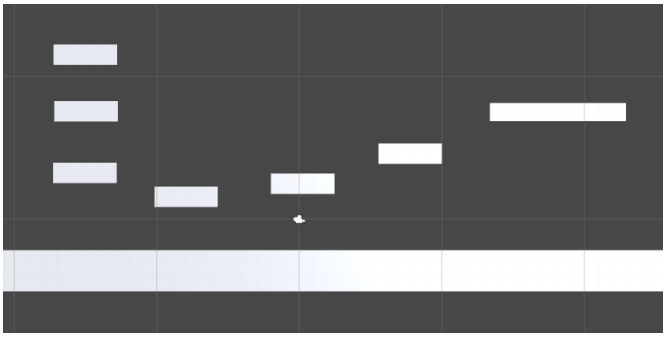
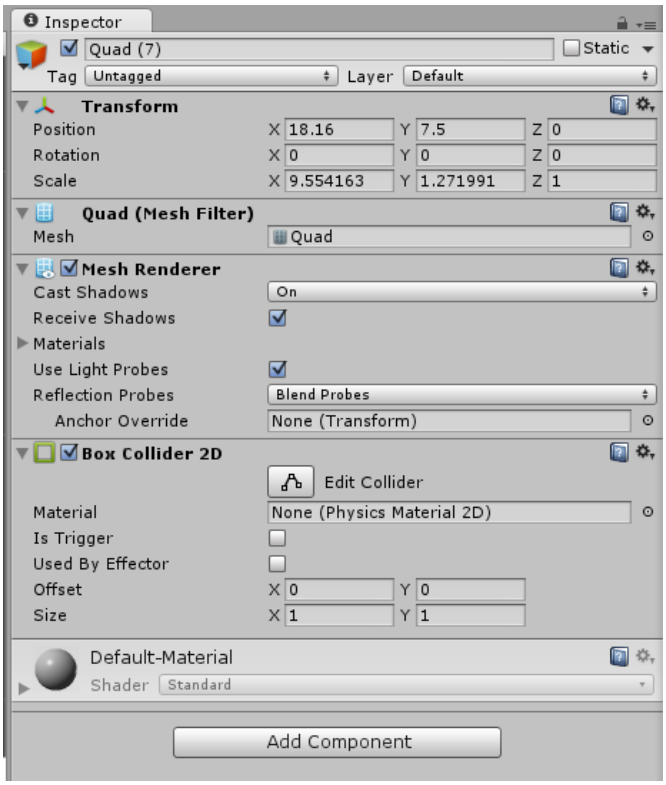
Tutorial – Case Study – 2D Platformer

In this tutorial, we're going to implement the player controller and camera for a simple 2D platformer in Unity. We're going to focus on the player and camera feeling really good. We'll look at multiple ways to set up the player and camera and what the trade-offs between them are.

Setting up the Scene

First we're going to create a new Unity Project for our 2D platformer. Make sure to set it as a 2D project in the new project window. Once the project is open, create a number of Quads in the scene and spread them out to make some test platforms. Make sure you remove the mesh collider and add a BoxCollider2D to them instead.





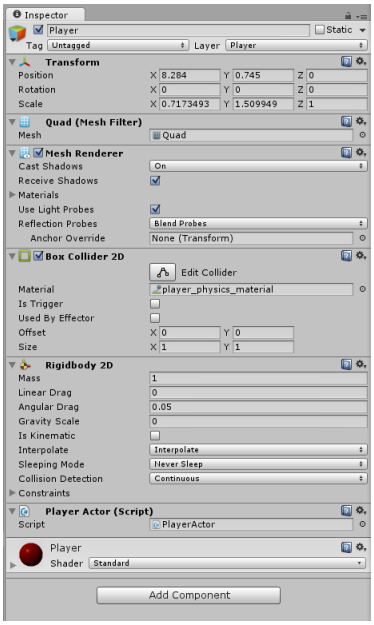
Creating the Player

Create a new script called PlayerActor. This is going to be the main script we use for controlling the movement of the player. Movement in 2D platformers is highly dependent on what kind of game you want to make. The bare minimum in most platformers includes moving left and right, and jumping, but there are many features we could add beyond that, including

- Running
- Dashing
- Double Jumps
- Wall Jumps
- Slides
- Crouching

For the tutorial, we are going to implement both walk and run modes, and a regular jump.
To set up our player, we're going to need to do a few things:

- Create a new Quad in the Scene called player, and add our empty PlayerActor script to it.
- We're going to be implementing several different Cameras later, but as we're building the players movement, we're just going to parent the camera to the player.
- Create a new material for the player, so we can tell the player apart from the rest of the scene.
- Add a RigidBody2D to the player. We won't be using the rigidbody to control the movement of the player, but we need it to correctly get collision to work properly.
- Set the linear drag field to 0, the interpolate mode to interpolate, the sleeping mode to never sleep and the Collision detection mode to Continuous. This gives us the best, most expensive collision detection on the player. Under constraints, tick the freeze rotation box.
- Create a new PhysicsMaterial2D, and set the friction and bounciness to 0. Add the material to the player's collider



Player Movement and Collision

Building the movement for a platformer is a big part of the process of developing one. There are many different options and there is no right answer to how yours should behave. That said there are a number of core ways that player controllers for platformers tend to be implemented.

The simplest way to get the player to move is to directly add to the position. We can see how this kind of movement feels by adding this code to the PlayerActor.

```
0 references
public class PlayerActor : MonoBehaviour {

    public float walk_speed;//speed for when walking
    public float run_speed; //speed for when running

    Rigidbody2D rigid_body; //rigid body for setting velocity

    0 references
    void Start()
    {
        rigid_body = GetComponent<Rigidbody2D>();
    }

    0 references
    void FixedUpdate()
    {
        float curr_speed = walk_speed;
        if (Input.GetKey(KeyCode.LeftShift)) curr_speed = run_speed;

        //set velocity based on inputs
        Vector2 velocity = Vector2.zero;
        if (Input.GetKey(KeyCode.A)) {
            velocity += Vector2.left * curr_speed;
        }
        if (Input.GetKey(KeyCode.D)) {
            velocity += Vector2.right * curr_speed;
        }
        //use the rigid body for velocity so we get good collision
        rigid_body.velocity = velocity;
    }
}
```

For some platformers, this very simple method will be all you will need, but most of the time, we want to use something a bit more elaborate.

There are many other options games take to control the movement of the player:

- Movement is driven by animation curves created by artists and designers
- A target speed is decided based on input, and the player interpolates towards that speed
- Key presses accelerate the player controller. Every frame a drag is also applied to the player proportional to its velocity. When a key is held, the player will accelerate until the drag balances out.
- The walk speed could happen instantly, but transitioning from walk to run and back happens over time

As an example, in Super Meat Boy, when you are on the ground, all transitions from stationary, to walk, to run are very close to instant. In the air, however, the acceleration and drag technique is used.

The movement schemes above should only operate on the X component of the velocity. The Y component is for gravity and jumping.

To implement movement that uses acceleration/drag, we should create a member variable for velocity, and instead of walk and run speed, we need a walk and run acceleration and drag. When the left and right keys are held, forces are accumulated.

```
public float walk_accel;
public float walk_drag;
public float run_accel;
public float run_drag;

Vector2 velocity;
0 references
void Start() {
    rigid_body = GetComponent<Rigidbody2D>();
}
0 references
void Update() {
    //choose correct walk/run variables
    float curr_accel = walk_accel;
    float curr_drag = walk_drag;
    if (Input.GetKey(KeyCode.LeftShift)) {
        curr_accel = run_accel;
        curr_drag = run_drag;
    }
    //accumulate forces
    float force = 0;
    if (Input.GetKey(KeyCode.A)) {
        force -= curr_accel;
    }
    if (Input.GetKey(KeyCode.D)) {
        force += curr_accel;
    }

    //apply drag to force and force to velocity
    force -= velocity.x * curr_drag;
    velocity.x += force * Time.deltaTime;
    //use the rigid body for velocity so we get good collision
    rigid_body.velocity = velocity;
```

One problem with doing this in Unity is that now, when we hit a wall, we want our internal velocity to update in response to the collision. Add this code to the bottom of your PlayerActor class

```
1 reference
void OnCollisionEnter2D(Collision2D hit)
{
    for ( int i = 0 ; i < hit.contacts.Length ; ++i ) {
        Vector2 n = hit.contacts[i].normal;
        float d = Vector2.Dot(velocity, n);
        if (d < 0) {
            velocity -= Vector2.Dot(velocity, n) * n;
        }
    }
}
```

This loops through all the contact points and removes the normal component from the velocity, stopping it in that direction.

Jumping and Grounded

Like the rest of our player controller, there are a lot of choices we need to make when creating a jump. Most of these choices don't have a right or a wrong answer – it depends on the game you're making. At the very least, you need to be able to tell if the player is touching the ground, so you can tell if the player should be able to jump.

