Breeky Studio

RBCC: Documentation

A physics-based character controller

Breeky
01/04/2022

Contents

Introduction

Thanks for buying RBCC, a physics-based character controller!

Tired of implementing physics manually? This asset is for you. RBCC uses Rigidbodies to move your character, allowing your character to react to any force and get a realistic result! With this asset, your character can walk, run, jump, bounce on walls, crouch, slide on walls and more. You also have at your disposal operational scripts to add dynamic platforms to your game. Finally, the gravity applied to your character can be easily modified to allow you to defy the laws of physics in realtime. With this documentation, you'll be able to add even more custom movements quickly and easily.

This is an asset targeting mainly programmers, but I hope there's enough tuning options to make it accessible for designers and other non-programmers as well. This controller is inspired from the one described by Toyful Games. If you haven't checked their blog, there's a lot of good stuff there.

Getting started

This asset uses Unity new input system and Cinemachine.

To try the controller, navigate to the testing scenes in Scenes/Playground folder. You'll find different levels to showcase the different features of this asset with all the possible movements of the controller.

To use this controller in your scene, use the already made prefab and adapt it to your needs. It is strongly advised to add the character controller script on an empty GameObject, and your graphics / model as a child of this GameObject. This is a nice way of structuring your character GameObject to keep things clean and easy to debug. This architecture is already implemented on the pre-made prefabs so don't worry!

Setting up the controller

Adding a character controller to your scene:

1. Drag and drop the Player prefab from Prefabs/Player into your scene.
2. On the PlayerStateController script of your Player, in the "Ground Layer" field, select all layers on which the character can move (default to "Ground", you can assign this layer to your objects).

Add your model for the character:

1. Replace the children of the "GFX" GameObject of the Player with your model.

Setting up cameras

This asset includes basic first and third person cameras, as well as a simple follow camera.

By default, the character controller will use the main Camera, but you can assign your own on the PlayerStateController script by a simple drag and drop on the "Cam" field.

The camera is used to define in which direction the character will go. You can modify this behavior in the script (See Handling horizontal movement). Remember to **only have one main camera in your scene for the automatic assignment to work**.

Adding a custom camera to control the character:

1. Drag and drop the camera object on the "Cam" field of the PlayerStateController script.

Setting up the pre-made 3rd Person Camera

1. Drag into your scene the **Third Person Camera prefab** from Prefabs/Cameras
2. Drag and drop the Player's **Camera Anchor** GameObject (child of the player prefab) to the **Follow Transform** field of the ThirdPersonCamera script.
3. Drag into your scene the **Main Camera (Fixed Update)** prefab from Prefabs/Cameras.
4. On the Player, open the *Events* dropdown of the **PlayerInput** component.
5. Open the *Player* dropdown.
6. Find the **Look (CallbackContext)** box and click on the "+" button.
7. Drag and drop the third person camera's GameObject to the newly created target field.
8. For the function, select ThirdPersonCamera->OnLook.

Setting up the pre-made 1st Person Camera

1. Drag into your scene the **First-Person Camera** prefab from Prefabs/Cameras
2. Drag and drop the Player's **Camera Anchor** GameObject (child of the player prefab) to the **Follow** field of the CinemachineVirtualCamera script.
3. Drag into your scene the **Main Camera (Smart Update)** prefab from Prefabs/Cameras.
4. That's it. If you want to hide some objects while in first person, set up culling on the (main) Camera. To do this, assign a layer to the objects you want to hide. On the Camera component, on "Culling Mask", deselect the layers you want to hide (note that you won't be able to see the object's shadows anymore).

Setting up the pre-made Follow Camera

1. Drag into your scene the **Follow Camera** prefab from Prefabs/Cameras
2. Drag and drop the Player's **Camera Anchor** GameObject (child of the player prefab) to the **Follow** field of the CinemachineVirtualCamera script.
3. Drag into your scene the **Main Camera (Fixed Update)** prefab from Prefabs/Cameras.

Setting up Platforms

This character controller supports moving platforms that allow your character to move with the environment. Some prefabs already exist to showcase basic results in the playgrounds scene, but you can easily create your own moving platform scripts. The controller supports both Rigidbody-based platforms and platforms with their transforms manually controlled in scripts. You can check the pre-made platforms prefabs in Prefabs/Platforms to see some examples of implementation.

