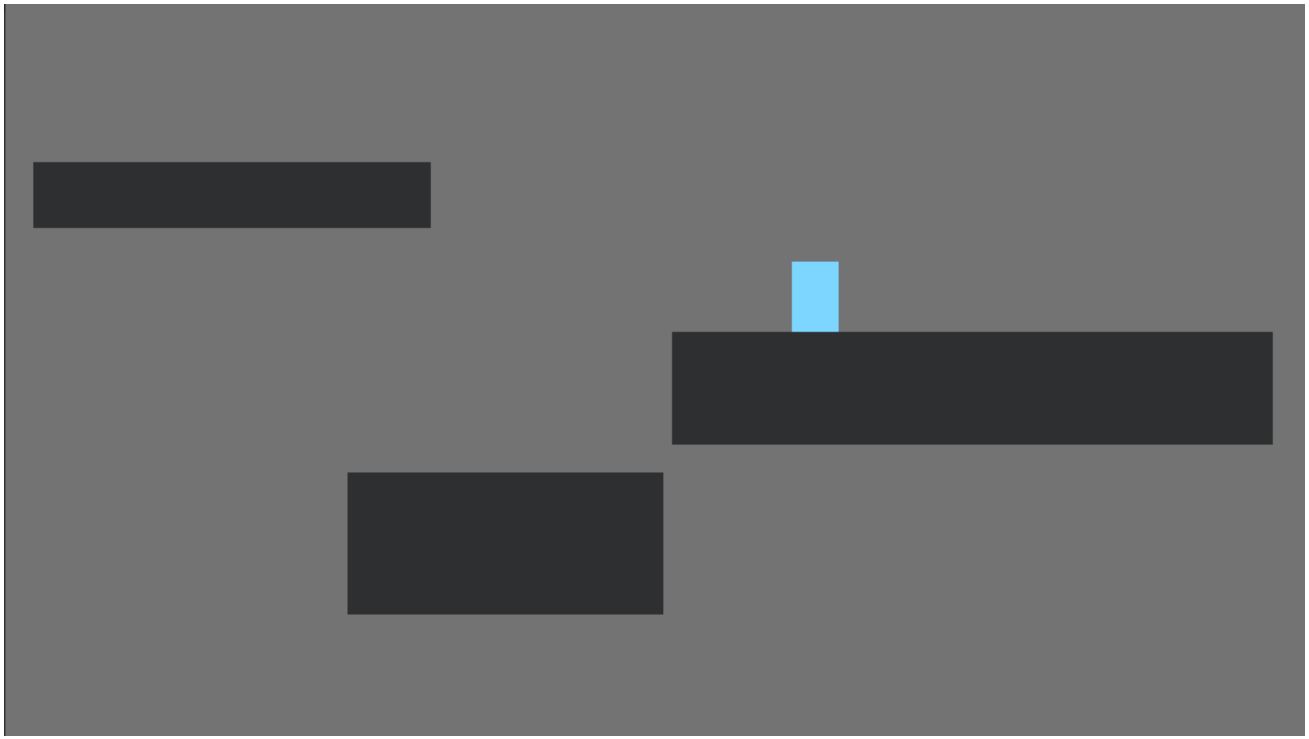
Simple 2D Character Controller

>> using Unity engine 2018.x



You will learn to build a 2D character controller for a platformer game using custom physics—no rigidbodies or forces.

