互動程式設計III

Interactive Programming Design Integration

Introduction to Player Control

- In this lesson you'll learn:
  - How to develop a structured player controller for top-down view gameplay.
  - How to apply key design patterns for player controllers, including the Template Method and Observer patterns.
- Learning Objectives
  - Understand the fundamental principles of translation and rotation in game development.
  - Differentiate between moving a GameObject using Transform and Rigidbody.
  - Review the process of creating and managing an Animator for character animations.
  - Gain proficiency in coding GameObject movement effectively and efficiently.
- Why This Chapter is Important
  - Core Character Functionality: Implement the essential mechanics that drive character interactions in gameplay.
  - Reusable Player Controllers: Design controllers that work seamlessly for both human players and NPCs.
  - Foundational Game Experience: Player control is a central aspect of the gaming experience, especially for character-driven games.
  - Genre-Specific Importance: In genres like 1st or 3rd shooters and action RPGs, the quality of player control is directly tied to user satisfaction and engagement.

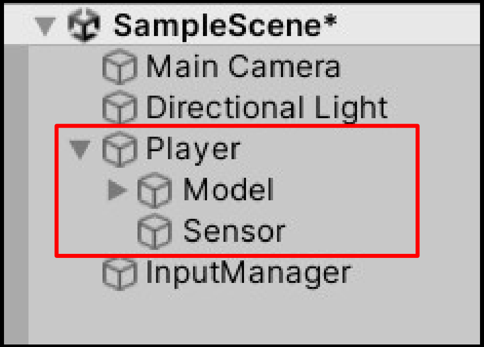
- Player Control
  - Player Structure
  - Planar Move
  - Fixed Camera
  - Block
  - Fall
  - Jump
  - Friction

互動程式設計III

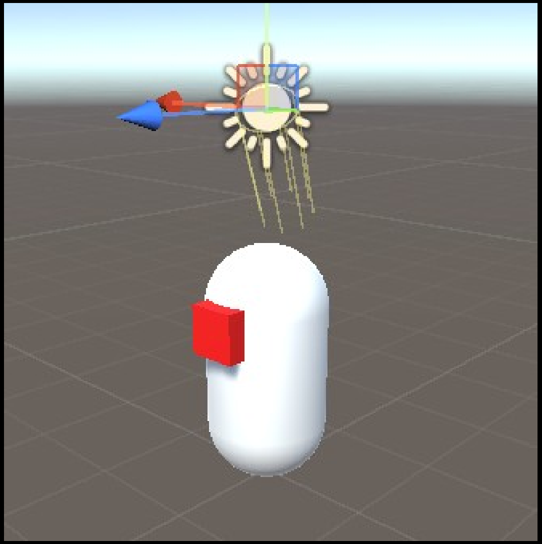
Interactive Programming Design Integration

Player Structure

- Purpose of Developing a Structured Player Controller:
  - Separation of Logic and Visual Representation: Decouple the underlying mechanics from the visual elements to maintain clarity and flexibility.
  - Seamless Collaboration: Facilitate smooth integration between the work of engineers and artists by defining clear boundaries between logic and visuals.
  - Reusability: Enable the reuse of the player structure across various character types, including both human players and NPCs.
  - Efficient Code Development: Ensure concise, maintainable code that simplifies testing and future updates.
- Core Components:
  - Player Handle: Serves as the primary container and collider, managing the physical presence and interactions of the player character.
  - Model: Represents the visual mesh of the character, which can be a static mesh or a skinned mesh for animations.
  - Sensor: Additional sensor components used to detect environmental elements, such as obstacles, triggers, or terrain changes.

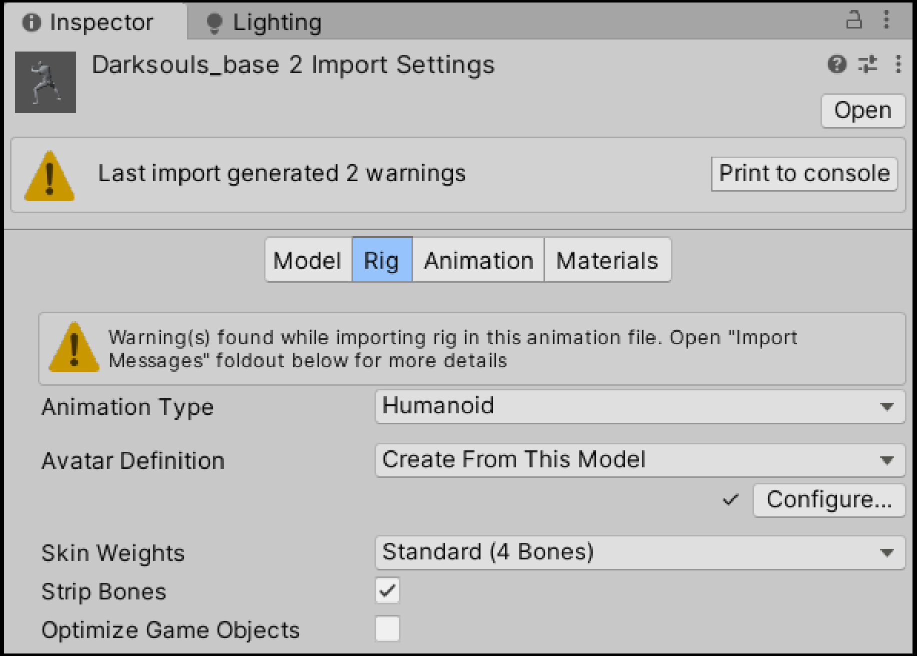


```
▼ SampleScene*
    Main Camera
    Directional Light
  ▼ Player
    ▶ Model
      Sensor
    InputManager
```



06_Resources.zip

- Animations
  - YBot animation from Mixamo.com
  - darksouls_base2.fbx
    - Set import rig to Humanoid
    - Avatar: create from this model



Inspector | Lighting

Darksouls_base 2 Import Settings

Open

Last import generated 2 warnings

Print to console

Model | Rig | Animation | Materials

Warning(s) found while importing rig in this animation file. Open "Import Messages" foldout below for more details

Animation Type: Humanoid

Avatar Definition: Create From This Model

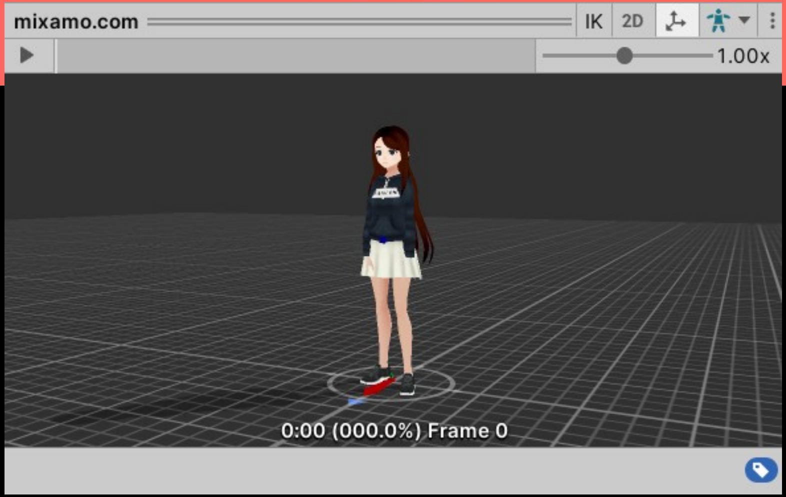
✓ Configure...

Skin Weights: Standard (4 Bones)

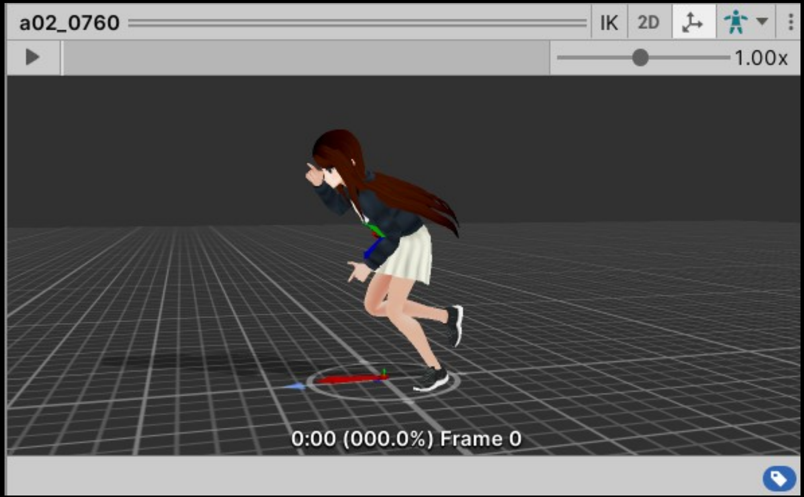
Strip Bones: ☑

Optimize Game Objects: □

https://assetstore.unity.com/packages/3d/characters/humanoids/casual-1-anime-girl-characters-185076



YBot idle



DS a02_0760 Claymore

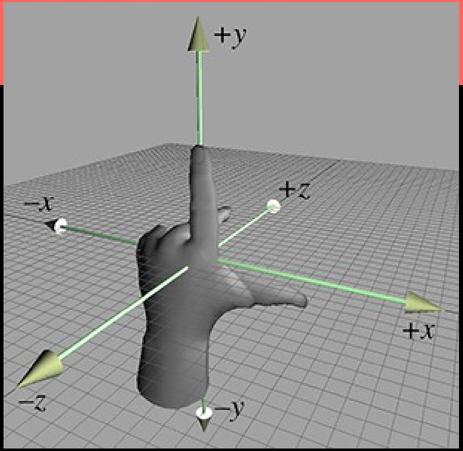
- Player Control
  - Player Structure
  - Planar Move
  - Fixed Camera
  - Block
  - Fall
  - Jump
  - Friction

互動程式設計III

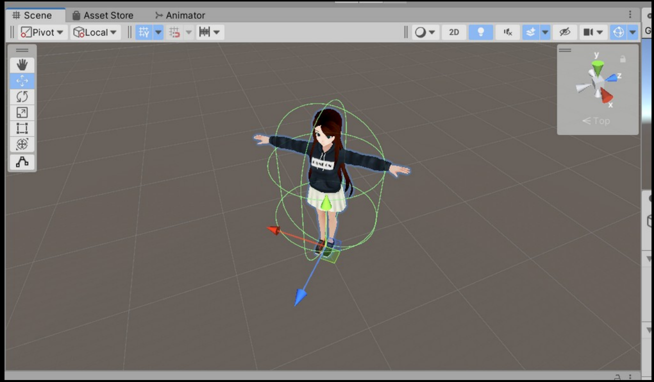
Interactive Programming Design Integration

Axis in Unity3D

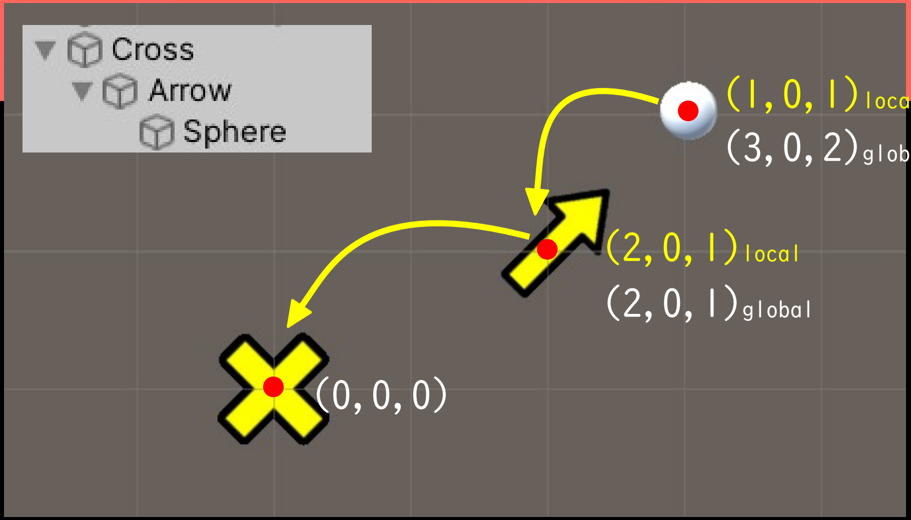
- Direction Rules: Left-Hand Rule [*]
  - Unity3D's Left-Handed Coordinate System:
    - In Unity3D, the X, Y, and Z axes follow the left-hand rule, where each axis corresponds to a finger:
      - Thumb: X-axis; Index Finger: Y-axis; Middle Finger: Z-axis
- Global vs. Local Axis [*]
  - Global Axis: Defines the coordinate system for the entire scene. Displayed in the top-left corner of the Unity editor, providing a reference for all objects within the scene.
  - Local Axis: Represents the coordinate system specific to individual GameObjects. Visualized through RGB axis handles (red for X, green for Y, blue for Z) when selecting a GameObject.
- Axis Modes and Gizmo Settings
  - Local/Global Axis Mode:
    - Switch between local and global modes for manipulating GameObject axis gizmos, affecting how objects are transformed.
  - Pivot vs. Center Mode:
    - Pivot Mode: Positions the Gizmo at the actual pivot point of the object's mesh, often used for precise manipulation.
    - Center Mode: Places the Gizmo at the center of the GameObject's rendered bounds, useful for overall object alignment.



How the 3 axis being mapped to 3 fingers in left-handed system [*]



Positions in Unity3D



- Global Position (World Space)
  - Defines the object’s absolute position within the world’s coordinate system.
  - The global position remains constant regardless of the object’s parent or its position in the hierarchy.

```
Vector3 globalPos = transform.position;
```

- Local Position (Object Space)
  - Represents the object’s relative position to its parent coordinate system.
  - When the parent object moves, the local position stays unchanged, but the global position of the object updates accordingly.

```
Vector3 localPos = transform.localPosition;
```

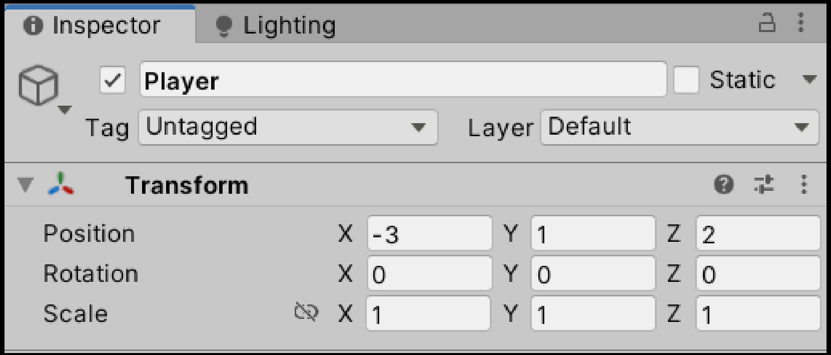
- Relationship Between Global and Local Position:
  - Global Position = Parent’s Global Position + Local Position
  - In a hierarchy, the local position determines the object’s placement relative to its parent, influencing how it behaves when the parent moves or rotates.
  - Transform.position always returns the global position of the object, while Transform.localPosition gives its position relative to its parent.

