Unit 04c: Refactoring

Introduction

Refactoring is the process of analysing your existing code, and finding better ways of writing it. Often you'll do this after you first get some functionality to work -- once you've hacked it together to work, you go back and figure out if it's working well, and how you can improve the code to be more functional.

Goal

The goal of this unit is to introduce ways to improve and refactor code.

Process

Variable Refactoring

If we take a look at our existing code:

```
          // check if we use absolute or relative motion
          if (useAbsoluteMotion)
          {
              // use absolute
              if (usePhysicalMotion)
              {
                  // use forces
                  GetComponent<Rigidbody>().AddForce(horizontalInput *
physicalSpeed, 0f, verticalInput * physicalSpeed);
              } else {
                  // use translate, no forces
                  transform.Translate(horizontalInput, 0f, verticalInput,
Space.World);
              }
          } else {
              // use relative
              transform.Translate(horizontalInput, 0f, verticalInput);
```

```
        }
```

you can see we repeat the code fragment `horizontalInput, 0f, verticalInput`. This little piece of code is basically a Vector3 that we create to move the player, and the fact that it repeats means there is a chance we can create a bug by editing it in one place, but not the other.

So instead of writing it twice, we are going to refactor the code into a variable. That way, we can just edit the variable in one place, and every time that variable is used it'll use the edited variable.

1. Right after we poll the Input system for keystrokes, create a new Vector3 variable:

```
        verticalInput = Input.GetAxis("Vertical") * Time.deltaTime *
translateSpeed;

        // Create movement Vector3 variable in refactor
        Vector3 movementDirection = new Vector3(horizontalInput, 0f,
verticalInput);

        // check if we use absolute or relative motion
```

This new variable, `movementDirection`, is a *local* variable. This means it's a variable we create inside of a method, and once the method is done, Unity deletes that variable. Unlike the *class* variables we create at the top of the script, local variables are not useable outside of the method they're created in.

2. Now we need to replace the old, repeated code with our new variable:

```
        // check if we use absolute or relative motion
        if (useAbsoluteMotion)
        {
            // use absolute
            if (usePhysicalMotion)
            {
                // use forces
                GetComponent<Rigidbody>().AddForce(horizontalInput *
physicalSpeed, 0f, verticalInput * physicalSpeed);
            } else {
                // use translate, no forces
                // transform.Translate(horizontalInput, 0f, verticalInput,
Space.World);
                transform.Translate(movementDirection, Space.World);
            }
        } else {
            // use relative
            // transform.Translate(horizontalInput, 0f, verticalInput);
            transform.Translate(movementDirection);
        }
```

And check to see if it's still working in Unity. It's definitely easier to read in the code – I'd rather read `transform.Translate(movementDirection)` than `transform.Translate(horizontalInput, 0f, verticalInput)`.

3. Okay, but what about the version that uses forces? It's still using the same input values, but multiplies the x and z values by our `physicalSpeed` variable. Can we refactor this as well? Of course we can! You can multiply an entire Vector3 by a float or an int, and each element in that Vector3 gets multiplied by that amount. Luckily, in this case, the y value is 0, and when multiplied still stays 0. So we can replace the line:

```
        GetComponent<Rigidbody>().AddForce(horizontalInput * physicalSpeed, 0f, verticalInput * physicalSpeed);
```

With:

```
        GetComponent<Rigidbody>().AddForce(movementDirection * physicalSpeed);
```

Which is also much easier to read!

4. If we wanted to get super crazy, we could also replace these lines:

```
        // Debug.Log(Input.GetAxis("Horizontal"));
        horizontalInput = Input.GetAxis("Horizontal") * Time.deltaTime * translateSpeed;
        verticalInput = Input.GetAxis("Vertical") * Time.deltaTime * translateSpeed;

        // Create movement Vector3 variable in refactor
        Vector3 movementDirection = new Vector3(horizontalInput, 0f, verticalInput);
```

With this one line:

```
        Vector3 movementDirection = new Vector3(Input.GetAxis("Horizontal"), 0f, Input.GetAxis("Vertical")) * Time.deltaTime * translateSpeed;
```

But sometimes spelling things out clearly is better than optimising too much too early.

Method Refactoring

There is another way we can refactor -- by creating new Methods. Our code isn't complicated enough (yet) to justify it, but we'll take a look at this later.

Wrap-Up

Refactoring is excellent practice, and another example of how game development involves iterative cycles. In this case, the cycle goes:

> Make it work → check for improvements → refactor → make something new work

Further Material

If you're really keen on it, you can check out a huge list of refactoring practices at The Refactoring Catalog. It presupposes a lot of prior knowledge, but take a poke around.