Basically, if your moving platform has a **Rigidbody**, the character will move while on it automatically. However, if you move your platform by modifying its transform, add a **MovingPlatformComponent** on your platform to register the movement for the character controller.

Setting up a kinematic platform (Rigidbody)

1. Make sure your platform has a Rigidbody and is set to kinematic.
2. Add a **Kinematic Translating Platform** or a **Kinematic Rotating Platform** on it.

Setting up basic platforms (Platforms transform directly modified in scripts)

1. Make sure your platform doesn't have a Rigidbody.
2. Add a **MovingPlatformComponent** on it.

Generally speaking, the quickest way to understand this asset is to take a look at the demo scenes to see how the different objects are set up. Don't hesitate to hover on the different variables of the scripts, many tooltips are waiting for you. In the same way, the code is documented according to the standards, allowing your IDE to describe each function easily.

Character controller

This character controller uses Unity Rigidbody to move your character making it dynamic (i.e., being able to react to forces). This controller is inspired by the one described by Toyful Games. This controller also uses a hierarchical state machine structure to make the process of adding more movement behaviors very quickly and easily. This also makes it less prone to errors by separating the different behaviors into states.

Floating capsule

The character can be represented as a floating capsule. When you move your character, it accelerates and decelerates in the desired direction while maintaining a certain height above the ground and smoothly rotating to the direction of movement.

There are several advantages of keeping the character above the ground:

- The character is not disturbed by the artifacts of the ground. It will move smoothly on uneven ground for instance.
- There is no friction. While not touching the ground, you have full control of how you want the friction to be applied. The player inputs feel also more responsive.
- The character sticks on slopes. By detecting the ground with a downward raycast, you control how far you want your character to stick to the ground. This is particularly useful on slopes to not have your character accelerating too quickly and taking off.
- You control the force applied by the character to the ground. While maintaining the character above the ground at a specific height, you have full control on which force to apply to the ground without having to handle the weight of your character.

To have a better understanding of how it works, check the video from Toyful Games here.

Ground detection

To detect the ground, a ray is cast along a certain direction.

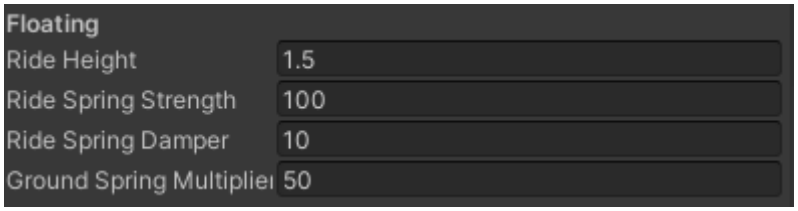
- The **ground layer** is used to define on which objects in the world the character will float.
- The **ray length** is the range of ground detection. Once detected, the floating force is applied.
- The **ground ray direction** is both used to set the direction of the ray and to define in which direction the gravity is applied on the character. For instance, in the custom gravity scene demo, the gravity cubes modify this direction to modify the gravity.

Ground detection parameters

Floating force

The floating force can be modeled as spring-damper system.

- **Ride height** defines the height between the character root position and the ground (point of impact).
- **Ride spring strength** sets how hard the force attract the player to the ground.
- **Ride spring damper** sets how quick the bounces stop (increase this value to reduce bouncing effects).
- **Ground spring multiplier** is a multiplier to set the force value applied to the ground (spring force * spring multiplier).

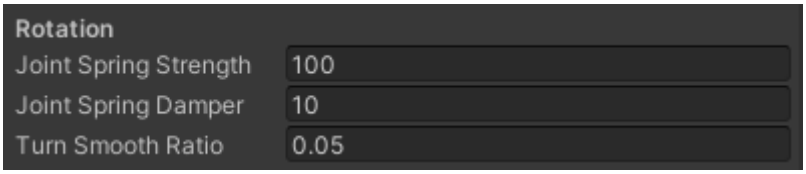


Floating parameters (Ground state)

Rotation

The rotation system of the character uses the same type of spring force to maintain the character vertical (along the ground ray direction). However, there is also a ratio that will speed up the rotation to obtain a smoother rotation.