Translate

- If the gameObject does NOT have a Rigidbody component, you can directly modify its position using the Transform component. This takes effect immediately!! (without noticing physics engine)
  - Direct position assignment
    - Transform.position [*]
  - Translation
    - Transform.Translate() [*]
- If your gameObject DO have a Rigidbody component, you should use Rigidbody to translate.
  - Direct position assigment (like teleporting)
    - Rigidbody.position [*]
  - Sweeping
    - Rigidbody.MovePosition [*]
    - Rigidbody.velocity [*] (migrate to linearVelocity in Unity6)
- !!!Impact on Performance and Why Use Rigidbody.Position
  - If you use Rigidbody.position or Rigidbody.MovePosition, the physics engine:
    - Defers recalculations to the next fixed physics update (in FixedUpdate()).
    - Avoids recalculating the collider’ s transform immediately.
    - Keeps the collider data more efficiently synchronized with the rigidbody’s position.
  - If you use Transform.position:
    - The transform and attached colliders are updated immediately outside the physics system’ s fixed step, causing more overhead.

Using Transform to Translate

- Direct Position Assignment
  - Assigns a new position to the GameObject instantly, without considering physics.
  - Use this method when you need to move objects without Rigidbody components.
    - Transform.position [*]
- Translation Using Transform.Translate()
  - Moves the GameObject by a specified vector relative to its current position.
  - Can apply movement in local or world space, depending on the parameters.
    - Transform.Translate [*]
- Use Time.deltaTime for frame-independent movement, ensuring consistent behavior across different frame rates.



☑ Player □ Static

Tag Untagged Layer Default

Position X -3 Y 1 Z 2

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

Using Rigidbody to Translate

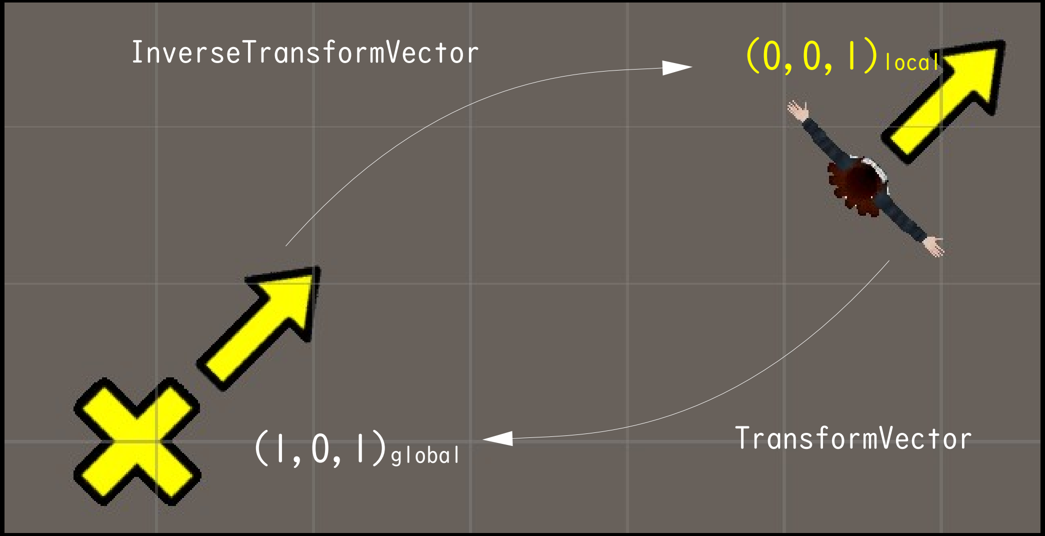
- Teleportation with Rigidbody.position
  - Directly assigns a new position to the GameObject, bypassing the physics system.
  - Use this for instantaneous movement (e.g., teleportation) without triggering any physics interactions.
    - Rigidbody.position [*]
- Sweeping with Rigidbody.MovePosition()
  - Moves the GameObject smoothly between two positions over time.
  - Ensures that the movement interacts properly with physics (e.g., colliders).
    - Rigidbody.MovePosition [*]
- Applying Velocity with Rigidbody.velocity
  - Sets the linear velocity of the Rigidbody, causing it to move based on physics rules.
  - Note: In Unity6, Rigidbody.velocity is replaced by linearVelocity.
    - Rigidbody.velocity [*]
- Key considerations
  - Rigidbody.MovePosition ensures smooth physics interactions, while Rigidbody.position should be used cautiously to avoid unexpected behavior.
  - Use Rigidbody.velocity for continuous motion that respects the physics system.

Rotate

- If the gameObject does NOT have a Rigidbody, then you can directly change the position of Transform. This takes effect immediately!! (without noticing physics engine)
  - Direct position assignment
    - Transform.rotation [*]: Sets the absolute rotation using a Quaternion.
    - Transform.eulaerAngles [*]: Sets the rotation using Euler angles (in degrees) along the X, Y, Z axes.
  - Rotate around an axis
    - Transform.Rotate() [*]: Applies a relative rotation (adds to the existing rotation).
- If the GameObject has a Rigidbody, it's recommended to rotate using Rigidbody methods to maintain proper physics behavior.
  - Direct position assigment (like teleporting)
    - Rigidbody.rotation [*]: Directly assigns a rotation (similar to teleporting).
    - Rigidbody.MoveRotation() [*]: Sweeps the rotation smoothly over time between physics frames.
  - Rotate around an axis
    - Rigidbody.angularVelocity [*]:Sets the rotational velocity in radians/second along the axes.
- Using Rigidbody.rotation or Rigidbody.MoveRotation():
  - Defers physics calculations to the next fixed update (FixedUpdate()).
  - Keeps collider synchronization efficient without immediate updates.
- Using Transform.rotation or Transform.Rotate():
  - Updates the rotation immediately outside the physics system.
  - May cause more overhead if used frequently on physics objects, as it forces instant recalculations.

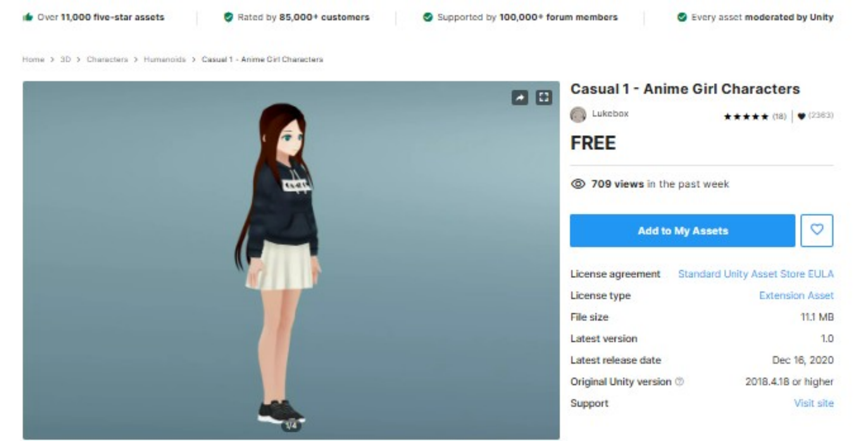
Coordinate Mapping

- When we want the character to move forward, we refer to moving along the (0, 0, 1) axis in its local coordinate system. However, what would this direction correspond to in the global coordinate system?
- From local to global
  - Transform.TransformVector [*]
- From global to local
  - Transform.InverseTransformVector [*]



EX: Player Structure

- Player handle:
  - Represent the true space volume that a player occupies. The position of the handle is the position of the player.
  - Owns a movement collider.
- Model:
  - The mesh model that shows the visual representation for the player.
- Sensor:
  - Extra sensor for different purposes. Ex: damage collider.



Over 11,000 five-star assets | Rated by 85,000+ customers | Supported by 100,000+ forum members | Every asset moderated by Unity

Home > 3D > Characters > Humanoids > Casual 1 - Anime Girl Characters

Casual 1 - Anime Girl Characters

Lukebox ★★★★★ (18) | ♥ (2363)

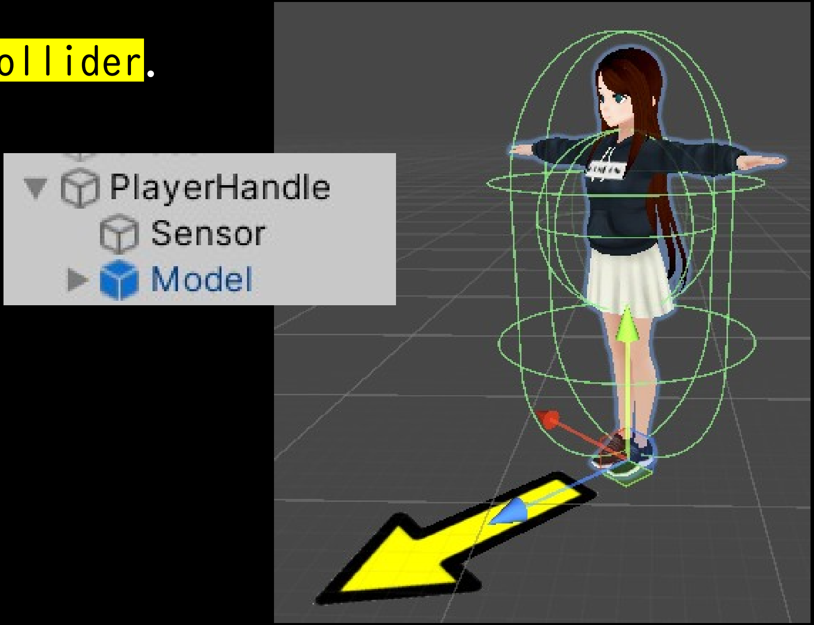
FREE

709 views in the past week

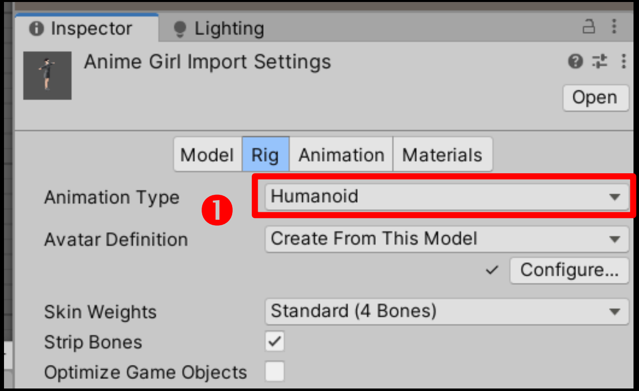
Add to My Assets

License agreement	Standard Unity Asset Store EULA
License type	Extension Asset
File size	11.1 MB
Latest version	1.0
Latest release date	Dec 16, 2020
Original Unity version	2018.4.18 or higher
Support	Visit site

Casual 1 - Anime Girl Characters [*]



- Since the casual1 mesh is imported as Humanoid rig❶, so we can apply humanoid animation to this model.
- Open the animator AC which is bundled with casual1❷. Drag our Ybot animation into the graph❸.
  - This will create a new state. Let's call this state walking and make it Layer Default State.
- Remember to check Loop Time.❹



Inspector | Lighting

Anime Girl Import Settings

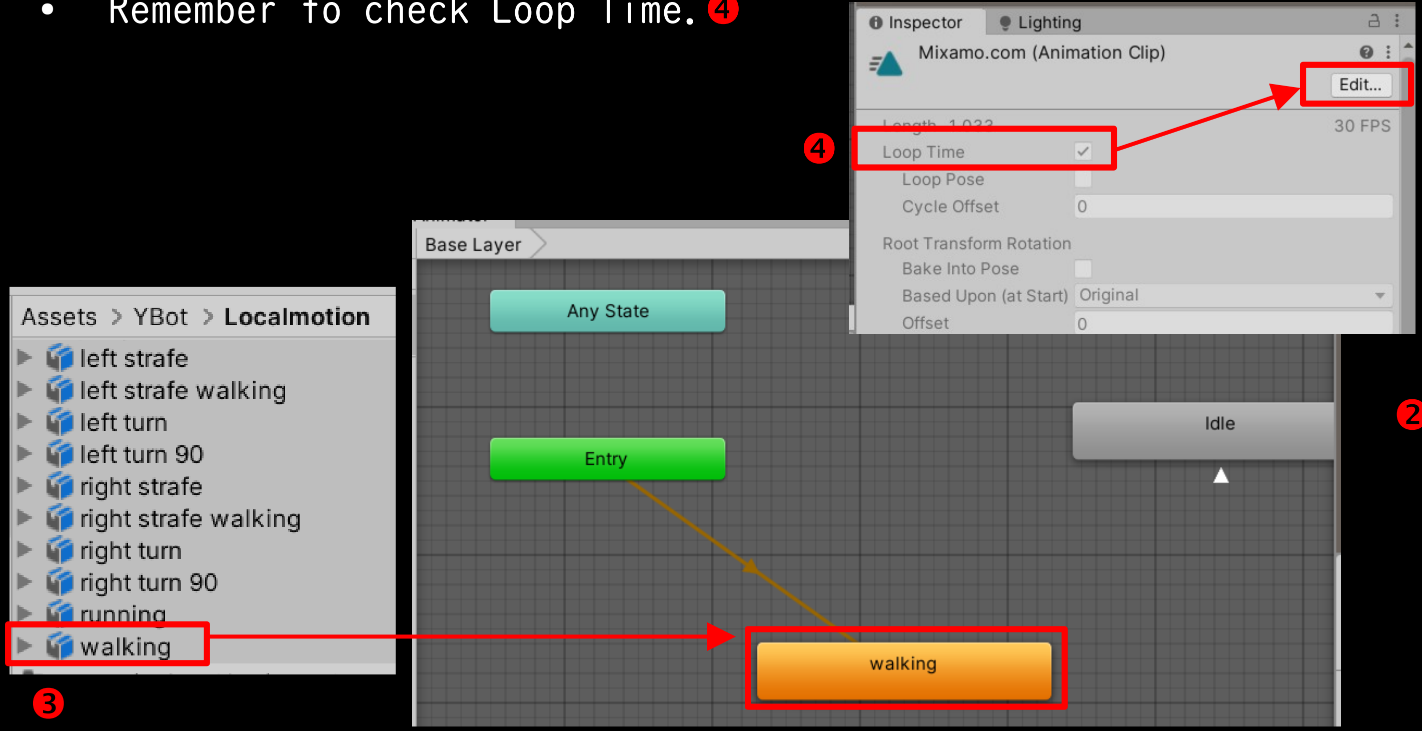
Animation Type: Humanoid

Avatar Definition: Create From This Model

Skin Weights: Standard (4 Bones)

Strip Bones: ☑

Optimize Game Objects: □



Inspector | Lighting

Mixamo.com (Animation Clip)

