositionX + ballRadius > width) {
  ballPositionX = ballPositionX - (ballPositionX + ballRadius - width);
  ballVelocityX = -ballVelocityX;
}

if (ballPositionX - ballRadius < 0) {
  ballPositionX = ballPositionX + (ballRadius - ballPositionX);
  ballVelocityX = -ballVelocityX;
}

if (ballPositionY + ballRadius > height) {
  ballPositionY = ballPositionY - (ballPositionY + ballRadius - height);
  ballVelocityY = -ballVelocityY;
}

if (ballPositionY - ballRadius < 0) {
  ballPositionY = ballPositionY + (ballRadius - ballPositionY);
  ballVelocityY = -ballVelocityY;
}
```

In each **if** block, I've added a line of code to change the ball's position by how much it's overlapping the edge of the screen. This way it can never be off the edge two frames in a row, which fixes the stuck-in-the-floor glitch.

Elasticity

All game physics are simplified versions of the laws of mechanics that real world objects follow. Depending on what your game needs, your simulated physics may need to be more or less accurate. One example of an additional level of accuracy we could add is *elasticity*. The word *elasticity* refers to how *bouncy* something is. Notice that in our sketch, the ball always bounces back up to the same height, and keeps bouncing forever.

This isn't quite realistic. Actual balls bounce a little bit lower each time, until they hit the floor and either roll away or stop moving altogether. The amount of height lost in each

bounce is related to the *elasticity* of both the ball and the walls. We *could* write a very detailed simulation of the mathematics of elastic collisions, but we could also add a little tweak to our math that gets us most of the way there. Often times in game physics, it's not important to be physically accurate as long as what happens *looks right*.

Declare a final **float** variable to store how bouncy the ball and walls are.

```
float elasticity;
```

Initialize its value to between **0** and **1**, for example **0.9**.

Now, when we compute the new velocity following a collision with the walls, multiply the reversed value by the value of **elasticity**. For example,

```
ballVelocityX = -ballVelocityX * elasticity;
```

Run the sketch again. If everything's done right, the ball should lose a little height with each bounce. Of course, when it finally stops bouncing altogether, it just glides around the bottom of the screen as if on ice. This is because our world doesn't have *friction*, which is the force that slows things down if they're in contact with each other. I won't go into how to implement that in this exercise, but if your game needs it, it might be worthwhile to look into!

Extra Credit

It's been a long and mathematical journey to get to this point, so now let's have a little fun. To add some flair to your bouncing ball program, replace the lines of code that actually draw the backround and the ball with the following to add a sweet trail effect!

```
fill(255, 70);
noStroke();
rect(0, 0, width, height);
fill(255);
stroke(0);
ellipse(ballPositionX, ballPositionY, ballRadius * 2, ballRadius * 2);
```

Neat, huh? Feel free to try some other things, like setting the value of **gravity** to a negative number, or **elasticity** to something greater than **1**. How big can the ball be before the sketch stops working right? How fast can the initial velocity go?

Summary

What you've just written is a simulation of a type of physics called *kinematics*. It's a long, fancy word for "stuff that moves." We learned how to update the position of an object by adding its velocity, how to make that object collide with walls, how to update the velocity

of an object by applying an acceleration, how to debug a physics glitch, and a couple ways to increase the realism and visual effects of our application.

What's Next?

Making a bouncing ball might not feel like much, but kinematic physics like this are crucial to hundreds, even thousands of video games, from very simple games like Asteroids or Doodle Jump to complicated 3D worlds like Grand Theft Auto or Rocket League. What kinds of games can you imagine that start here?

If you find simulating real world physics interesting, there's a whole wide world of it in games that you could implement, from increasingly realistic collision simulations which add support for *friction*, *rotation*, or *air resistance* to entirely different concepts like *optics* (the way in which light interacts with objects), *fluid mechanics* (the physics of things like water and smoke), and more. The math can get pretty complex, of course.

If you found this tedious and boring, don't worry! Many of these problems have already been solved by other programmers, which means that if you don't want to do them yourself, you don't necessarily have to. There exist *libraries* for physical calculations of various kinds in 2D and 3D that you can add to your projects so that you don't have to do all of this yourself! Of course, you still have to learn to use those libraries.