- **Joint Spring Strength** sets the torque applied to the character to make it vertical.
- **Joint spring damper** value reduces the bouncing effects as it increases.
- **Turn Smooth Ratio** is a linear interpolation ratio used to rotate the character transform directly.



Rotation parameters (Ground State)

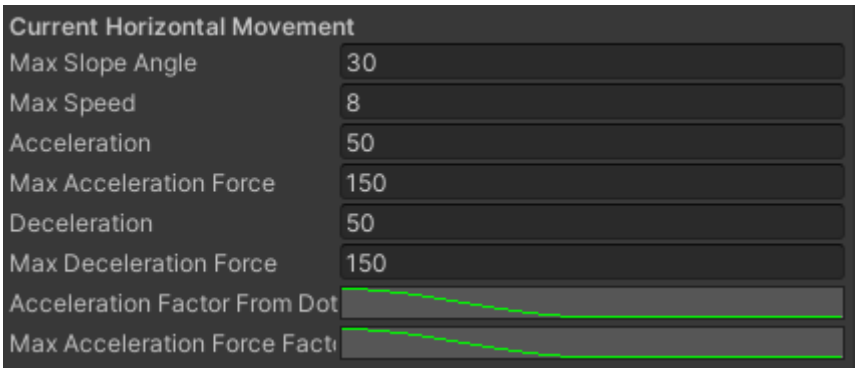
Walk and Run

To move around, a force is applied to the character each frame. The following parameters are common to walking and running, with only their value changed between the two states.

- **Max Speed**: max speed the character can reach by itself.
- Acceleration: acceleration value
- **Max Acceleration Force**: maximum force magnitude to prevent the character to be overpowered and be able to push too heavy objects.
- Deceleration: deceleration value
- **Max Deceleration Force**: maximum deceleration force to prevent the character to apply an over-sized force.
- **Acceleration Factor from dot**: acceleration multiplier based on the velocity and movement. If the movement direction is opposed to the desired direction (x=-1), the value is 2. Otherwise if

the desired direction doesn't go against the movement direction (x>0) the value is 1. This is used to reduce the amount of time needed to go from a direction to an opposite one.

- **Max Acceleration Force Factor**: same as above but for the Maximum Acceleration force.

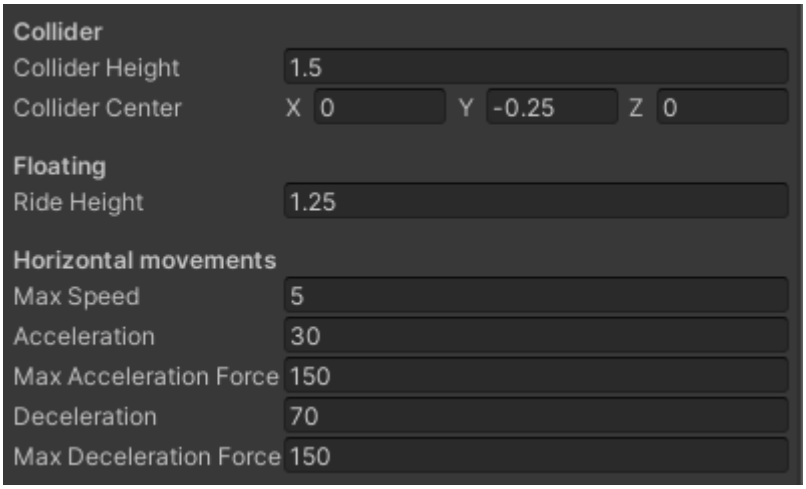


Horizontal Movement Parameters (root PlayerStateController)

Crouch

Crouching is basically walking but with other speed values and a modified collider to adapt to the character.