Length 1.033 30 FPS

Loop Time ☑

Loop Pose □

Cycle Offset 0

Root Transform Rotation

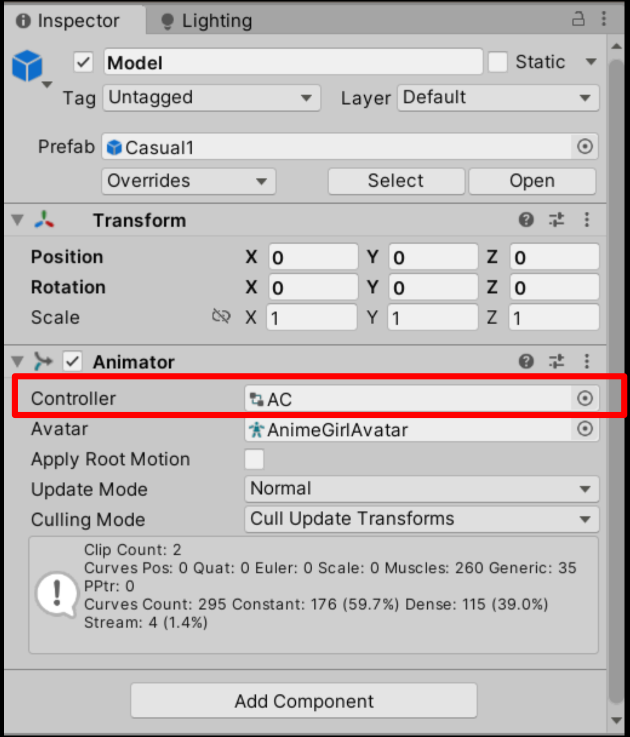
Bake Into Pose □

Based Upon (at Start) Original

Offset 0

Assets > YBot > Localmotion

- left strafe
- left strafe walking
- left turn
- left turn 90
- right strafe
- right strafe walking
- right turn
- right turn 90
- running
- walking



Inspector | Lighting

☑ Model □ Static

Tag Untagged Layer Default

Prefab Casual1

Transform

Position	X	0	Y	0	Z	0
Rotation	X	0	Y	0	Z	0
Scale	X	1	Y	1	Z	1

☑ Animator

Controller: AC

Avatar: AnimeGirlAvatar

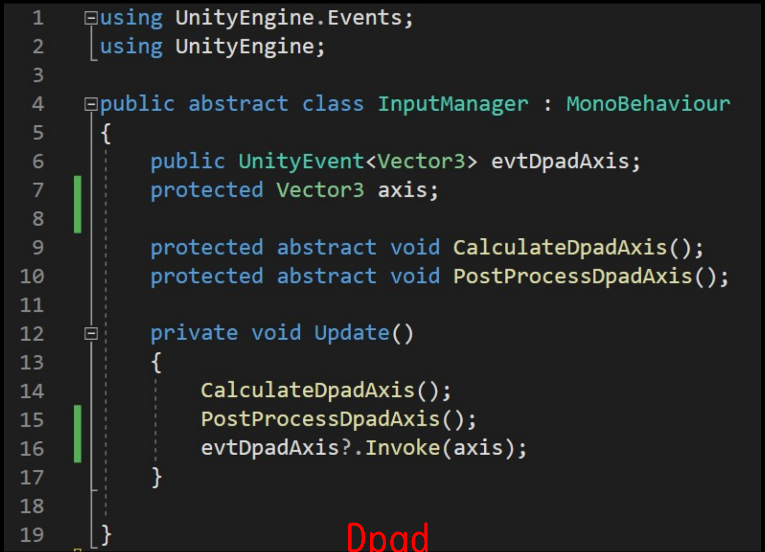
Apply Root Motion: □

Update Mode: Normal

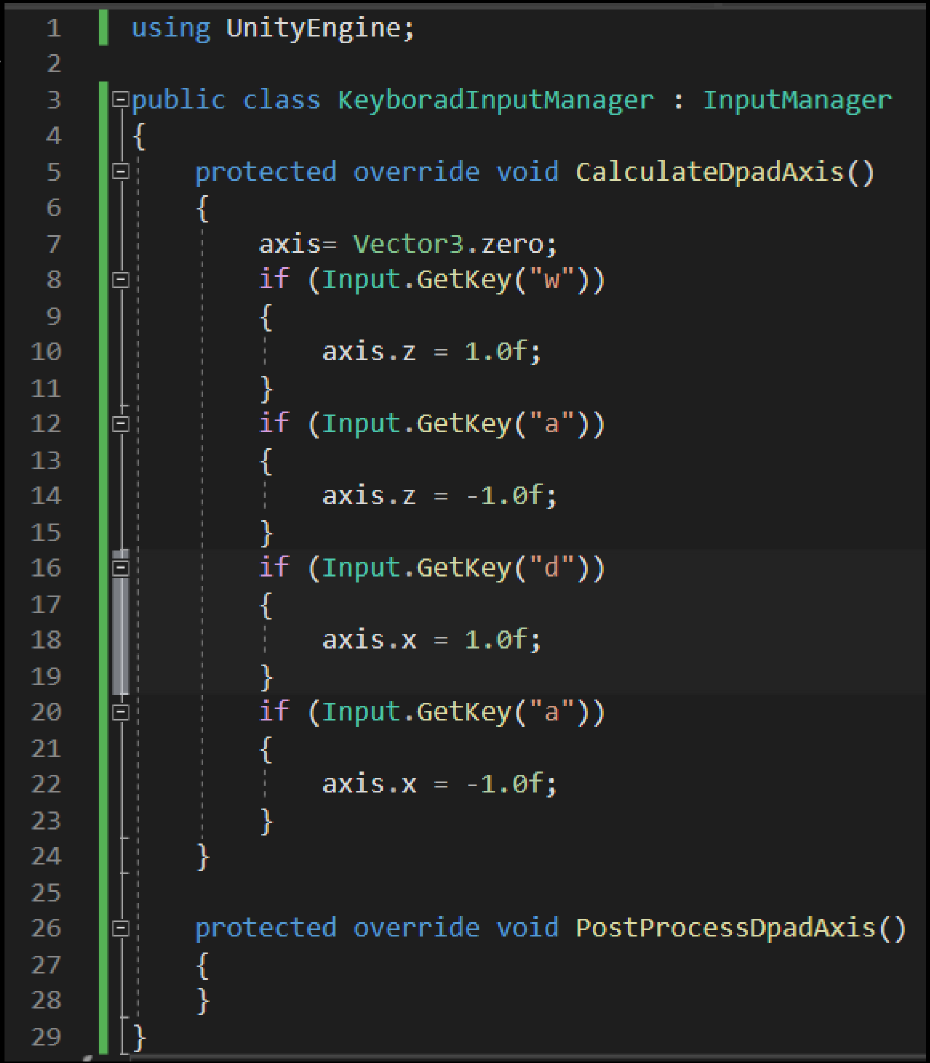
Culling Mode: Cull Update Transforms

Clip Count: 2
Curves Pos: 0 Quat: 0 Euler: 0 Scale: 0 Muscles: 260 Generic: 35 PPtr: 0
Curves Count: 295 Constant: 176 (59.7%) Dense: 115 (39.0%) Stream: 4 (1.4%)

- Design pattern: Template Method
  - The abstract class InputManager defines the process for deriving a vector of Dpad axis.
  - The implementation for keyboard input is in KeyboardInputManager.
- What is template method and why do we need it here?

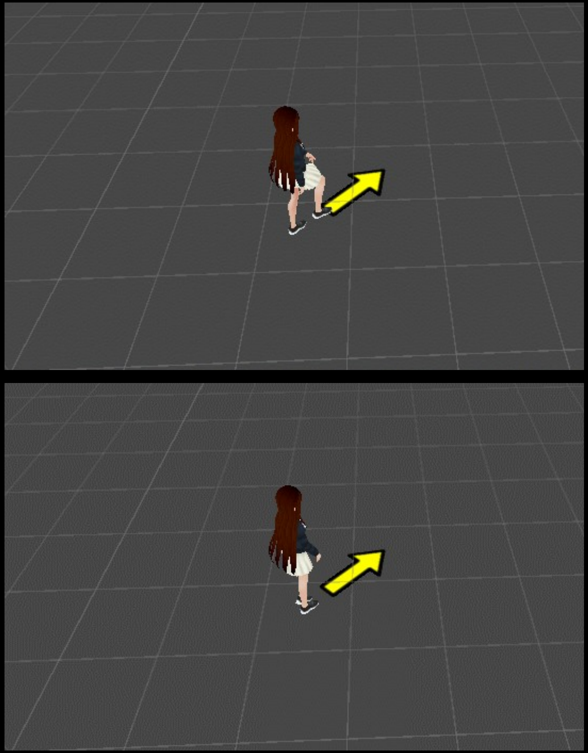


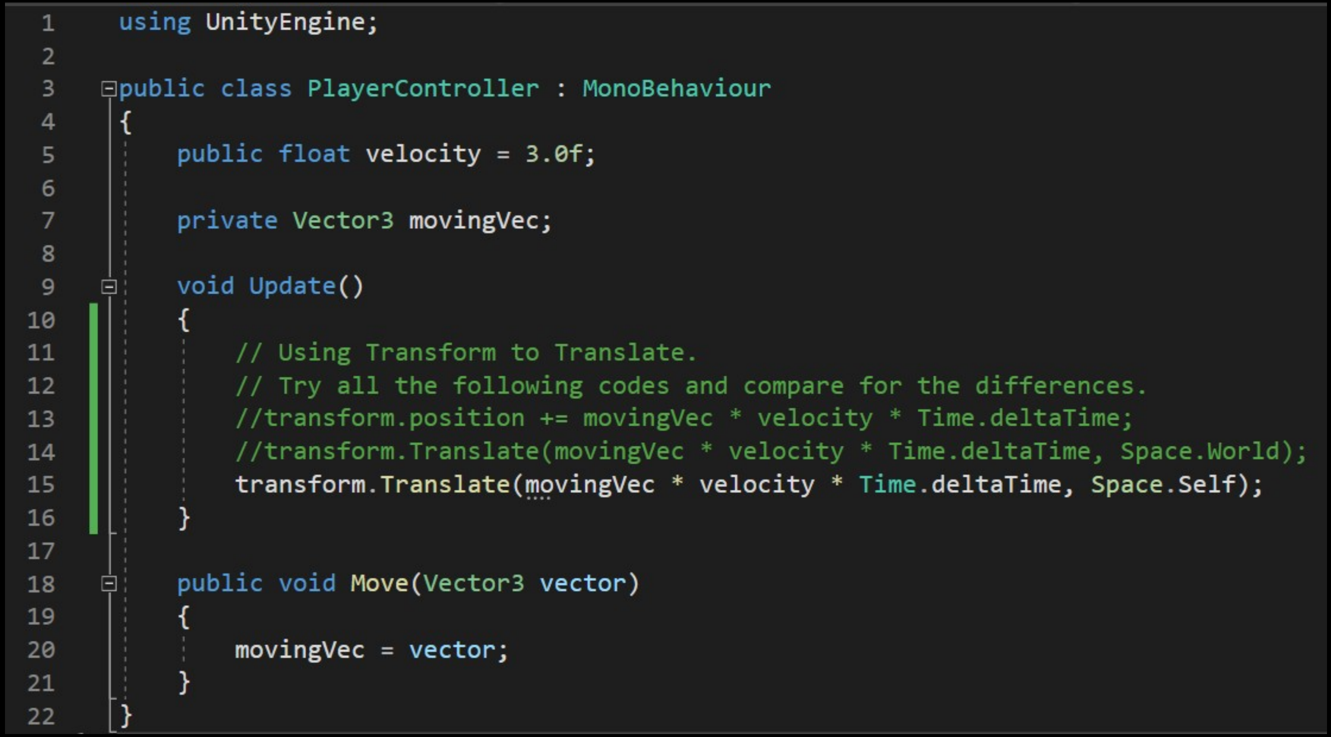
```
using UnityEngine.Events;
using UnityEngine;

public abstract class InputManager : MonoBehaviour
{
    public UnityEvent<Vector3> evtDpadAxis;
    protected Vector3 axis;

    protected abstract void CalculateDpadAxis();
    protected abstract void PostProcessDpadAxis();

    private void Update()
    {
        CalculateDpadAxis();
        PostProcessDpadAxis();
        evtDpadAxis?.Invoke(axis);
    }

}
```



```
using UnityEngine;

public class KeyboradInputManager : InputManager
{
    protected override void CalculateDpadAxis()
    {
        axis= Vector3.zero;
        if (Input.GetKey("w"))
        {
            axis.z = 1.0f;
        }
        if (Input.GetKey("a"))
        {
            axis.z = -1.0f;
        }
        if (Input.GetKey("d"))
        {
            axis.x = 1.0f;
        }
        if (Input.GetKey("a"))
        {
            axis.x = -1.0f;
        }
    }

    protected override void PostProcessDpadAxis()
    {
    }
}
```

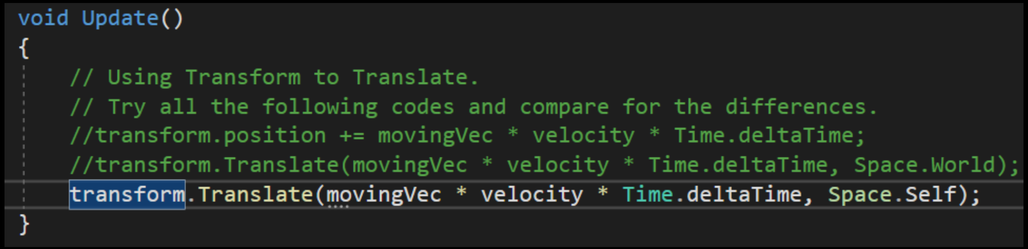
- PlayerController
  - Serves as a ”Motor” to the player character. It drives the whole handle to move.
  - Can be manipulated by another InputManager



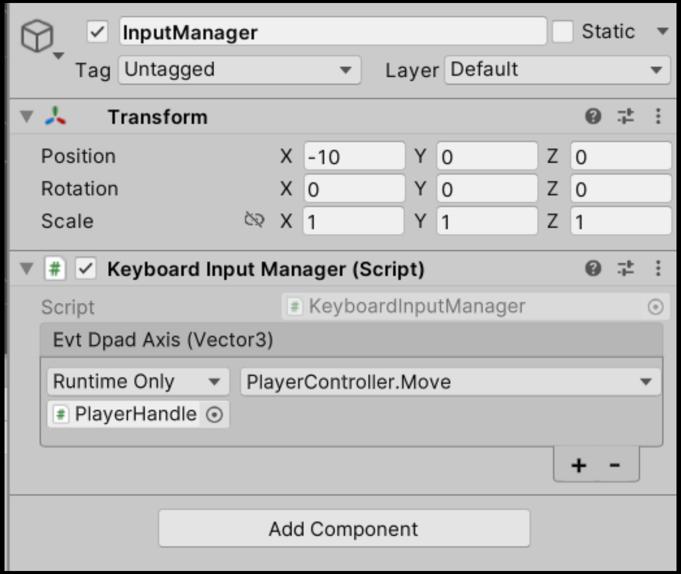


```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float velocity = 3.0f;

    private Vector3 movingVec;

    void Update()
    {
        // Using Transform to Translate.
        // Try all the following codes and compare for the differences.
        //transform.position += movingVec * velocity * Time.deltaTime;
        //transform.Translate(movingVec * velocity * Time.deltaTime, Space.World);
        transform.Translate(movingVec * velocity * Time.deltaTime, Space.Self);
    }

    public void Move(Vector3 vector)
    {
        movingVec = vector;
    }
}
```

- Configure and run.
- Describe what you see and found.



```
void Update()
{
    // Using Transform to Translate.
    // Try all the following codes and compare for the differences.
    //transform.position += movingVec * velocity * Time.deltaTime;
    //transform.Translate(movingVec * velocity * Time.deltaTime, Space.World);
    transform.Translate(movingVec * velocity * Time.deltaTime, Space.Self);
}
```