Add a new bool to the PlayerActor script called grounded. This variable tells us if the player is touching the ground right now.

```
bool grounded = false;
```

We're going to add to our OnCollisionEnter function. In it, if the normal of the surface is pointing up, and we are going down, we set grounded to true.

```
1 reference
void OnCollisionEnter2D(Collision2D hit)
{
    for ( int i = 0 ; i < hit.contacts.Length ; ++i ) {
        Vector2 n = hit.contacts[i].normal;

        float d = Vector2.Dot(velocity, n);
        if (d < 0) {
            velocity -= Vector2.Dot(velocity, n) * n;
        }

        if ( Vector3.Dot(n, Vector2.up) > 0.5f &&
            Vector3.Dot(velocity, Vector2.down) >= 0f)
        {
            grounded = true;
        }
    }
}
```

Many implementations of a grounded variable in platformers give a little leeway to the player. They can leave the ground for a small fraction of a second before grounded is set to false. This also smooths out small flaws in the level collision. We can create a variable called ground_timer. We'll use ground_timer to count up how long it has been since we were on the ground. Every time we touch the ground, we reset the timer to 0.

```
if ( Vector3.Dot(n, Vector2.up) > 0.5f &&
    Vector3.Dot(velocity, Vector2.down) >= 0f)
{
    grounded_timer = 0f;
    grounded = true;
}
```

Then in update we add delta time to it. If it's been more than 0.2 seconds without the collision system resetting the timer, we set grounded to false.

```
0 references
void Update()
{
    grounded_timer += Time.deltaTime;
    if (grounded_timer > 0.2f) grounded = false;
```

Because the timer reset now needs to constantly be happening when we're touching the ground, we need it to happen in both OnCollisionEnter and OnCollisionStay

```
2 references
void ManageGroundTimer(Collision2D hit)
{
    for (int i = 0; i < hit.contacts.Length; ++i)
    {
        Vector2 n = hit.contacts[i].normal;
        if (Vector3.Dot(n, Vector2.up) > 0.5f &&
            Vector3.Dot(velocity, Vector2.down) >= 0f)
        {
            grounded_timer = 0f;
            grounded = true;
        }
    }
}
0 references
void OnCollisionStay2D(Collision2D hit)
{
    ManageGroundTimer(hit);
}
0 references
void OnCollisionEnter2D(Collision2D hit)
{
    ManageGroundTimer(hit);
}
```

Now we need to decide how the jump will work. Again, with a jump in platformers there are many options for what we might want the jump to feel like.

- Canned jump height
- Air control
- Holding jump makes a longer jump

The simplest way to implement a jump is just to set the y velocity when the player hits space

```
if (Input.GetKeyDown(KeyCode.Space) && grounded) {
    velocity.y = jump_power;
    grounded = false;
}
```

Basic Camera Movement

So far, we've been using the simplest possible camera, just parenting it directly to the player. Like the most basic version of movement shown earlier, this simple approach is used by some games, however most need something more complex. One very simple change to our existing scheme will make it feel much better. Create a new script called CameraActor. Un-parent the camera from the player and add the following code to the CameraActor.

```
0 references
public class CameraActor : MonoBehaviour {

    public Transform target;
    public float speed;

    Vector3 vector_from_target;

    0 references
    void Start () {
        vector_from_target = transform.position - target.position;
    }

    0 references
    void Update () {
        Vector3 desired_pos = target.position + vector_from_target;

        transform.position =
            Vector3.Lerp(transform.position, desired_pos, Time.deltaTime * speed);
    }
}
```

On start-up, this code simply gets the vector from the player to the camera. It then tries to keep the camera in the same position relative to the player. To keep the camera moving smoothly, instead of setting the position directly to the desired point (which would look the same as parenting it), we interpolate towards the goal. Make sure to drag the player into the target in the inspector.

There are many other options for implementing a camera

- Camera Box
    - There is a box in the screen where the player is free to move. As long as they stay in the box, the camera doesn't change position. If they move outside of the box, the camera will pan to keep the player in. You can combine this with the smoothed camera technique from earlier to make the panning smoother.
- Screen Anchors
    - Screen anchors move the camera to anchor the player to one side of the screen or another. If the player is moving to the right, the camera shifts to the right to show what's in front of them and vice versa.

Exercises

1. Modify your player so that it can double jump
2. Modify your player so that it can wall jump
3. Most cameras in 2D platformers combine many of the techniques we've talked about here. Often, different cameras are used in different areas of the game world. Create a new CameraAreaActor script that sets the behaviour of the camera when the player enters that area
4. Add a dash to the player