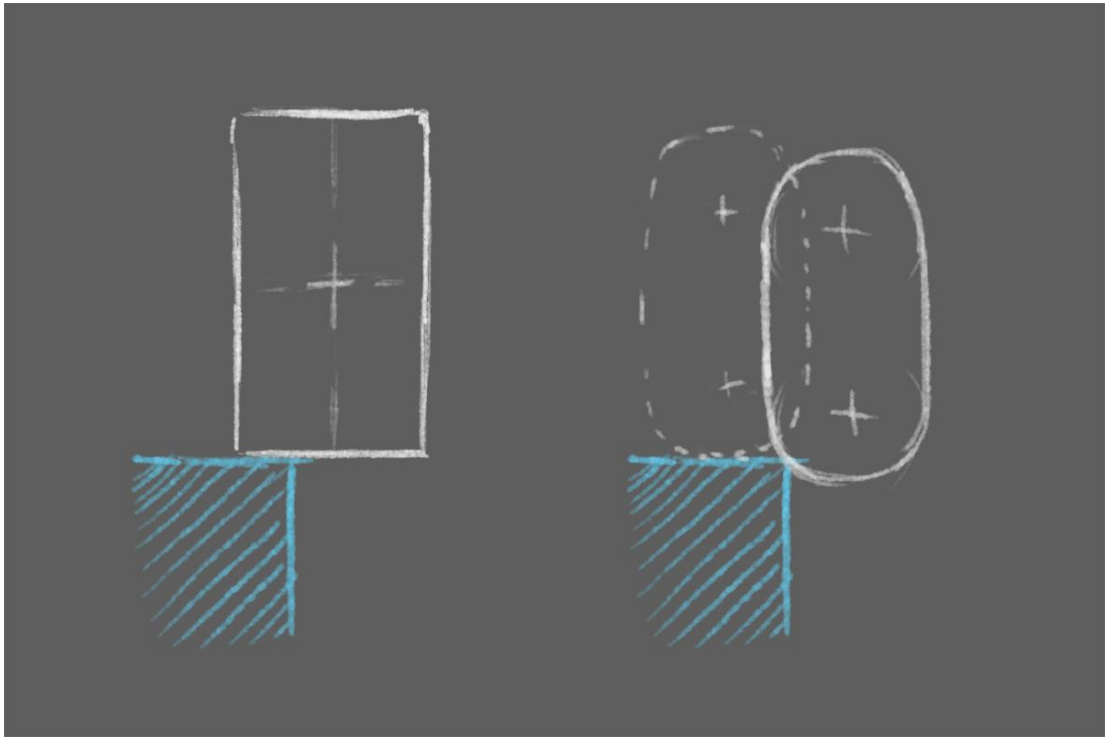
Getting started

Download the starter project provided above and open it in the Unity editor. We will build off this project to implement our character controller. Open the `Main` scene (located at the project root), and open the `CharacterController2D` script in your preferred code editor.

Inside `CharacterController2D` there are already a number of private fields exposed to the inspector. As well, there is a `Vector2` to keep track of our controller's velocity, and a field that references a `BoxCollider`. Most 2D platformer games either use a box or a capsule shape; the shape of our controller will be a box.

Why choose a box over a capsule shape?

There is no one correct choice, but boxes can offer the player a more predictable collision response to their actions. By having a flat bottom, players can more easily guess how far they can edge out over cliffs before falling.