☑ InputManager □ Static

Tag Untagged Layer Default

Transform

Position X -10 Y 0 Z 0

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

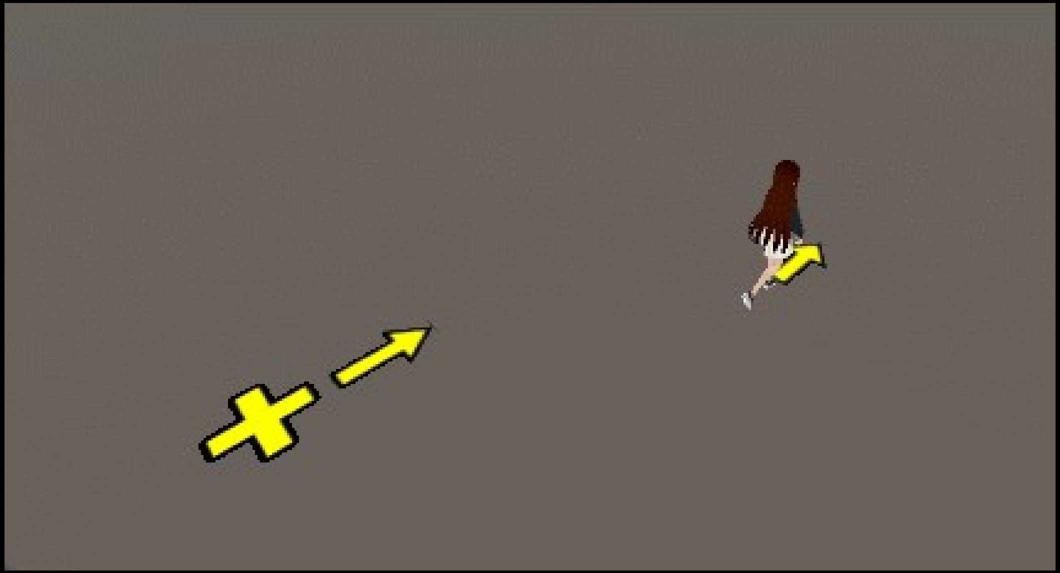
☑ Keyboard Input Manager (Script)

Script KeyboardInputManager

Evt Dpad Axis (Vector3)

Runtime Only PlayerController.Move

PlayerHandle



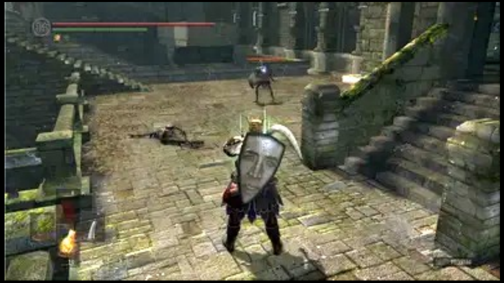
- Player Control
  - Player Structure
  - Planar Move
  - Fixed Camera
  - Block
  - Fall
  - Jump
  - Friction

互動程式設計III

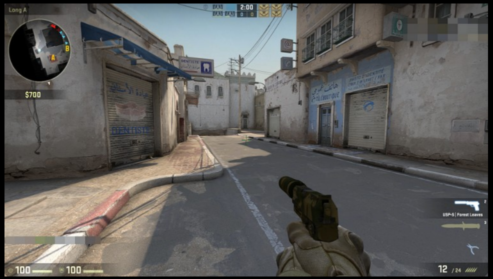
Interactive Programming Design Integration

The Symphony of Player Handle and Camera

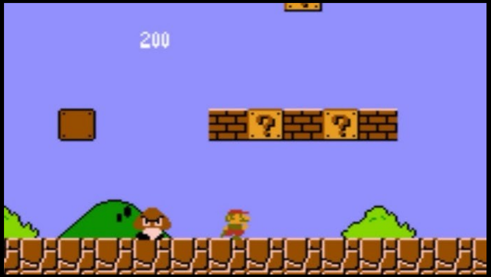
- The camera tracks the position of player handle. That means the handle is coworking very tightly with camera.
- When the camera renders its image and present to user, the user control the game according the the rendered image.
- So,
  - What will happen when the user presses up key?
  - What will happen when the user presses right key?



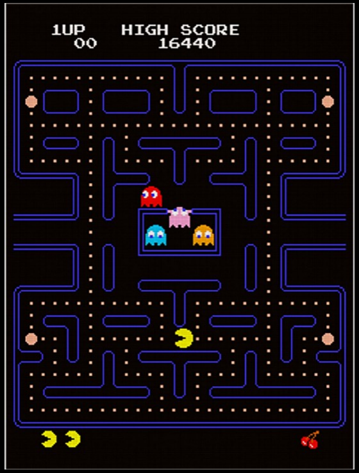
3rd person 3D lockon mode



1st person 3D free mode



3rd person 2D platform

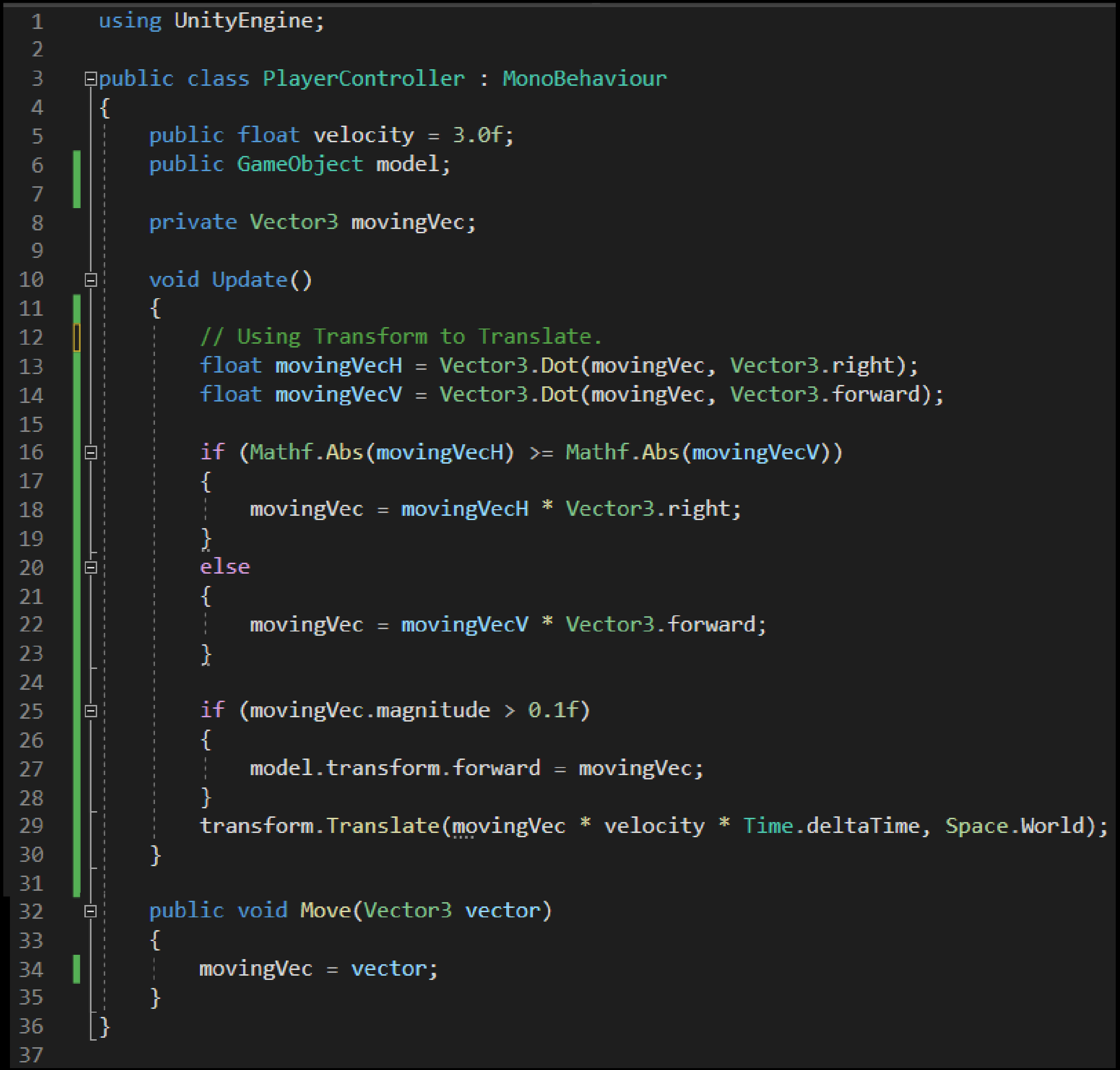


3rd person 2D top-down

EX: 2D Top-down

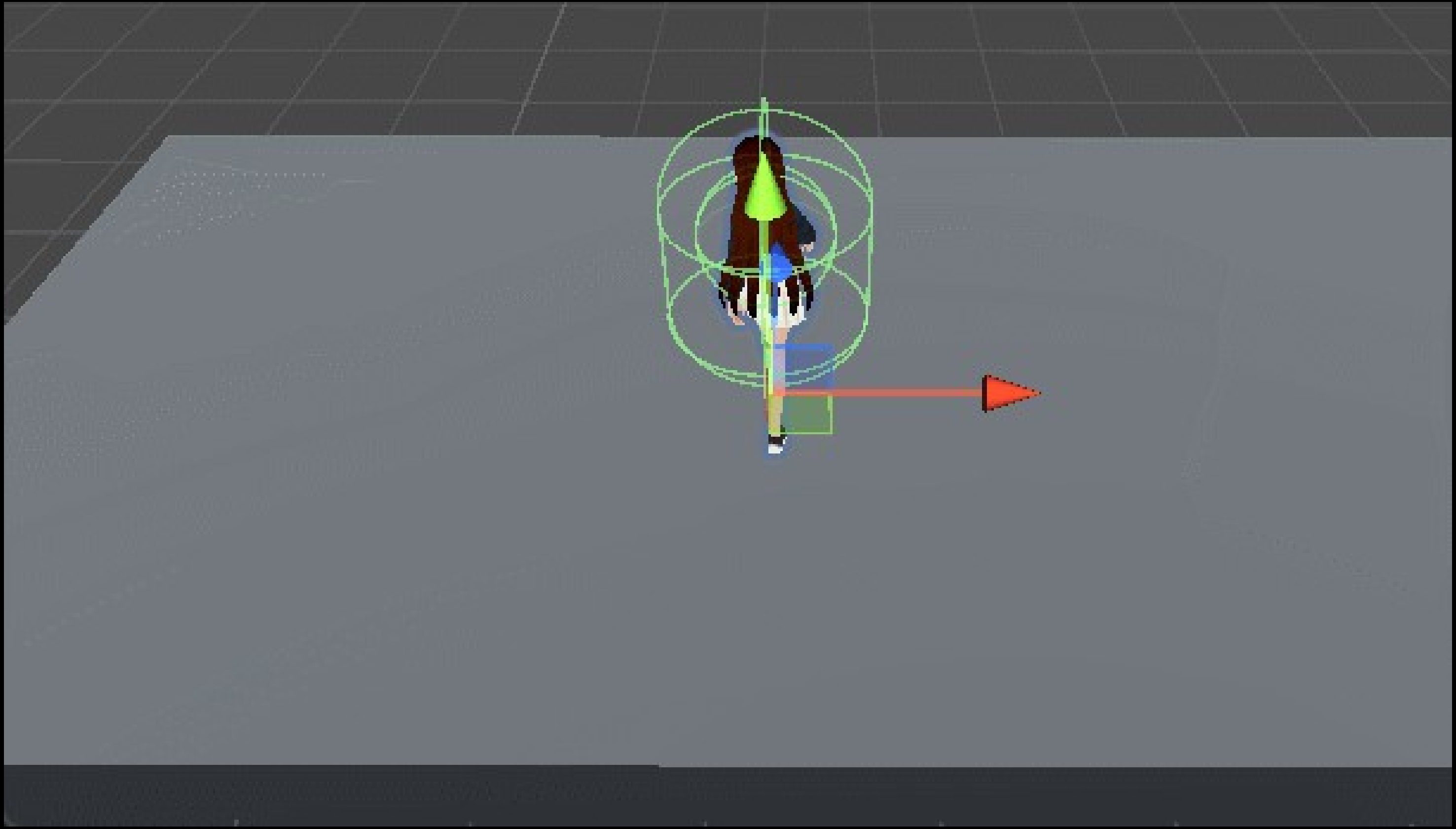
- So now the questions would be like: under 3rd 2D top-down,
  - What will happen when the user presses up key? Move north.
  - What will happen when the user presses right key? Move east.
- We are going to keep the handle fixed in angle. And just rotate the model inside.
- In order to avoid slanted movement, we are going to rewrite the CalculateDpadAxis method with else.

- Use inner product to detect which direction has a larger vector component.
- Keep only the directions pointing to forward/back/left/right.
- Rotate the inner model if movingVec is above threshold intensity. This avoid the model to ratate when there is no input signal.