- **Collider height while crouching**
- **Collider center while crouching**



Crouch parameters (Crouch State)

Slopes

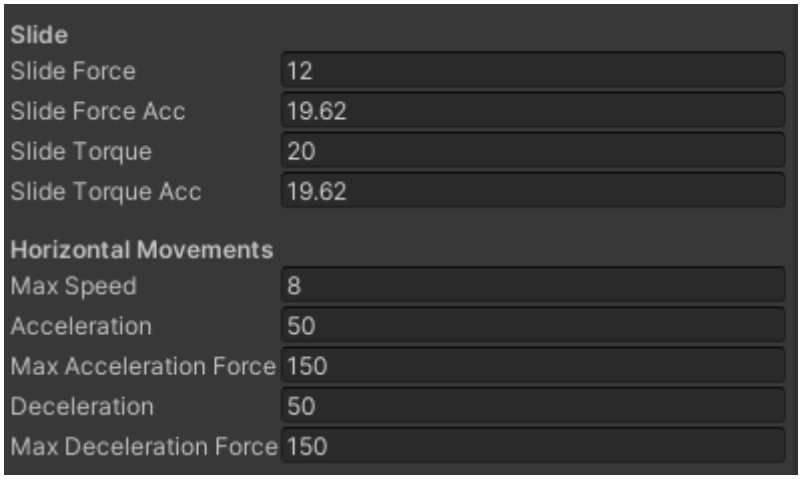
The character is able to step on slopes natively, with a sliding force applied to it when the slope angle is above a certain value. To detect the slope angle, the resulted hit from the ground raycast is used to detect the ground normal and calculate the angle from there.

- **MaxSlopeAngle** defines the slope angle above which the character is considered as sliding, which can trigger a change of state.

Slope parameter (under Ground State)

The following parameters are used in the crouch state which is triggered by default when over a slope with a slope angle above the maximum value defined above.

- **Slide Force**: force applied to the character in the direction of the slope
- **Slide Force Acc**: acceleration of this force (smoother effect)
- **Slide Torque**: torque applied to the character to face the bottom of the slope
- **Slide Torque Acc**: acceleration of the previous torque

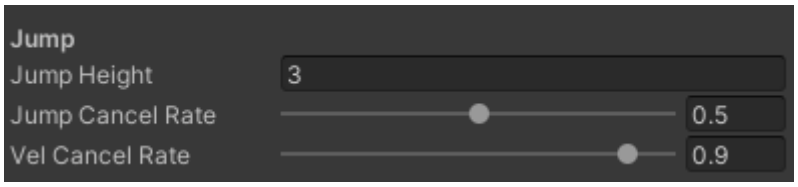


Slide Parameters (under Slide State)

Jump

The character can perform a jump and multiple jumps while in air. To obtain a nice jump, additional parameters other than the jump height have been added.

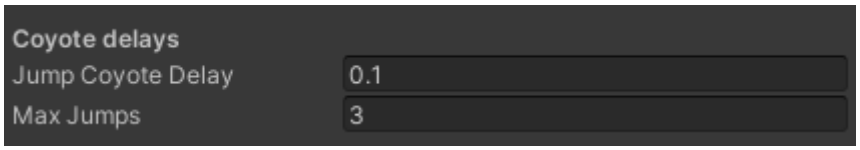
- **Jump height**: how high the character jumps from its current position
- **Jump Cancel Rate**: what % of the vertical velocity to reduce while releasing the jump key before reaching the peak of the jump. This is the Mario jump effect: if you press the jump key longer you jump higher.
- **Velocity Cancel Rate** (for multiple jumps): while in air and pressing jump, what % of the vertical velocity to reduce before adding the jump force. If set to 1, the character will jump as it has no previous vertical velocity. If set to 0, a jump force is added on the character upon pressing jump, but it may have no effect if it is falling very fast.



Jump parameter (under jump state)

- **Jump Coyote Delay**: delay given to the player to be able to jump after leaving the ground. This makes the jump more forgivable and is very appreciated from players.

- **Max Jumps**: how many jumps the character can perform. If set to 1, the character cannot jump while in air. If set to more than 1, the character can perform multiple jumps in air.



Jump parameters (on the root PlayerStatecontroller)

Wall jump

The walls are detected with a ray casted in front of the player. The number of jumps is shared with the classic jumps, meaning that if you set the Max Jumps variable to 2, you will be able to jump and perform only one wall jump. This behavior can be modified in scripts.

- **Wall layer**: on which layers the ray is casted to detect walls.
- **Wall Ray Length**: detection range of walls.
- **Wall Ray Local Dir**: direction of the ray in local space.