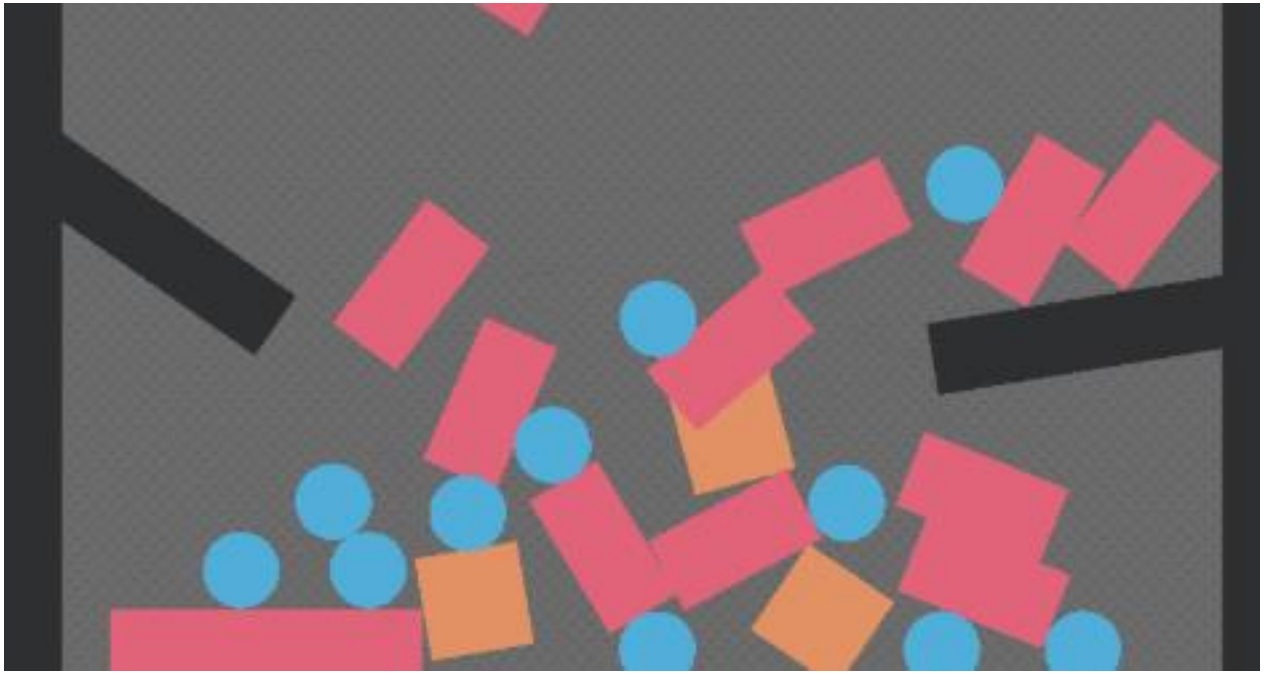
1. Movement

Objects in Unity are moved primarily in two different ways: either by modifying the position of a transform directly or by applying a force to an object with a rigidbody and allowing Unity's physics system to decide how that object should move.

Why not use rigidbodies and forces?

Unity comes packaged with two physics engines: PhysX for 3D, and Box2D for 2D. Both of these extensively support rigidbodies. So why not use them for character controllers?

Rigidbodies behave very similarly to the real world objects they represent. Box rigidbodies will excel at representing a wooden crate, cylinder rigidbodies as a an oil drum, and so on. Using constraints and joints, they can be used to model more complex objects, like a multi-limb robot.



By instead directly modifying the position of our controller, we are able to very finely tune exactly how it interacts with the world, which is essential to crafting a game that feels and plays fluidly.

Add the following line of code in the Update method.

```
  transform.Translate(velocity * Time.deltaTime);
```

Our velocity isn't being modified yet, so our controller won't move. Let's change that by adding some horizontal velocity when the left or right keys are pressed.

```
  float moveInput = Input.GetAxisRaw("Horizontal");
  velocity.x = Mathf.MoveTowards(velocity.x, speed * moveInput,
walkAcceleration * Time.deltaTime);
```

Mathf.MoveTowards is being used to move our current x velocity value to its target, our controller's speed (in the direction of our sampled input).

$$\mathbf{s}(t) = \mathbf{s}_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}t^2$$

$$\mathbf{v}(t) = \mathbf{v}_0 + \mathbf{a}t$$

Note that when no keys are being pressed, `moveInput` will be zero, causing our controller to slow to a stop. This is fine, but we might want to have the deceleration rate different than our `walkAcceleration`. We can handle this by checking to see if moveInput has a non-zero value.

```
  if (moveInput != 0)
  {
      velocity.x = Mathf.MoveTowards(velocity.x, speed * moveInput,
walkAcceleration * Time.deltaTime);
  }
  else
  {
      velocity.x = Mathf.MoveTowards(velocity.x, 0, groundDeceleration *
Time.deltaTime);
  }
```