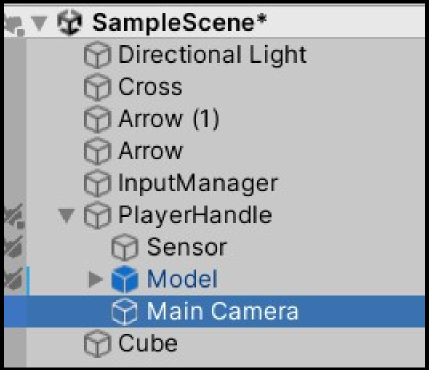
```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float velocity = 3.0f;
    public GameObject model;

    private Vector3 movingVec;

    void Update()
    {
        // Using Transform to Translate.
        float movingVecH = Vector3.Dot(movingVec, Vector3.right);
        float movingVecV = Vector3.Dot(movingVec, Vector3.forward);

        if (Mathf.Abs(movingVecH) >= Mathf.Abs(movingVecV))
        {
            movingVec = movingVecH * Vector3.right;
        }
        else
        {
            movingVec = movingVecV * Vector3.forward;
        }

        if (movingVec.magnitude > 0.1f)
        {
            model.transform.forward = movingVec;
        }
        transform.Translate(movingVec * velocity * Time.deltaTime, Space.World);
    }

    public void Move(Vector3 vector)
    {
        movingVec = vector;
    }
}
```

- Try it

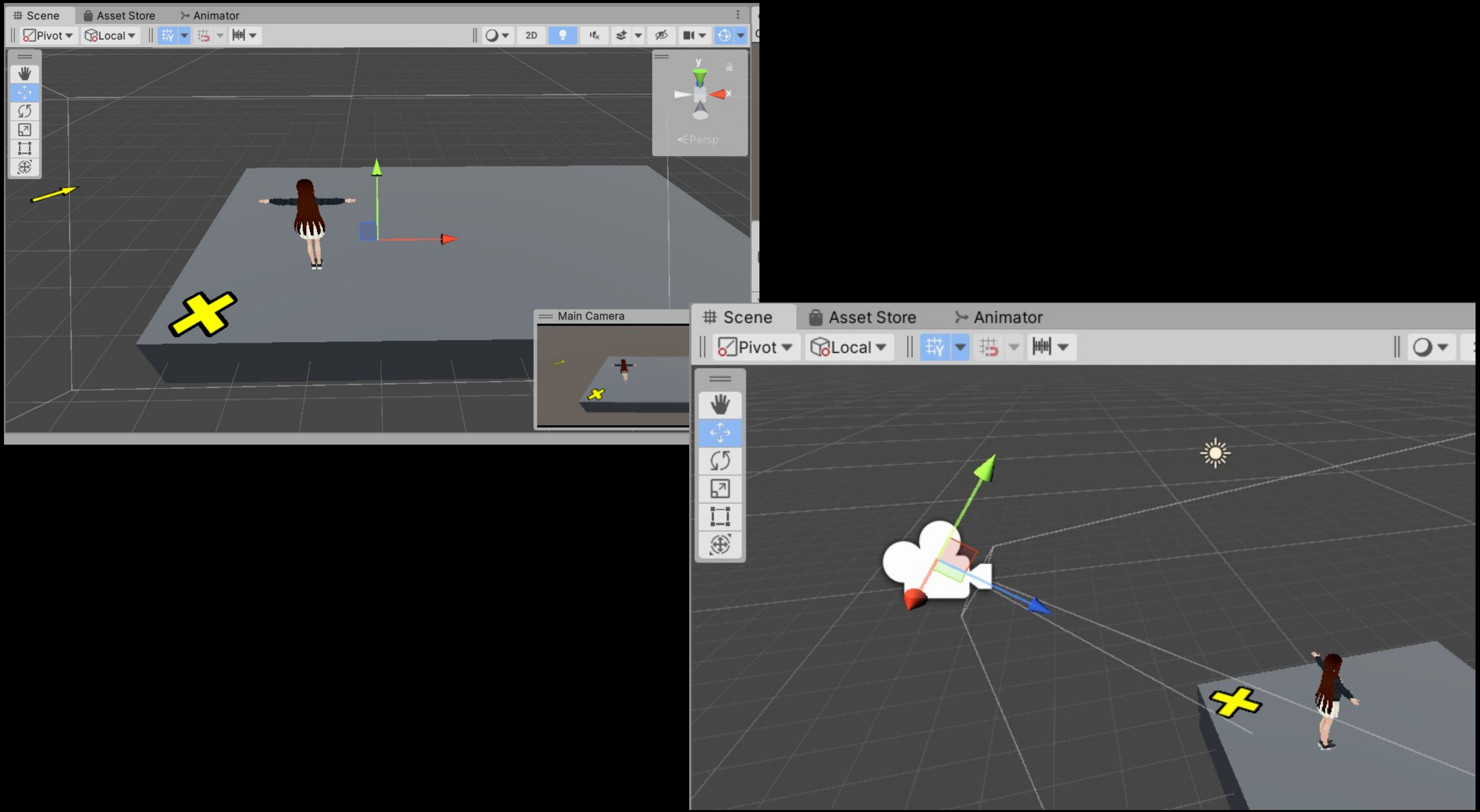


EX: 2D Top-down Camera

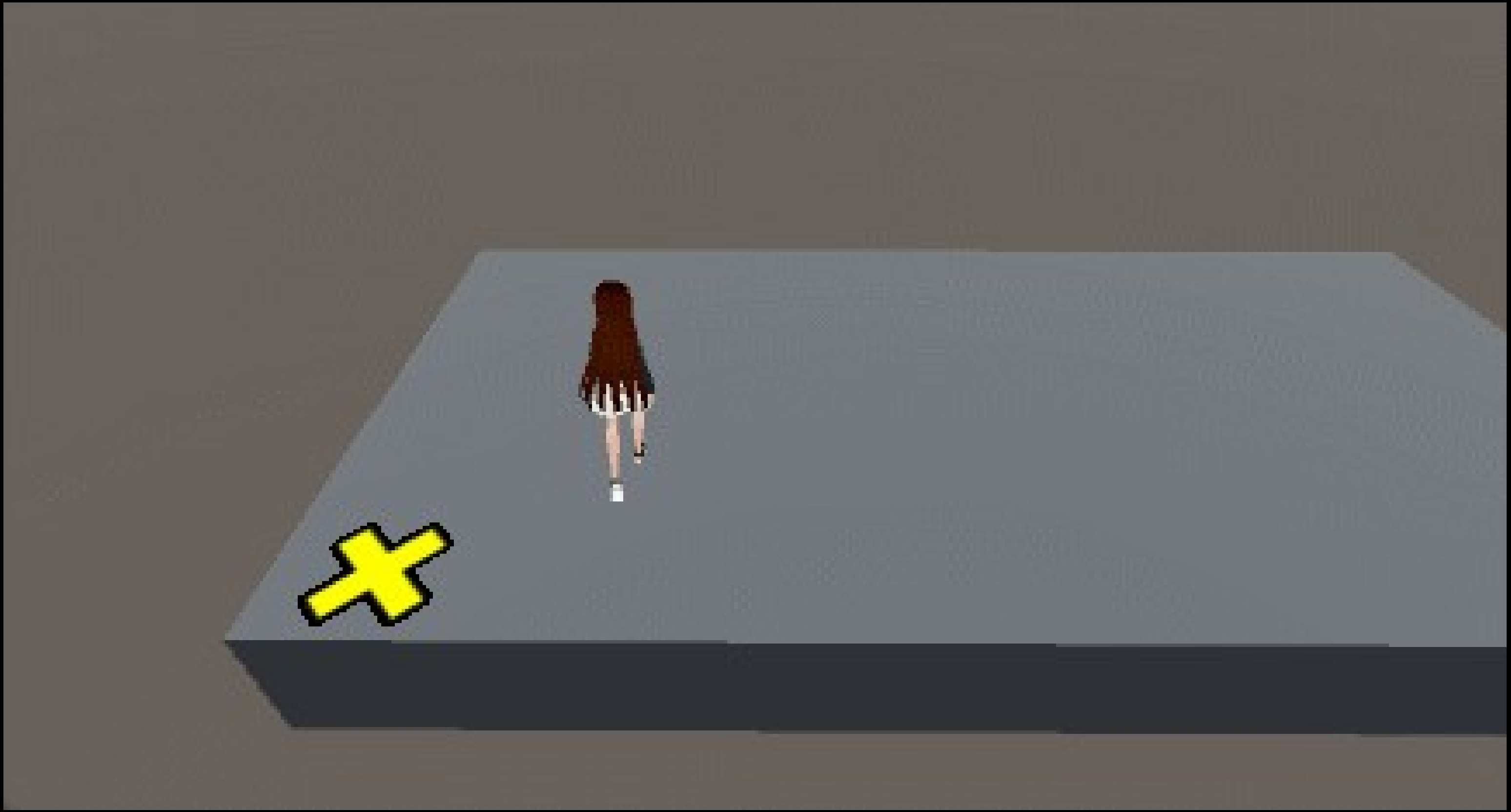
- The simplest way implement a following camera in 2D top-down view is to drag the camera into player handle.



- SampleScene*
  - Directional Light
  - Cross
  - Arrow (1)
  - Arrow
  - InputManager
  - PlayerHandle
    - Sensor
    - Model
    - Main Camera
  - Cube

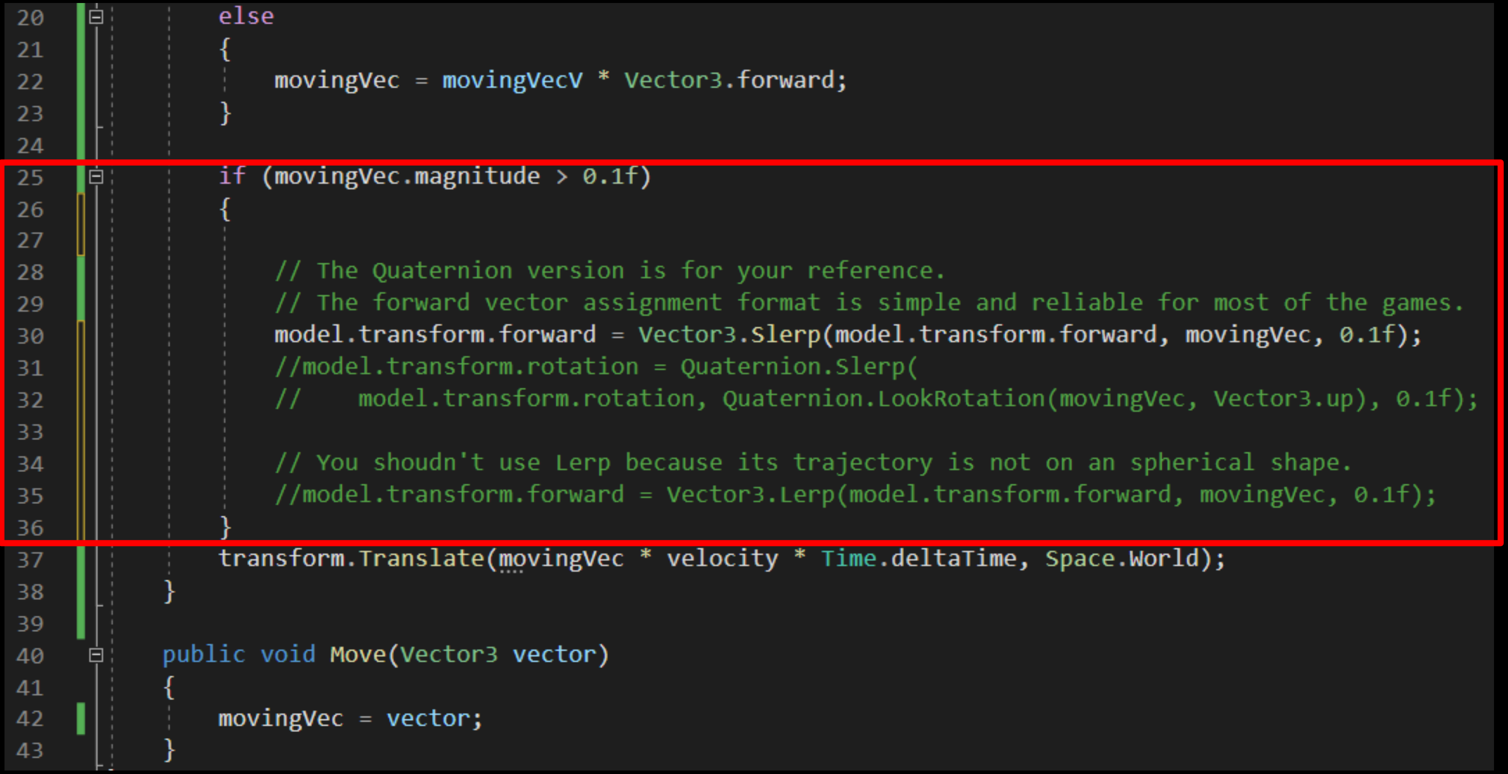


- Try it



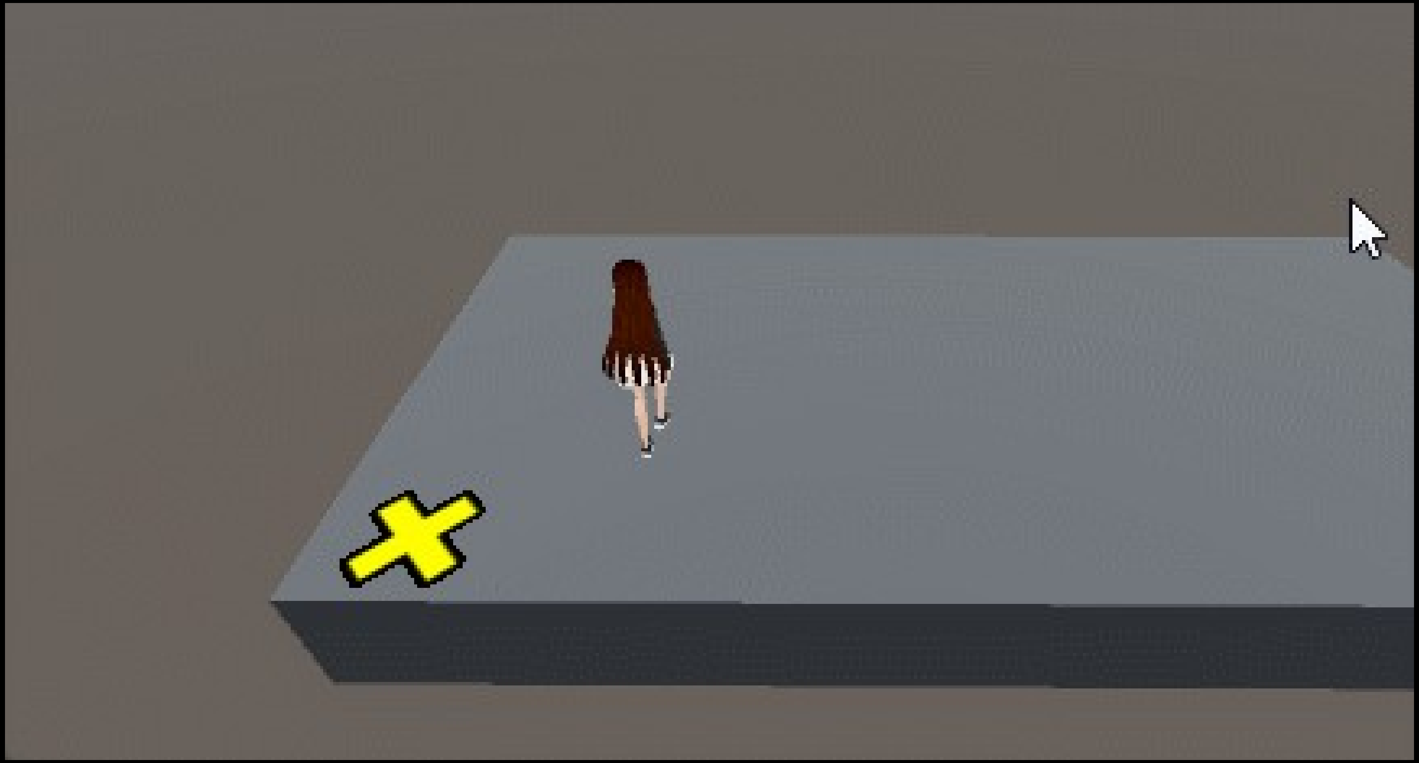
EX: Smooth Model Rotate

- In order to rotate the model smoothly, we use Slerp ().
- You find some exmaple on internet using Quaternion. But in most of your game project, dierectly assigning transform.forward is fast and reliable enough.



