The Pragmatic Programmers

Advanced Hands-on Rust

Herbert Wolverson

edited by Tammy Coron

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 8

Build Obstacles and Collision Detection

Many games require *collision detection*—a process of detecting when one object has "hit" another object. For example, in Flappy Dragon, the dragon crashes into walls. In space shooter games, lasers obliterate monsters, and the monsters' weapons damage the player. In physics games, collision detection is used to determine bounce and movement. The list is endless. Collision detection is important, and it's a good service to offer in a game making library.

At the moment, Flappy Dragon is very inefficient in detecting collisions. On each frame, the distance between Flappy and every wall is calculated. If the distance is sufficiently small, a collision has occurred and the game ends. Flappy Dragon can get by with inefficient collision detection because there aren't very many potentially colliding objects. However, increasing the game's complexity would lead to a rapid drop in performance.

In this chapter, you'll follow a common journey for the library developer. You'll identify a performance problem and construct a test-bed, focused on the particular problem you need to solve. You'll then incrementally test algorithmic improvements, graphing the results. After you have a good solution, you'll convert it into generic library-friendly code and update Flappy Dragon to use it.

Let's get started by building a collision test-bed.

Building a Collision-Detection Test-bed

Often, when you're optimizing one element of a game or engine it's helpful to create a test-bed that emphasizes the problem you are trying to optimize. Rather than add thousands of objects to Flappy Dragon—and then take them away again, it's easier to build an example designed to stress the feature you are optimizing. You're going to set Flappy to one side for a moment, and build

a billiards table with an easy way to add more and more balls. The balls collide with one another, and the test-bed reports statistics—allowing you to *measure* your optimization progress.

Let's build a billiards table in hyperspace. Balls spawn on the table in a random position, with a random velocity. If balls collide, they're propelled away from the ball with which they collide. The table wraps at the edges—if a ball goes off one side of the screen, it reappears on the opposite side of the screen.

The basic framework of the simulation is similar to the other games you've created, so rather than fill pages with code listings, you can download the code from the accompanying source folder: code/FlappyCollision/bouncy. There are some parts of the program that differ from what you've written before, so let's look at them before moving on to the next step.

The show_performance() uses Bevy's Diagnostics plugin to obtain performance details:

FlappyCollision/bouncy/src/main.rs

```
fn show_performance(
  mut egui_context: egui::EguiContexts,
1 diagnostics: Res<DiagnosticsStore>,
  mut collision_time: ResMut<CollisionTime>,
  mut commands: Commands,
  mut rng: ResMut<RandomNumberGenerator>,
  assets: Res<AssetStore>,
  query: Query<&Transform, With<Ball>>,
  loaded_assets: Res<LoadedAssets>,
) {
2 let n_balls = query.iter().count();
3 let fps = diagnostics
    .get(FrameTimeDiagnosticsPlugin::FPS)
    .and_then(|fps| fps.average())
    .unwrap();
  collision_time.fps = fps;
  egui::egui::Window::new("Performance").show(
    egui_context.ctx_mut(),
    |ui| {
4     let fps_text = format!("FPS: {fps:.1}");
5     let color = match fps as u32 {
        0..=29 => Color32::RED,
        30..=59 => Color32::GOLD,
        _ => Color32::GREEN,
      };
      ui.colored_label(color, &fps_text);
      ui.colored_label(
        color,
        &format!("Collision Time: {} ms", collision_time.time),
      );
      ui.label(&format!("Collision Checks: {}", collision_time.checks));
```

```
        ui.label(&format!("# Balls: {n_balls}"));
6       if ui.button("Add Ball").clicked() {
          println!(
            "{n_balls}, {}, {}, {:.0}",
            collision_time.time, collision_time.checks, collision_time.fps
          );
          spawn_bouncies(1, &mut commands, &mut rng, &assets,
            &loaded_assets);
        }
        if ui.button("Add 100 Balls").clicked() {
          println!(
            "{n_balls}, {}, {}, {:.0}",
            collision_time.time, collision_time.checks, collision_time.fps
          );
          spawn_bouncies(100, &mut commands, &mut rng, &assets,
            &loaded_assets);
        }
        if ui.button("Add 1000 Balls").clicked() {
          println!(
            "{n_balls}, {}, {}, {:.0}",
            collision_time.time, collision_time.checks, collision_time.fps
          );
          spawn_bouncies(1000, &mut commands, &mut rng, &assets,
            &loaded_assets);
        }
      },
    );
}
```

❶ Request Bevy's Diagnostics type as a resource.

❷ Count the number of balls being simulated.

❸ Diagnostics provides a number of statistics. You're requesting the frames-per-second count as an average of recent frames. An average is useful because the actual number can fluctuate.

❹ Format the FPS to one decimal place.

❺ Change the FPS color to indicate "good," "warning," or "terrible" using the colors green, orange, and red, respectively.