Wall detection parameters (on root PlayerStateController)

- **Vertical angle**: angle between the normal of the wall and the direction to jump to
- **Jump Height**: height of the jump vertically

Wall jump parameters (under Wall Jump State)

Input bindings

TODO

Hierarchical State Machine

The architecture of the asset code follows the principles of a hierarchical state machines. If you don't know what it is, I recommend the video from iHeartGameDev available here.

Basically, a PlayerStateController script holds all the basic functions and variables needed to move the character. A PlayerStateFactory script assign the different movement state to the state controller. A PlayerState script defines a type of movement (walking, running, jumping, etc.). Finally, A PayerStateData stores all the information relative to a state (such as walk speed, jump height, etc.). All these scripts are located in Scripts/RBCC/Player and a lot of them are documented.

Within each State (inheriting from PlayerBaseState), you have a reference to the PlayerStateController with the 'Ctx' attribute.

Root and Sub State

There are two kinds of State: Root State and Sub State. They differ as one can be seen as a parent and the other as a child. For instance, the current root states are Ground State and Air State while the currently implemented sub states are walking, running, jumping, falling, etc.

This architecture is useful to handle the state transitions and movement separately. While in ground state, the character can float above the ground (the functions are implemented in its script). However, in the air, the character decelerates much less. You could add another Root state such as a Water State for when the character is in the water. While the RootStates handle commonly used functionality across substates, the substates define custom movements. For example, the walk state has specific values for MaxSpeed and Acceleration, and the CrouchState has some collider size properties to change it when entering the state.

To define a State as a RootState, override the PlayerBaseState attribute in the constructor:

```
IsRootState = true;
InitializeSubState();
```

Make sure to also Initialize the substate to handle state transitions properly. If you want to create a substate, leave the IsRootState attribute as default (false).

State Transitions

Each state defines its own transitions to other states. When updating the state (automatically done in the PlayerStateController), make sure to call CheckSwitchStates() to transition to another state depending on the context.

For example, in the PlayerWalkState:

```
public override void CheckSwitchStates()
{
    if (!Ctx.IsSliding)
    {
        // To Idle
        if (!Ctx.HorizontalInputsDetected)
        {
            SwitchState(Factory.Idle());
            return;
        }

        // To Run
        if (Ctx.HorizontalInputsDetected && Ctx.RunKeyPerformed)
        {
            SwitchState(Factory.Run());
            return;
        }

        // To crouch
        if (Ctx.CrouchKeyPerformed)
        {
            SwitchState(Factory.Crouch());
        }
    }

    // To Slide
    if (Ctx.IsSliding)
    {
        SwitchState(Factory.Slide());
        return;
    }
}
```

We check if the character needs to change state in the CheckSwitchStates() function.

- If the character is sliding, the character goes into the SlideState.
- If not, we check for all the possible states we want depending on the key pressed.

You can think of these transitions as the one in the Animator view. Each transition can be represented by an arrow that points to another state. You define which parameter to check to change state.

When you add a new state, go into the different states you want to be able to switch from, and check to apply the transition to your new state.

How to add a new type of movement

In order to simplify the explanations, we take as an example the addition of a Dash state that allows the player to move quickly over a short distance.

1. Create a PlayerDashState and a PlayerDashData script.
2. Make PlayerDashState inheriting from PlayerBaseState and implement all the necessary methods (you can look to the other state to see how it's done):

```
using RBCC.Player;
using RBCC.Player.PlayerStates;


public class PlayerDashState : PlayerBaseState
{
    private readonly PlayerDashData data;

    public PlayerDashState(PlayerStateController ctx, PlayerStateFactory
factory, PlayerDashData dashData) : base(ctx, factory)
    {
        this.data = dashData;
    }

    public override PlayerState State => PlayerState.PlayerDashState;

    public override void EnterState()
    {}

    public override void UpdateState()
    {}

    public override void FixedUpdateState()
    {}

    public override void ExitState()
    {}

    public override void CheckSwitchStates()
    {}

    public override void InitializeSubState()
    {}
}
```

3. Make PlayerDashData Serializable and remove any inheriting member.

```
using System;


[Serializable]
public class PlayerDashData
{

}
```

4. Add the value "PlayerDashState" to the enum PlayerState located in the PlayerStateFactory script. Setup the State property of the PlayerDashState (using RBCC.Player.PlayerStates if needed):

```
public override PlayerState State => PlayerState.PlayerDashState;
```