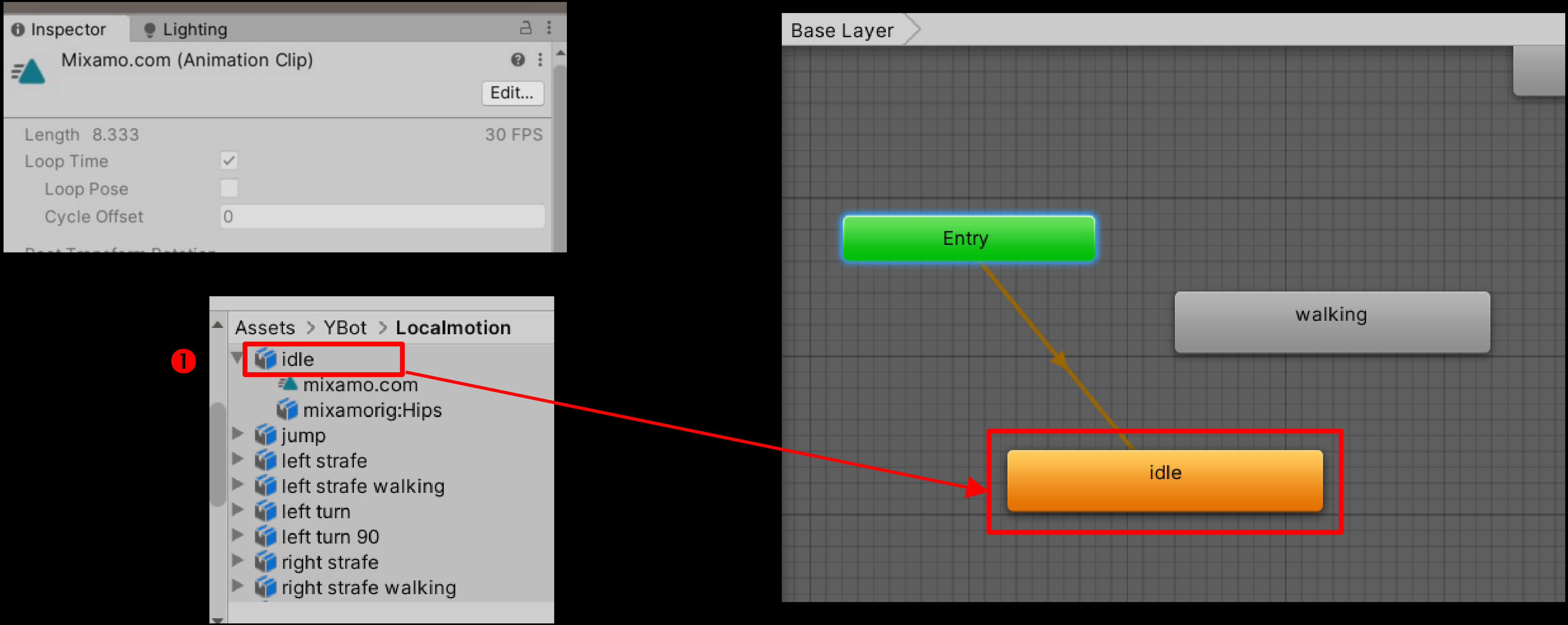
```
        else
        {
            movingVec = movingVecV * Vector3.forward;
        }

        if (movingVec.magnitude > 0.1f)
        {

            // The Quaternion version is for your reference.
            // The forward vector assignment format is simple and reliable for most of the games.
            model.transform.forward = Vector3.Slerp(model.transform.forward, movingVec, 0.1f);
            //model.transform.rotation = Quaternion.Slerp(
            //    model.transform.rotation, Quaternion.LookRotation(movingVec, Vector3.up), 0.1f);

            // You shoudn't use Lerp because its trajectory is not on an spherical shape.
            //model.transform.forward = Vector3.Lerp(model.transform.forward, movingVec, 0.1f);
        }
        transform.Translate(movingVec * velocity * Time.deltaTime, Space.World);
    }

    public void Move(Vector3 vector)
    {
        movingVec = vector;
    }
```

- Try it. You may fine-tune your turning angle speed.



EX: Add Idle State

- Find the YBot idel animation and drag it to the animator AC.❶
- Name the new state idel and set it to default.
- Remember to check loop time.



Inspector | Lighting

Mixamo.com (Animation Clip)

Edit...

Length 8.333 30 FPS

Loop Time ☑

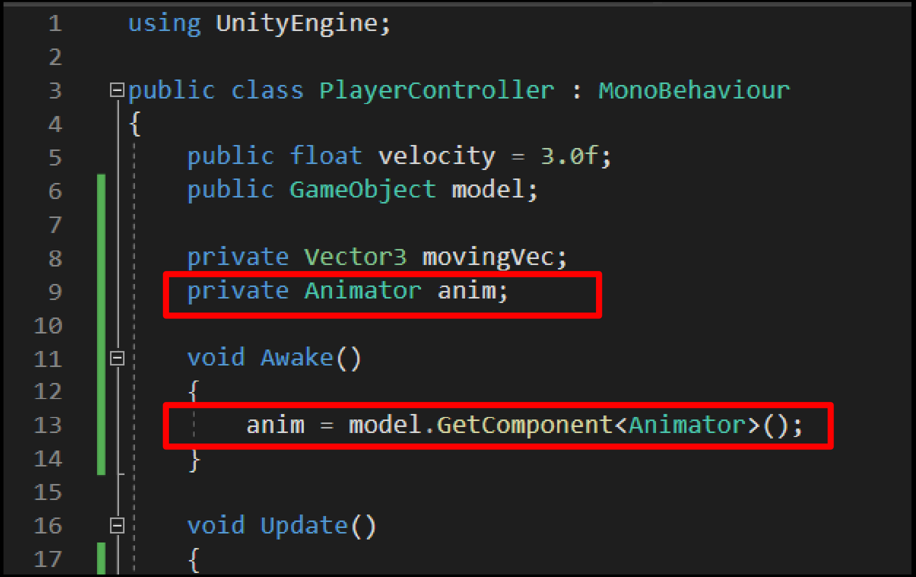
Loop Pose ☐

Cycle Offset 0

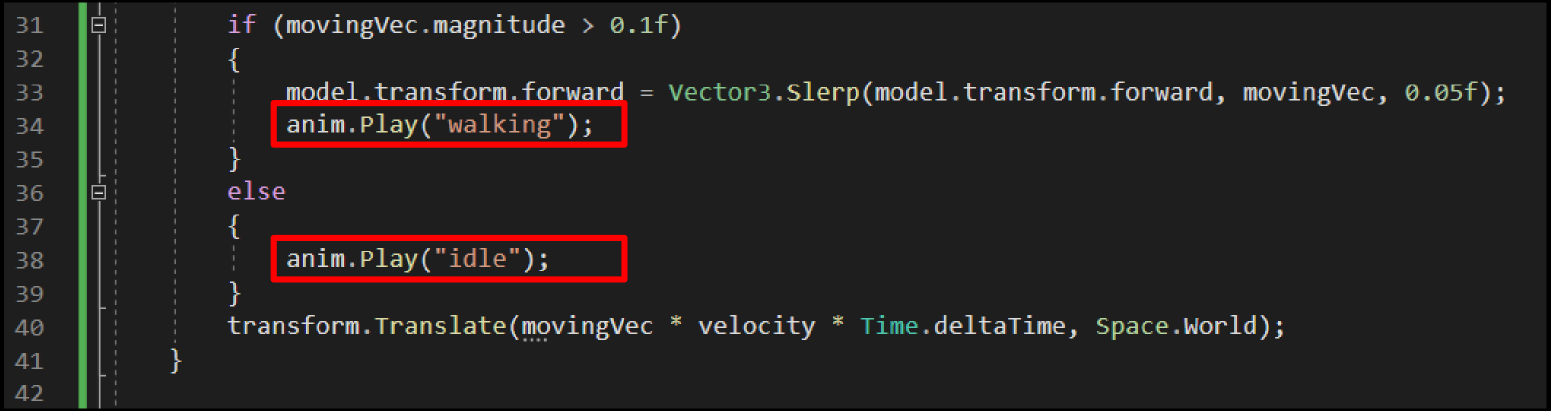
Assets > YBot > Localmotion

❶

- Let make it a ”hard transition” (this means the transition is abrupt)
- You can switch animator state by calling animator.Play()

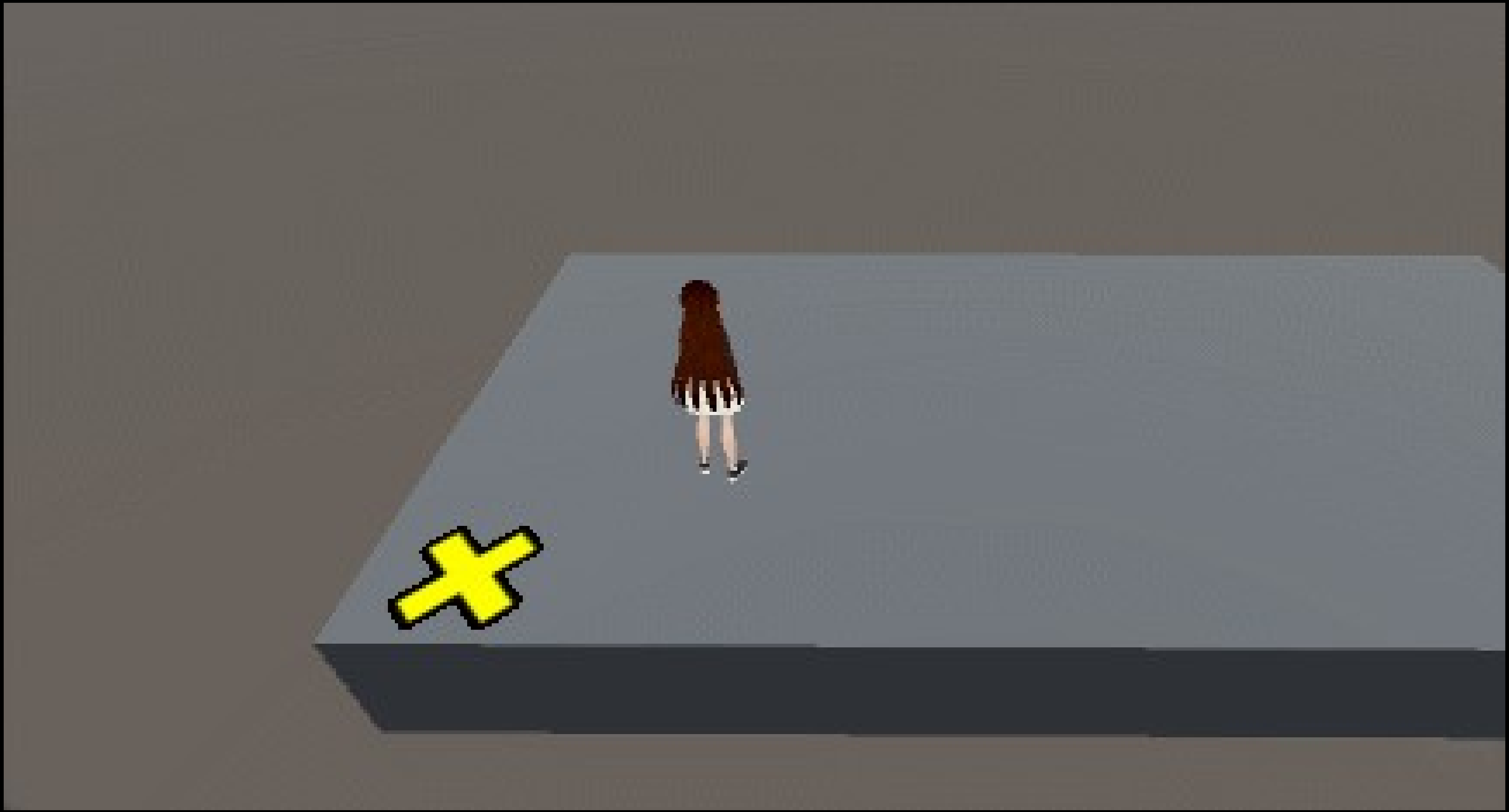


```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float velocity = 3.0f;
    public GameObject model;

    private Vector3 movingVec;
    private Animator anim;

    void Awake()
    {
        anim = model.GetComponent<Animator>();
    }

    void Update()
    {
```



```
            if (movingVec.magnitude > 0.1f)
            {
                model.transform.forward = Vector3.Slerp(model.transform.forward, movingVec, 0.05f);
                anim.Play("walking");
            }
            else
            {
                anim.Play("idle");
            }
            transform.Translate(movingVec * velocity * Time.deltaTime, Space.World);
        }

```

- Try it.



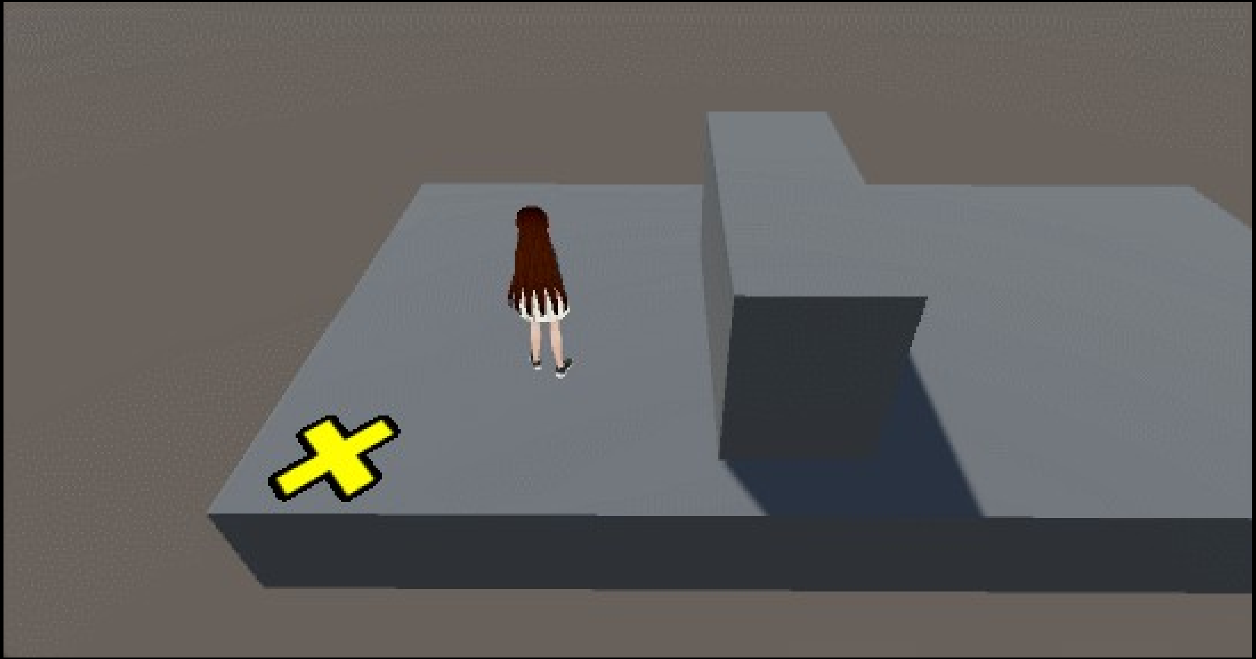
- Player Control
  - Player Structure
  - Planar Move
  - Fixed Camera
  - Block
  - Fall
  - Jump
  - Friction

互動程式設計III

Interactive Programming Design Integration

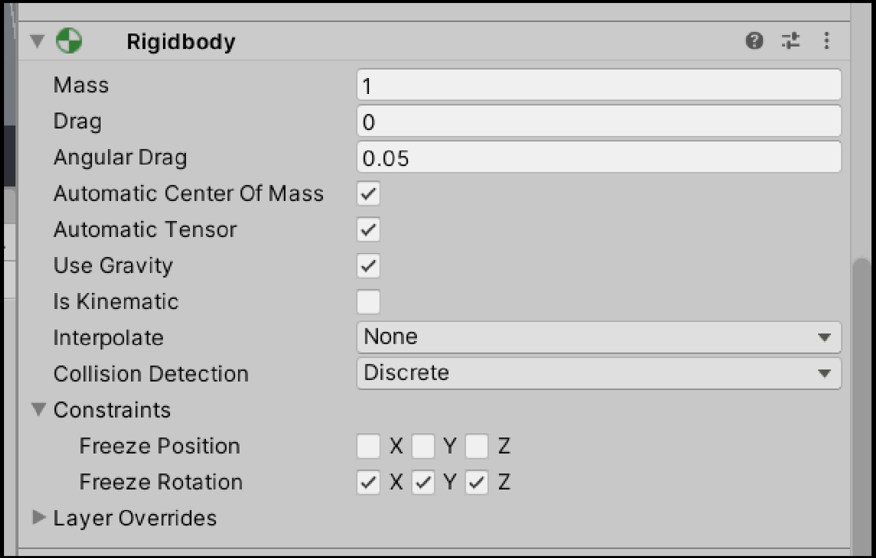
Nothing Can Stop Our Player Now

- If we put a huge cube (containing a Box Collider component) as a blocker in front of the player. You'll find that the blocker is not able to block the player's move.
- To block the player, we need the help from Unity's Physics Engine.
  - To update the positions of all colliders in the level.
  - To push-out solid colliders which shouldn't overlap together.