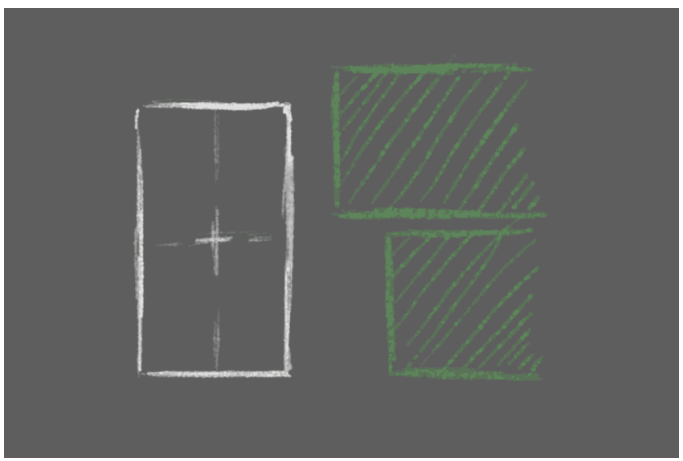
2. Collision

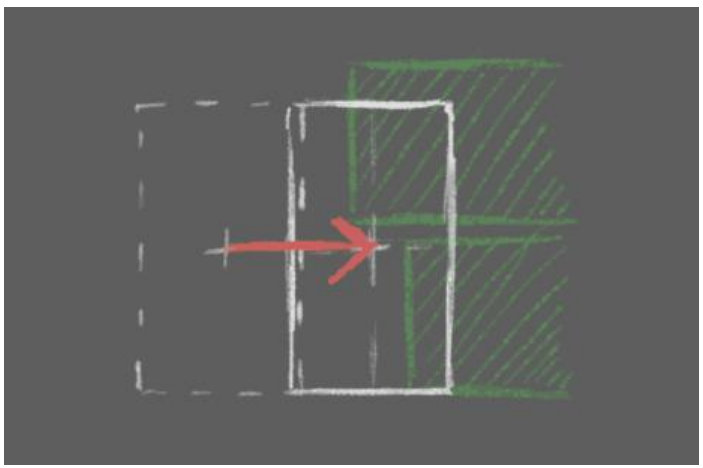
2.1 Detection

We'll start with detection. Our goal is to find all colliders our controller is currently touching. We should do this *after* we have translated our controller to ensure no further movement will occur after we resolve collisions. Unity provides a series of Physics2D overlap functions to help detect intersections. We'll use OverlapBox. Insert the following code at the bottom of `Update`.

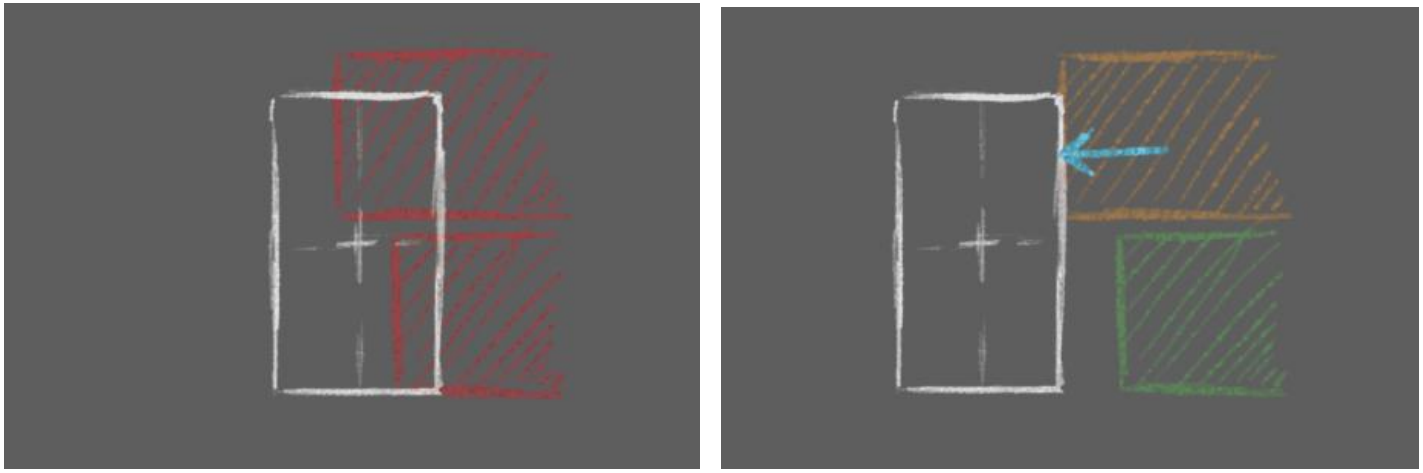
```
  Collider2D[] hits = Physics2D.OverlapBoxAll(transform.position,
boxCollider.size, 0);
```

This will give us an array of all colliders that are intersected by the box we defined, which is the same size as our `BoxCollider` and at the same position. Note that because of this, the array will also contain our own `BoxCollider`.





2.1 Resolution



The main problem is to decide which direction, and how far, we need to translate our controller to depenetrate from each collider. Ideally, we should move it *the minimum distance required to be no longer touching the other collider.* Unity provides a method to find that distance for us, Collider2D.Distance.

```
  foreach (Collider2D hit in hits)
  {
      if (hit == boxCollider)
            continue;

      ColliderDistance2D colliderDistance = hit.Distance(boxCollider);

      if (colliderDistance.isOverlapped)
      {
            transform.Translate(colliderDistance.pointA -
colliderDistance.pointB);
      }
  }
```

As noted above, the array will contain our own `BoxCollider`—we skip it during our foreach loop.

`ColliderDistance2D.isOverlapped`, tells us if the two colliders are touching. Once we have ensured they are, we take the `Vector2` from `pointA` to `pointB`. This is the shortest vector that will push our collider out of the other, resolving the collision.