5. In the PlayerStateFactory class, under the attribute fallData, add the following line:

```
[SerializeField] private PlayerDashData dashData;
```

6. Under enableFallState add:

```
[SerializeField] private bool enableDashState = true;
```

7. In PlayerDashState, add a private variable to store the dash force applied to the character:

```
public float dashForce = 2000f;
```

8. In the constructor assign it to the value passed in parameter:

```
private readonly PlayerDashData data;

public PlayerDashState(PlayerStateController ctx, PlayerStateFactory
factory, PlayerDashData dashData) : base(ctx, factory)
{
    this.data = dashData;
}
```

9. Back in the PlayerStateFactory create a new method:

```
// Add more here if you like

public PlayerBaseState Dash()
{
    if (enableDashState)
        return new
            PlayerDashState(_context, this, dashData);
    return null;
}
```

10. In PlayerDashState, add a force to the character when entering the state to throw the character in the player looking direction:

```
public override void EnterState()
{
    Ctx.Rb.AddForce(data.dashForce * Ctx.LookingDirection,
ForceMode.Impulse);
}
```

11. In UpdateStates, check if the character needs to change states:

```
public override void UpdateState()
{
    CheckSwitchStates();
}
```

12. As we want to exit the state just after applying the force, we go back to Idle right away:

```
public override void CheckSwitchStates()
{
    SwitchState(Factory.Idle());
}
```

To enter the Dash State, we want to be walking and pressing the F key.

13. In the PlayerStateController, add an event to support the new binding under OnCrouchPressed:

```
public event InputFired OnDashPressed;
```

14. Below the OnJump() function of the PlayerStateController add the following:

```
public void OnDash(InputAction.CallbackContext context)
{
    OnDashPressed?.Invoke(context);
}
```

15. To add the transition from the Walk State to the Dash State go into the PlayerWalkState and add the following method:

```
private void OnDash(InputAction.CallbackContext context)
{
    if (context.performed)
    {
        SwitchState(Factory.Dash());
    }
}
```

16. To listen for inputs from the PlayerStateController, register this function when entering the state (and unregister when exiting):

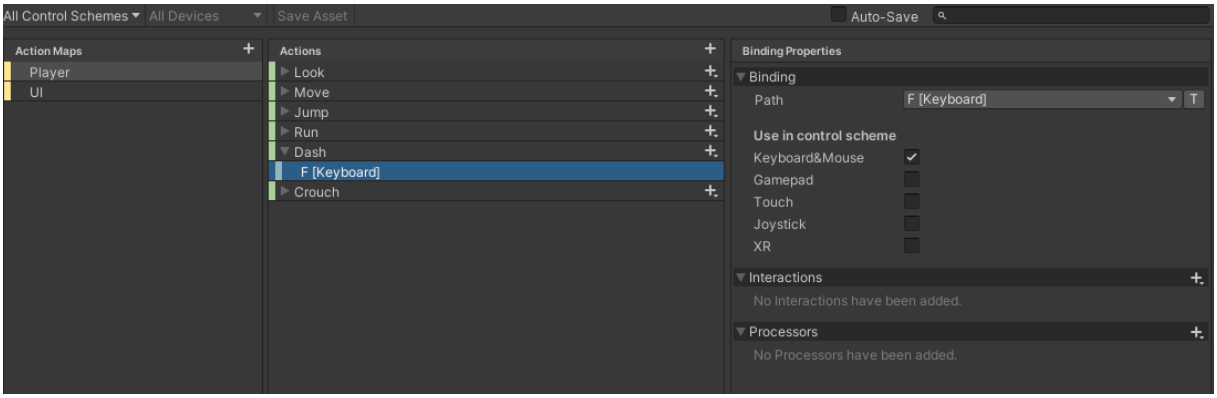
```
public override void EnterState()
{
    (…)

    Ctx.OnDashPressed += OnDash;
}

(…)

public override void ExitState()
{
    Ctx.OnDashPressed -= OnDash;
}
```