Add a RigidBody to Player Handler

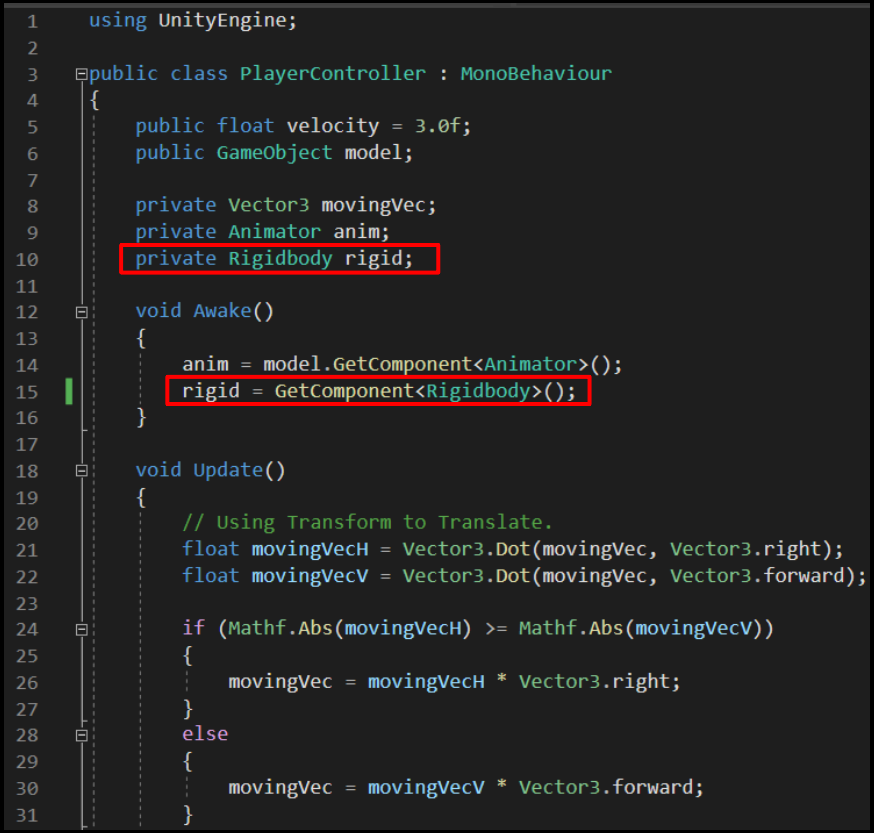
- The Rigidbody component synchronizes the GameObject with the physics engine, enabling realistic movement and interactions with other objects.
- Essential for handling collisions, gravity, and physics-based motion.
- Freezing Rotation for Better Control
  - Freeze X, Y, Z Rotation to prevent the handle from being rotated by physics forces, such as friction with other objects.
  - This allows rotation to be fully controlled by scripts, ensuring precise character control.
- Why Freezing Rotation is Important:
  - Prevents unwanted rotation caused by external forces (e.g., friction from collisions).
  - Ensures the handle rotates only when explicitly assigned by your script, improving gameplay precision.



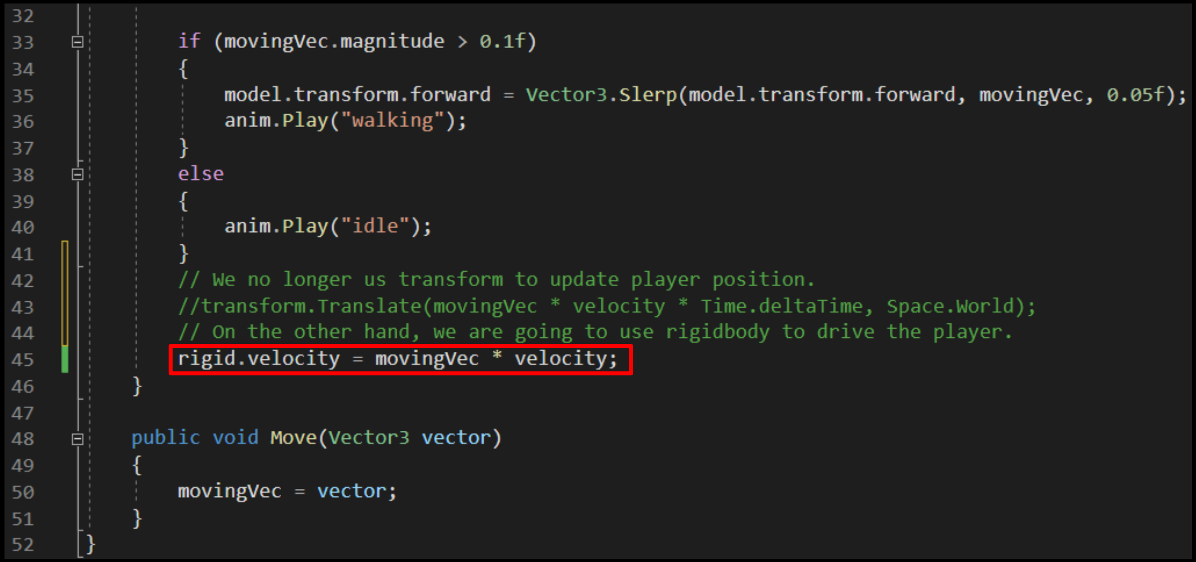
Rigidbody

Mass: 1
Drag: 0
Angular Drag: 0.05
Automatic Center Of Mass: ✓
Automatic Tensor: ✓
Use Gravity: ✓
Is Kinematic: □
Interpolate: None
Collision Detection: Discrete
Constraints
Freeze Position: □ X □ Y □ Z
Freeze Rotation: ✓ X ✓ Y ✓ Z
Layer Overrides

- Update the codes

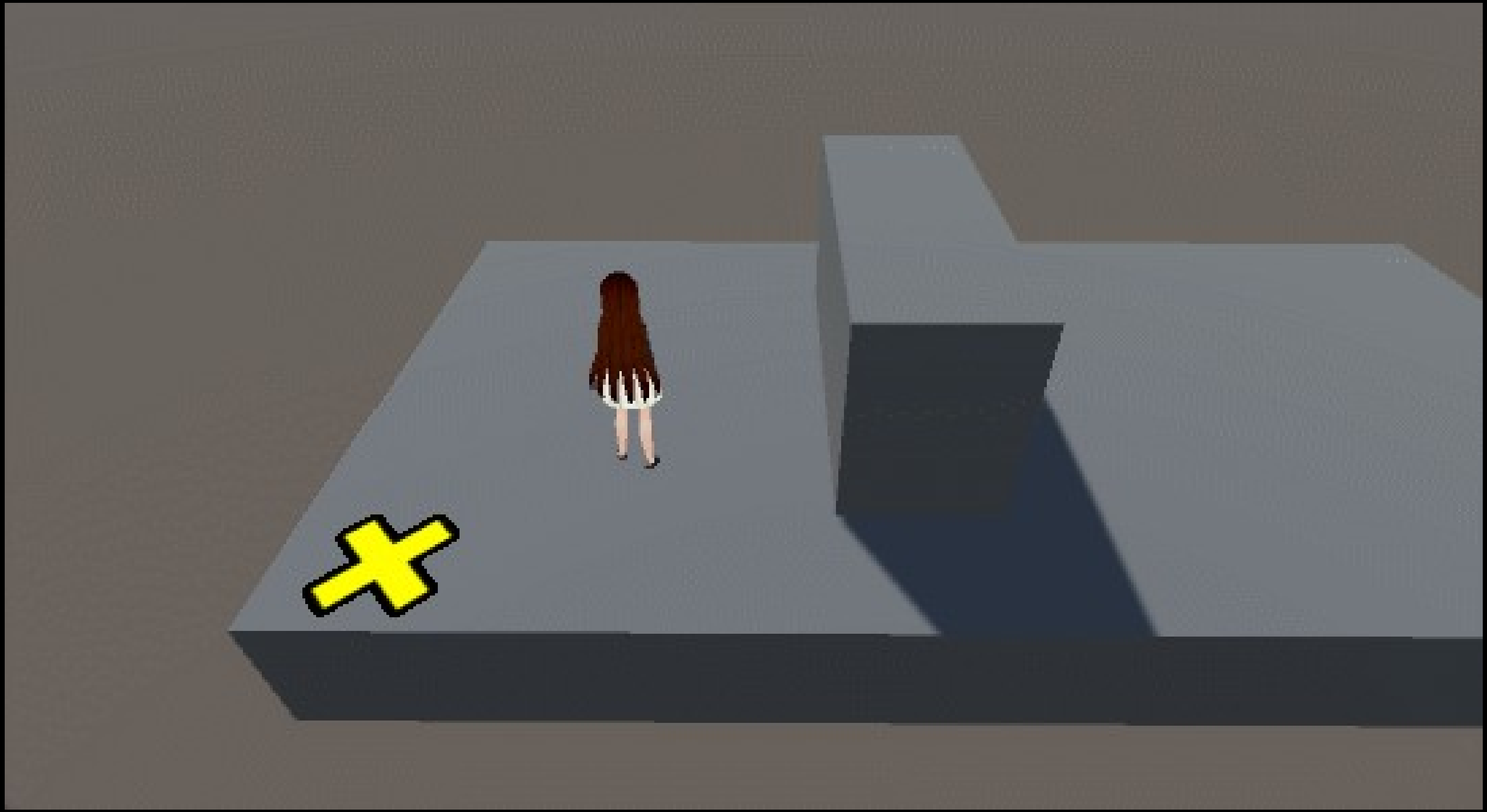


```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float velocity = 3.0f;
    public GameObject model;

    private Vector3 movingVec;
    private Animator anim;
    private Rigidbody rigid;

    void Awake()
    {
        anim = model.GetComponent<Animator>();
        rigid = GetComponent<Rigidbody>();
    }

    void Update()
    {
        // Using Transform to Translate.
        float movingVecH = Vector3.Dot(movingVec, Vector3.right);
        float movingVecV = Vector3.Dot(movingVec, Vector3.forward);

        if (Mathf.Abs(movingVecH) >= Mathf.Abs(movingVecV))
        {
            movingVec = movingVecH * Vector3.right;
        }
        else
        {
            movingVec = movingVecV * Vector3.forward;
        }
```



```

        if (movingVec.magnitude > 0.1f)
        {
            model.transform.forward = Vector3.Slerp(model.transform.forward, movingVec, 0.05f);
            anim.Play("walking");
        }
        else
        {
            anim.Play("idle");
        }
        // We no longer us transform to update player position.
        //transform.Translate(movingVec * velocity * Time.deltaTime, Space.World);
        // On the other hand, we are going to use rigidbody to drive the player.
        rigid.velocity = movingVec * velocity;
    }

    public void Move(Vector3 vector)
    {
        movingVec = vector;
    }
}
```

- Try it.



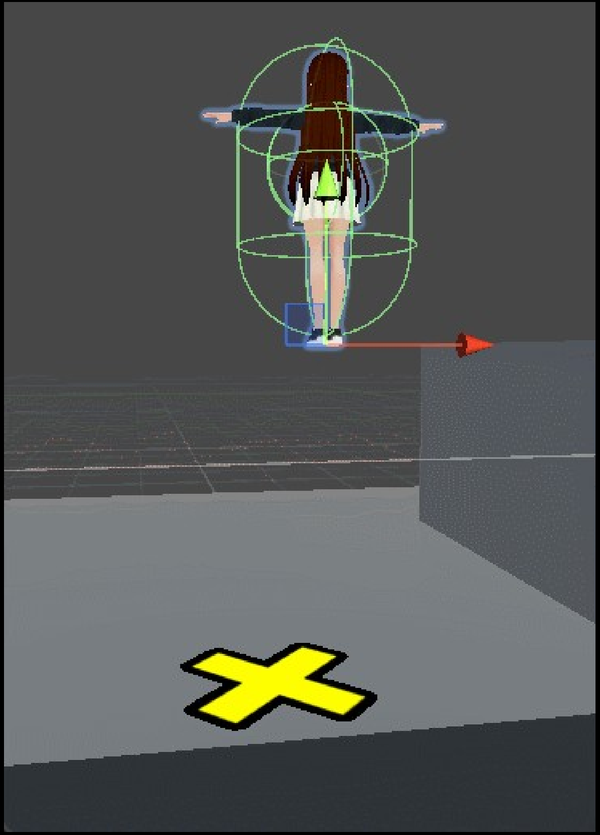
互動程式設計III

Interactive Programming Design Integration

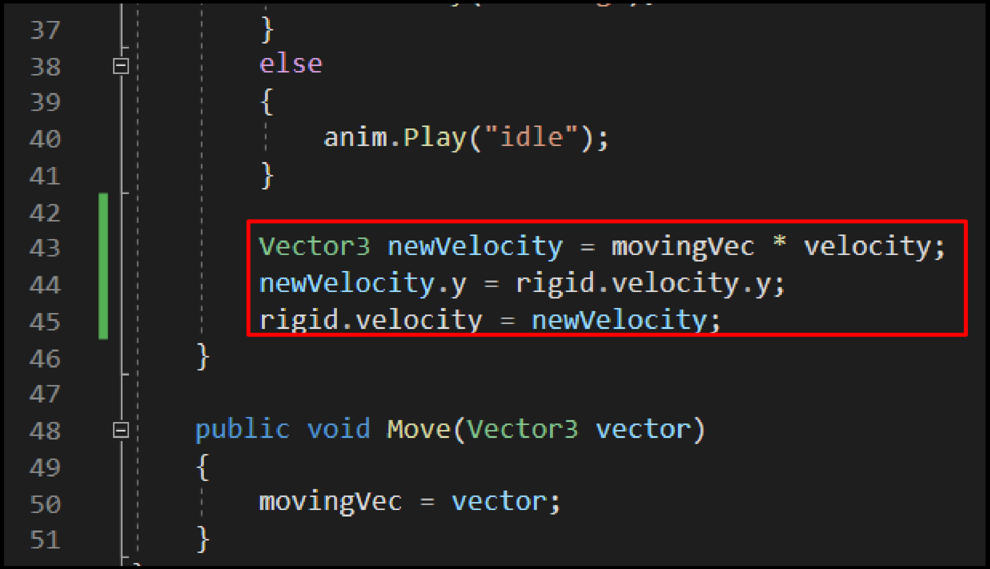
- Player Control
  - Player Structure
  - Planar Move
  - Fixed Camera
  - Block
  - Fall
  - Jump
  - Friction

EX: Falling with Gravity

- Right now the player doesn't fall correctly. It seems to fall with constant velocity. The gravity doesn't apply on player.
  - This is because the y component of movingVec is zeroed out every tick. So it accelerate from 0 over and over in every tick.
  - Everytime when we update rigid.velocity, we should keep the current rigid.velocity.y unchanged. The physics engine will update the y component according to gravity for us.