❻ Add buttons to the user interface to add more balls to the simulation. Provide increments of 1, 100 and 1,000 balls.

bounce_on_collision() is another useful function. When two balls collide, it calculates the vector between them and sends a ball away from the object with which it collided. The function runs for both balls, providing relatively realistic—but not perfect—collision:

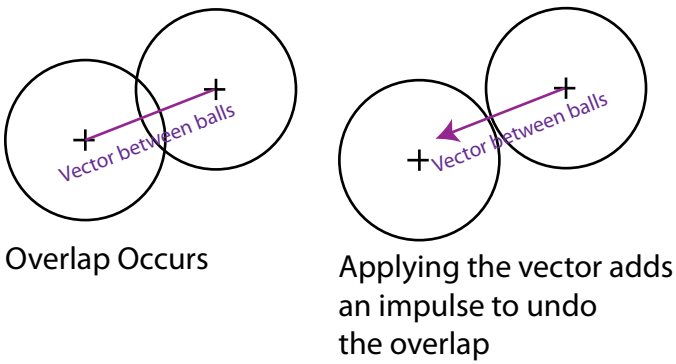
FlappyCollision/bouncy/src/main.rs

```
fn bounce_on_collision(
  entity: Entity,
  ball_a: Vec3,
  ball_b: Vec3,
  impulse: &mut EventWriter<Impulse>,
) {
  let a_to_b = (ball_a - ball_b).normalize();
  impulse.send(Impulse {
    target: entity,
    amount: a_to_b / 8.0,
    absolute: false,
  });
}
```

❶ Bevy's `Vec3` type implements the `Sub` trait, allowing you to subtract one vector from another. Normalizing is an operation that scales the total distance represented by the vector (from zero) to 1.0.

❷ Rather than try to calculate a perfect elastic collision—a topic on which whole books have been written—you're applying a force based on the direction of the collision.

You can illustrate the `bounce_on_collision()` function's effect like this:



Since you're applying an *impulse*—not an absolute force—momentum is preserved, and the result is a convincing bounce.

Finally, the `collisions()` queries each ball against every other ball, and detects if they overlap:

FlappyCollision/bouncy/src/main.rs

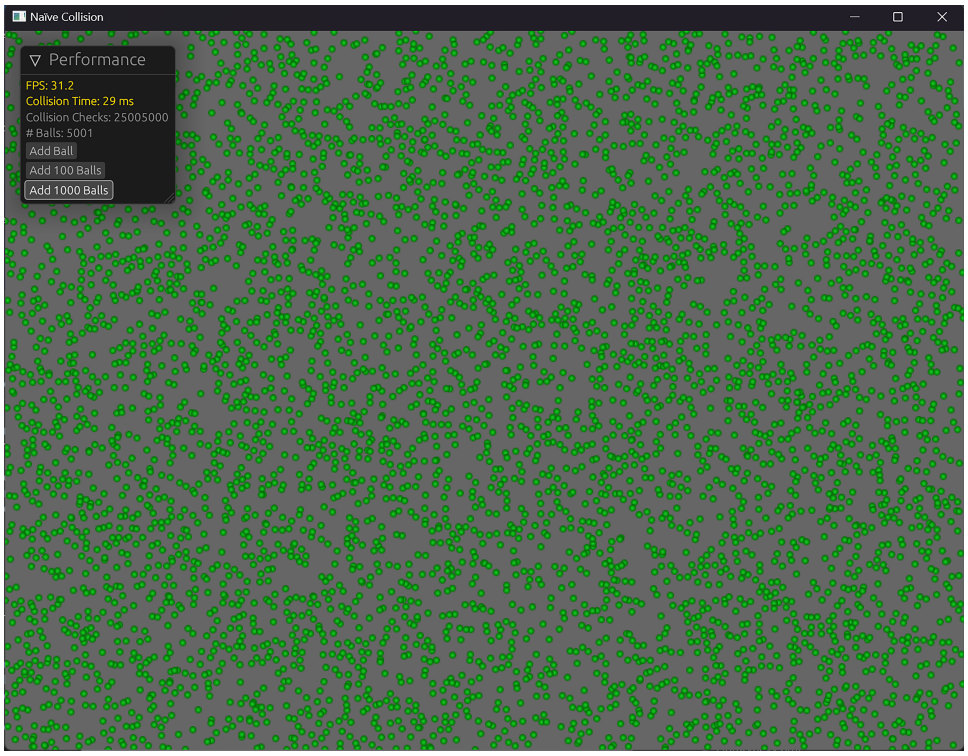
```
fn collisions(
  mut collision_time: ResMut<CollisionTime>,
  query: Query<(Entity, &Transform), With<Ball>>,
  mut impulse: EventWriter<Impulse>,
```

```
) {
  // Start the clock
  let now = std::time::Instant::now();

  // Naïve Collision
  let mut n = 0;
  for (entity_a, ball_a) in query.iter() {
    query
      .iter()
      .filter(|(entity_b, _)| *entity_b != entity_a)
      .filter(|(_, ball_b)| {
        n += 1; // Count the collision check
        ball_a.translation.distance(ball_b.translation) < 8.0
      })
      .for_each(|(_, ball_b)| {
        bounce_on_collision(
          entity_a,
          ball_a.translation,
          ball_b.translation,
          &mut impulse,
        );
      });
  }

  // Store the time result
  collision_time.time = now.elapsed().as_millis();
  collision_time.checks = n;
}
```

The collision function is equivalent to the collision detection used in Flappy Dragon. Let's start by running the example:



Now that you have a working test-bed, you can begin to analyze and optimize the collision detection system.