17. Back in Unity add the dash binding to the PlayerInput map located in the Inputs/ folder:



18. On your PlayerInput component on the character, open the Events->Player dropdown and click on the “+” button under “Dash”. Assign the character as the target and select the function PlayerStateController->OnDash:

19. You should be able to try the dash in game now!

Here is the complete code:

- PlayerDashState

OpenDocument Text

- PlayerDashData

OpenDocument Text

- PlayerStateFactory

OpenDocument Text

- PlayerStateController

OpenDocument Text

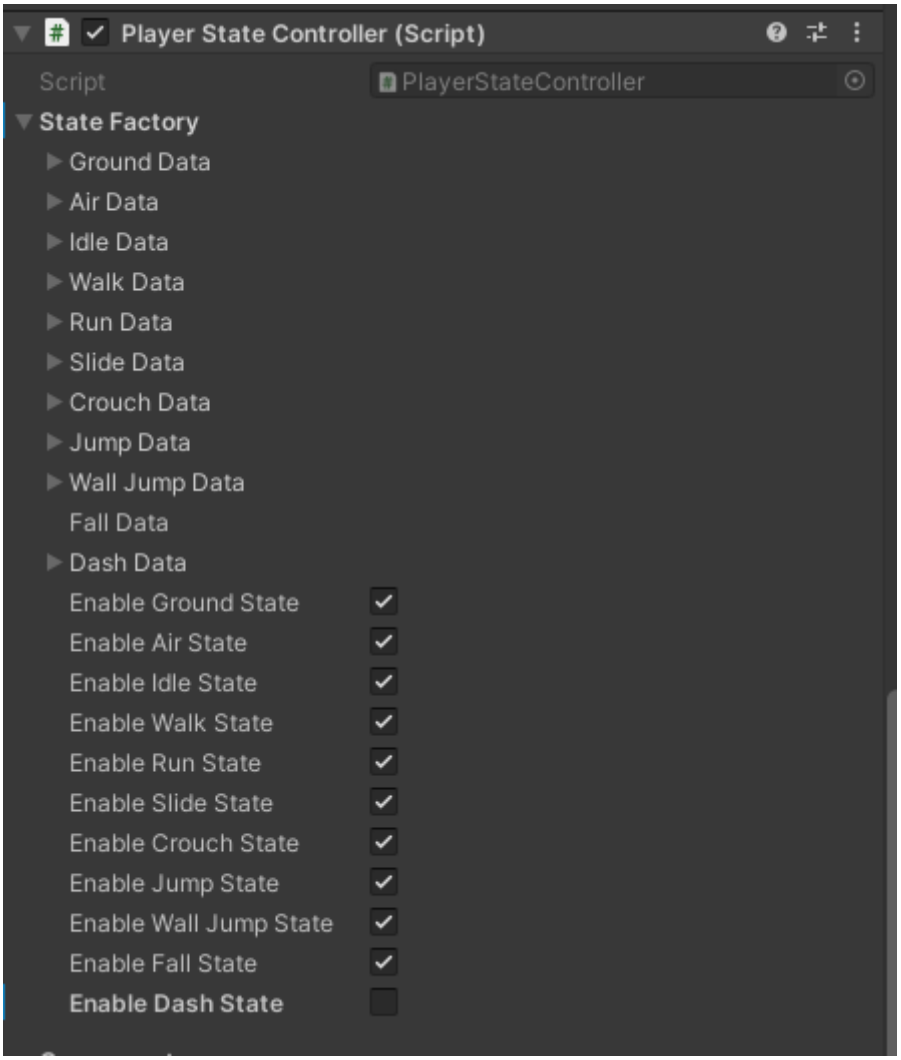
- PlayerWalkState

OpenDocument Text

This dash is very basic but you have all the tools to go on from now. You could for example not exit the state right away and apply a strong deceleration to not go that far. After a small delay, check to exit the state. You can also customize the transitions between the states.

How to disable a type of movement

1. Uncheck (or set to False) the relative Enable State Value on the PlayerStateController.



How to create an animation controller

For inspiration, you can check the demo CapsuleAnimationController and MixamoAnimationController in Scripts/RBCC/Animations.