In **Project Settings → Physics 2D**, make sure that `Auto Sync Transforms` is **checked**. This property automatically syncs physics objects (such as our controller's `Box Collider`) to the physics engine; when it is disabled, objects are only synced at the end of the frame.

3. Jumping

3.1 Air movement

We'll begin by adding in the ability to jump by pressing the "Jump" button (this defaults to Spacebar in Unity). Insert the following at the top of `Update`.

```
  if (Input.GetButtonDown("Jump"))
  {
      velocity.y = Mathf.Sqrt(2 * jumpHeight *
Mathf.Abs(Physics2D.gravity.y));
  }
```

Once we're in the air, we'll need gravity to pull us back to the surface.

```
  velocity.y += Physics2D.gravity.y * Time.deltaTime;
```

Note that in the skeleton project, `Physics2D.gravity` is set to (0, -25). This value can be modified in `Edit > Project Settings > Physics 2D`.

You'll notice two issues at this point: our controller can jump while in mid-air, and gravity is continuously applied even when on the ground. To resolve this, we'll need to know when our controller is *grounded.*

How does the code calculating jump velocity work?

$$\mathbf{s}(t) = \mathbf{s}_0 + \mathbf{v}_0 t + \tfrac{1}{2}\mathbf{a}t^2 = \mathbf{s}_0 + \frac{\mathbf{v}_0 + \mathbf{v}(t)}{2}t$$
$$\mathbf{v}(t) = \mathbf{v}_0 + \mathbf{a}t$$
$$v^2(t) = {v_0}^2 + 2\mathbf{a}\cdot[\mathbf{s}(t) - \mathbf{s}_0]$$

where

- t is the elapsed time,
- $\mathbf{s}_0$ is the initial displacement from the origin,
- $\mathbf{s}(t)$ is the displacement from the origin at time t,
- $\mathbf{v}_0$ is the initial velocity,
- $\mathbf{v}(t)$ is the velocity at time t, and
- $\mathbf{a}$ is the uniform rate of acceleration.

3.2 Detecting ground

Before being able to know when our character is on the ground, we have to first define what "ground" is in this context. We will define ground as any surface at with angle of **less than** 90 degrees with respect to the world up. Our controller will be defined as *grounded* if they have contacted a ground.

We will accomplish this by testing the normal of each surface we collided with to see if it fulfills our criteria as "ground".

```
  // Insert as a new field in the class.
  private bool grounded;

  …

  // Place above the foreach loop to clear the ground state each
frame.
  grounded = false;

  …

  // Place inside the foreach loop, just after the
transform.Translate call.
  if (Vector2.Angle(colliderDistance.normal, Vector2.up) < 90 && velocity.y <
0)
  {
      grounded = true;
  }
```

Ground is now being correctly detected and stored in a field. We can use this data to solve the problems stated at the end of 3.1. Wrap the jumping code in the following if statement.

```
  if (grounded)
  {
      velocity.y = 0;

      // Jumping code we implemented earlier—no changes were
made here.
      if (Input.GetButtonDown("Jump"))
      {
            // Calculate the velocity required to achieve the
target jump height.
            velocity.y = Mathf.Sqrt(2 * jumpHeight *
Mathf.Abs(Physics2D.gravity.y));
      }
  }
```

Jumping is now only permitted when our controller is grounded. As well, we set our y velocity to zero each frame we are grounded.

3.2 Air momentum

Most platforming games tend to restrict a player's control while they are in the air, typically by reducing how quickly they can accelerate. As well, there isn't usually any automatic deceleration applied when there is no movement input from the player. These design choices help add a feeling of weight and commitment to jumping, making it more exciting.

Let's add them to our controller, inserting the code below immediately after gravity is applied, replacing the previous `if` statement.

```
  float acceleration = grounded ? walkAcceleration : airAcceleration;
  float deceleration = grounded ? groundDeceleration : 0;

  // Update the velocity assignment statements to use our
selected
  // acceleration and deceleration values.
  if (moveInput != 0)
  {
      velocity.x = Mathf.MoveTowards(velocity.x, speed * moveInput,
acceleration * Time.deltaTime);
  }
  else
  {
      velocity.x = Mathf.MoveTowards(velocity.x, 0, deceleration *
Time.deltaTime);
  }
```

Conclusion

The controller built in this lesson can be used as a solid foundation for nearly any kind of 2D project. Although the mechanics and interactions will vary from game to game, the core fundementals—velocity, collision detection and resolution, grounding—tend to always be present.