In the PlayerStateController, there are two useful events triggered when changing states:

- OnEnterState
- OnExitState

One way to handle animations is just to register functions to these events to animate the character accordingly. You can also make use of the PlayerState enum located in the PlayerStateFactory to identify what is the new state.

```
private void OnEnable()
{
    _isRunningKey = Animator.StringToHash(name: "isRunning");
    _isSlidingKey = Animator.StringToHash(name: "isSliding");
    _isJumpingKey = Animator.StringToHash(name: "isJumping");

    player.OnEnterState += UpdateEnterAnimationStates;
    player.OnExitState += UpdateExitAnimationStates;
}

Event function  Samuel Chaput
private void OnDisable()
{
    player.OnEnterState -= UpdateEnterAnimationStates;
    player.OnExitState -= UpdateExitAnimationStates;
}

Frequently called  2 usages  Samuel Chaput
private void UpdateEnterAnimationStates(PlayerState newState)
{
    switch (newState)
    {
        case PlayerState.PlayerWalkState:
        case PlayerState.PlayerRunState:
            animatorController.SetBool(_isRunningKey, value: true);
            break;
```

Programming notes

This asset contains multiple scripts. Some are needed for the character controller while others are just for demonstration. In general, if you have doubts about how to implement a new function, refer to the existing code. Many comments are there to guide you. Here is a quick overview of the scripts:

- RBCC/Animations holds example scripts to animate the capsule and the mixamo character.
- RBCC/Cameras holds basic first- and third-person camera scripts.
- RBCC/Environment holds simple scripts to show how platforms can be handled.
- RBCC/Examples holds simple scripts to make the playgrounds scene functional (enable respawn, changing from one scene to another, etc.)
- RBCC/Player holds all the different scripts needed for the character controller to work properly.
- RBCC/Utils holds static utils function to simplify the code elsewhere.

Handling horizontal movement

The basic movements of the character are described in this video. Here is how it is handled each frame through the fixed update function:

1. Detect the ground, wall in front of the character, and slope below the character with ray casts. Store the hit value.
2. Read the horizontal inputs (WASD keys). Adjust the direction of the input with the camera (forward becomes camera forward). Set up 3 different directions:
    i. **_inputDirection**: raw input direction
    ii. **_movingDirection**: modified later. Direction in which the moving force will be applied.
    iii. **_headingDirection**: _inputDirection adapted to the camera. Direction in which the character is trying to move.
    iv. (property) **LookDirection**: horizontal forward direction of the camera.
- Apply gravity by adding a force proportional to _currentGravity.
- Apply torque to maintain the character vertical.
- Apply movement multipliers if any (speed or acceleration multipliers for example).
- Apply force for horizontal movement.
    i. Calculate the velocity the character needs to reach (relative to maxSpeed and its moving direction).
    ii. Calculate the target velocity for this frame (what value to add to the current velocity considering its acceleration value to go towards the goal velocity).
    iii. Calculate the needed acceleration force to apply to the character to make it reach target velocity.
    iv. Clamp this acceleration force to limit the power of the character to interact with the environment.
    v. Apply the acceleration force.

Where to put my custom function for movement?

With all the different player scripts, it's understandable that you can be a bit lost. Here is a reminder of the different role of each script:

- **PlayerStateController**: holds the current data and functions for basic movement (horizontal movement). If you need to add a property accessible for any movement state, add it here. If you need a function to be performed whatever the current state, add it here.
- **PlayerStateFactory**: each time the character changes state, this script creates a new State from the state data. This way, even if you modify the current state, when you'll go back to it, it'll be as new. If you want to change the state values upon creation, change the state date in the PlayerStateFactory. If you need to perform some function when creating a state, add them here.
- **PlayerBaseState**: this is an abstract class inherited by every state. If you add something here, there's a good chance that you have to change every state accordingly. Do not touch it unless you want to modify the structure of all the state.
- **PlayerXState**: Movement states that inherits from PlayerBaseState (X=Walk, Run etc.) If you want a new behavior for the player, create a new state and add all the functions you want. Do not use Unity's default *Update()* or *FixedUpdate()* functions but rather *UpdateState()* and *FixedUpdateState()* inherited from PlayerBaseState and called from the PlayerStateController.

Contact

If you need help, have any questions or just want to say thank you, you can contact me via email at TODO.

Be aware that I am a single developer working on this asset so I might take some time to respond but will do as soon as I can.