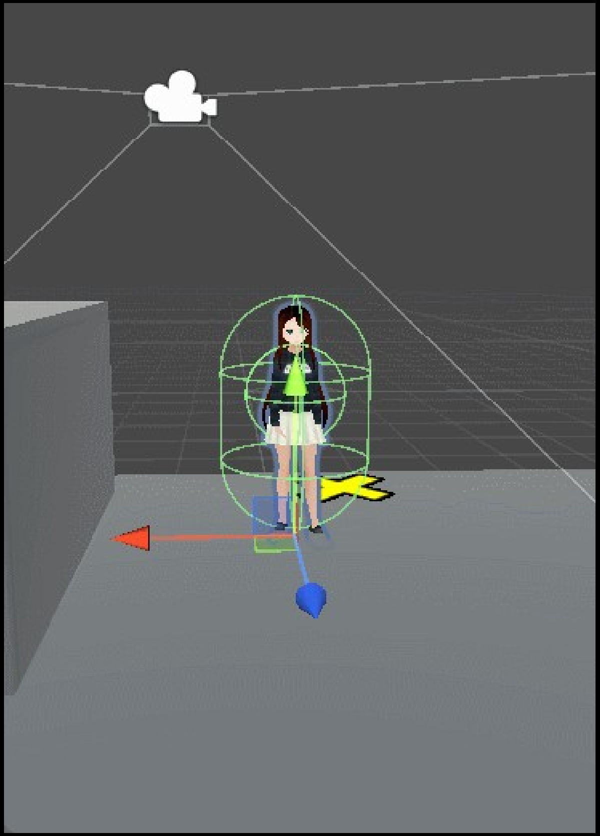


- Save the target velocity to a buffer Vector3. Overwrite the y component to the current rigid.velocity.y. And then store the Vectore3 back to the rigid.



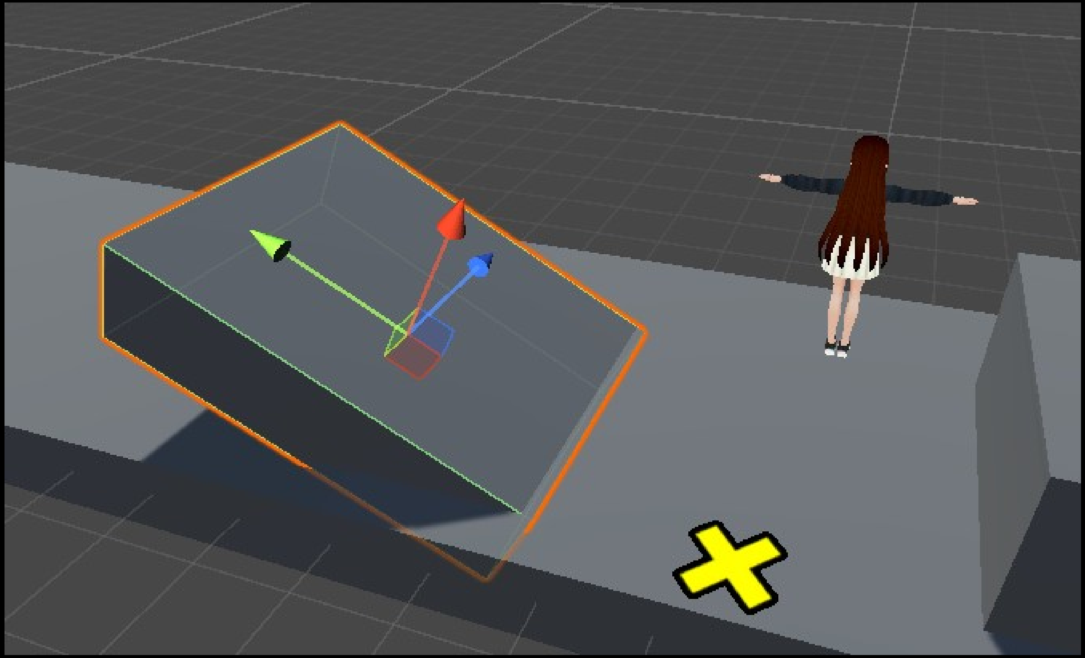
```
    }
    else
    {
        anim.Play("idle");
    }

    Vector3 newVelocity = movingVec * velocity;
    newVelocity.y = rigid.velocity.y;
    rigid.velocity = newVelocity;
}

public void Move(Vector3 vector)
{
    movingVec = vector;
}
```

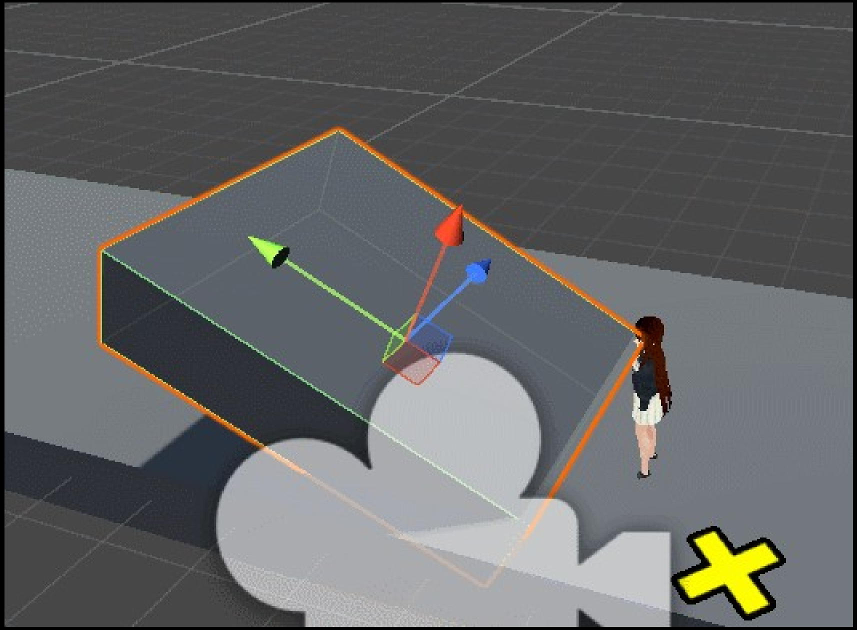
- Try it. Drag the player handle up and watch it falling.



EX: Walking On Slope

- Place and rotate a cube and make it a slope. Set its tilt angle to about 30 degree.
- The player should be able to walk on the slope in all directions.





互動程式設計III

Interactive Programming Design Integration

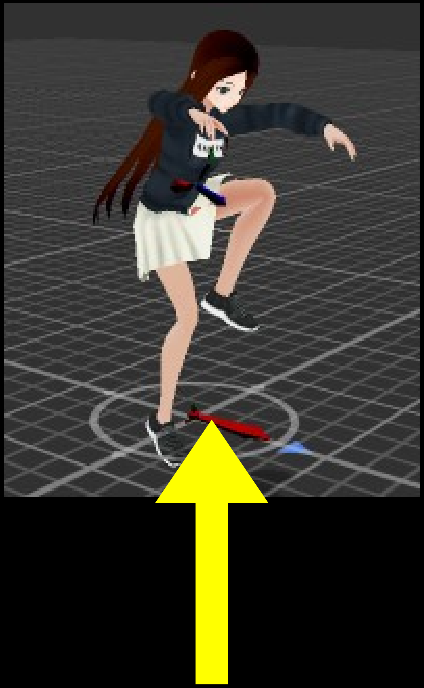
- Player Control
  - Player Structure
  - Planar Move
  - Fixed Camera
  - Block
  - Fall
  - Jump
  - Friction

Jump Thrust

- In order to jump, we need to apply a vertical thrust on the player.
  - In the context of a rigid body (like an aircraft, spacecraft, or game object), thrust is the force that accelerates the object in the direction of motion. The thrust equation for a rigid body depends on Newton’s second law of motion and can be expressed as:

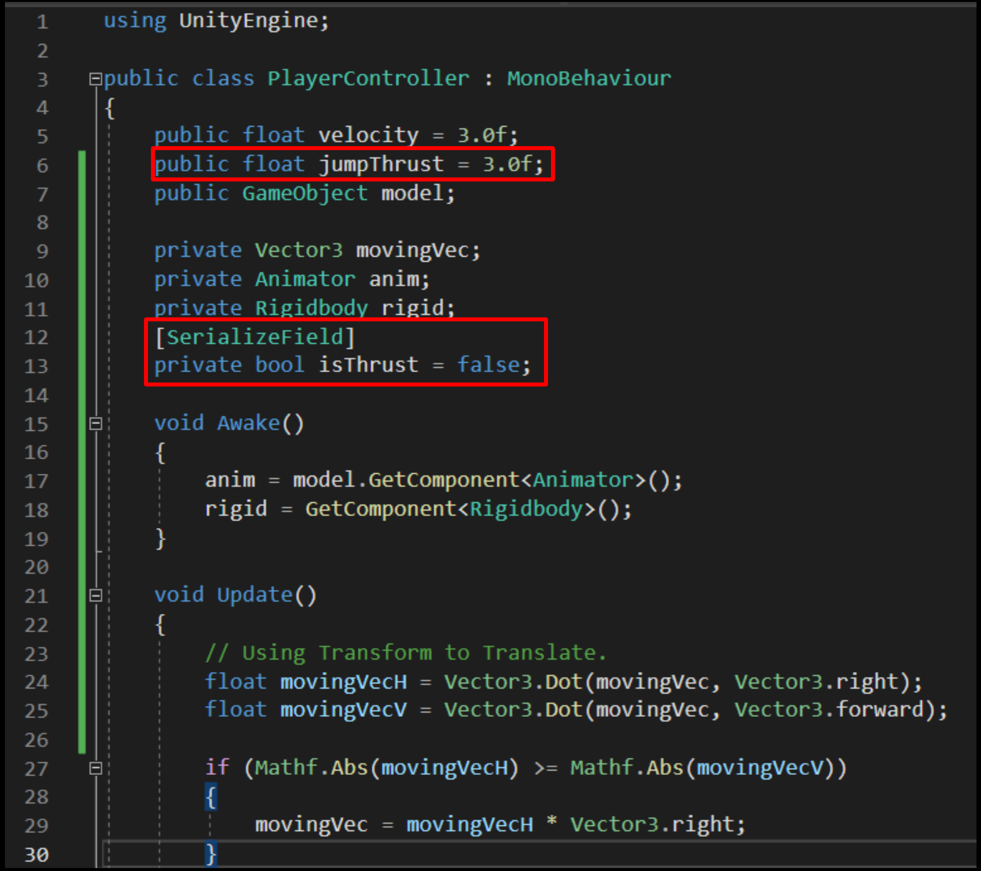
$$F_{thrust} = m \cdot a_{thrust}$$

  - F_{thrust} = Thrust force (in Newtons)
  - m = Mass of the rigid body (in kilograms)
  - a_{thrust} = Acceleration produced by the thrust (in m/s^2)
- In game design, we often directly apply a vector V_{thrust} to represent the velocity derived from jump thrust.

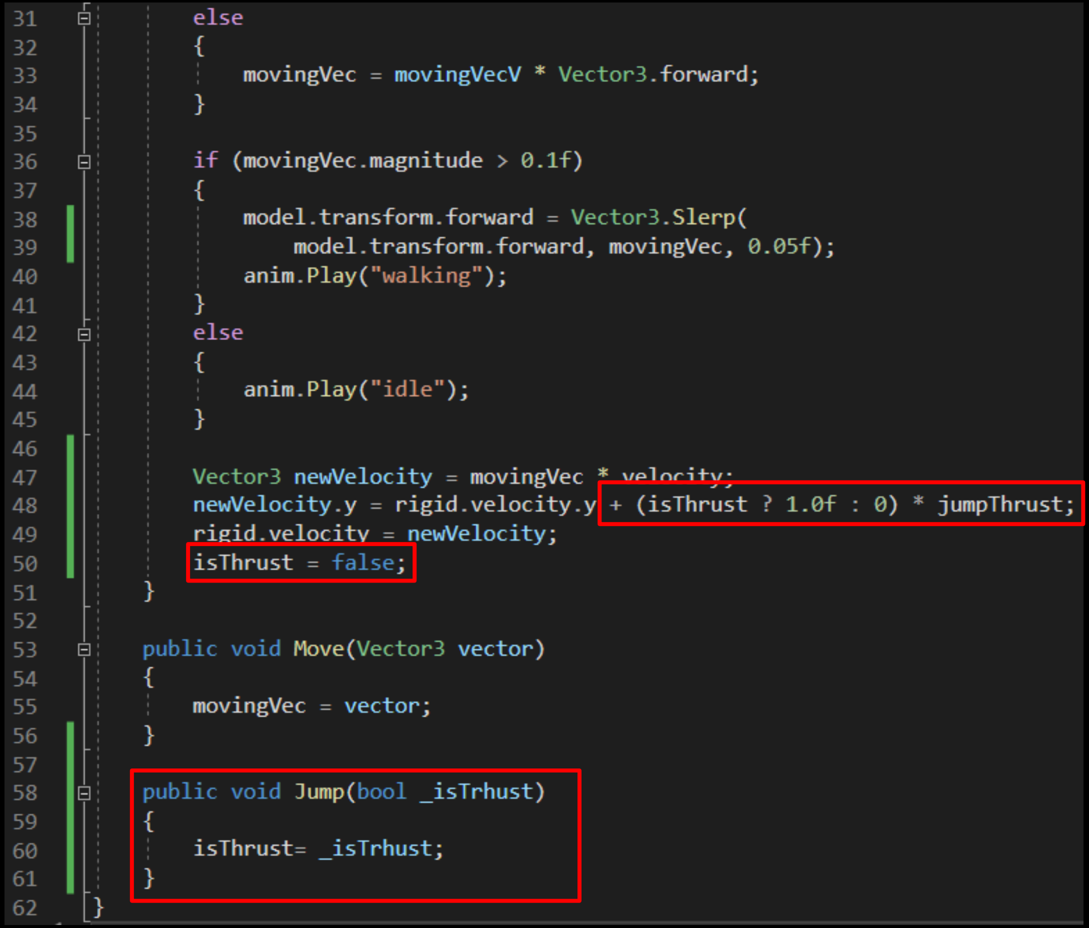


EX: Jump

-

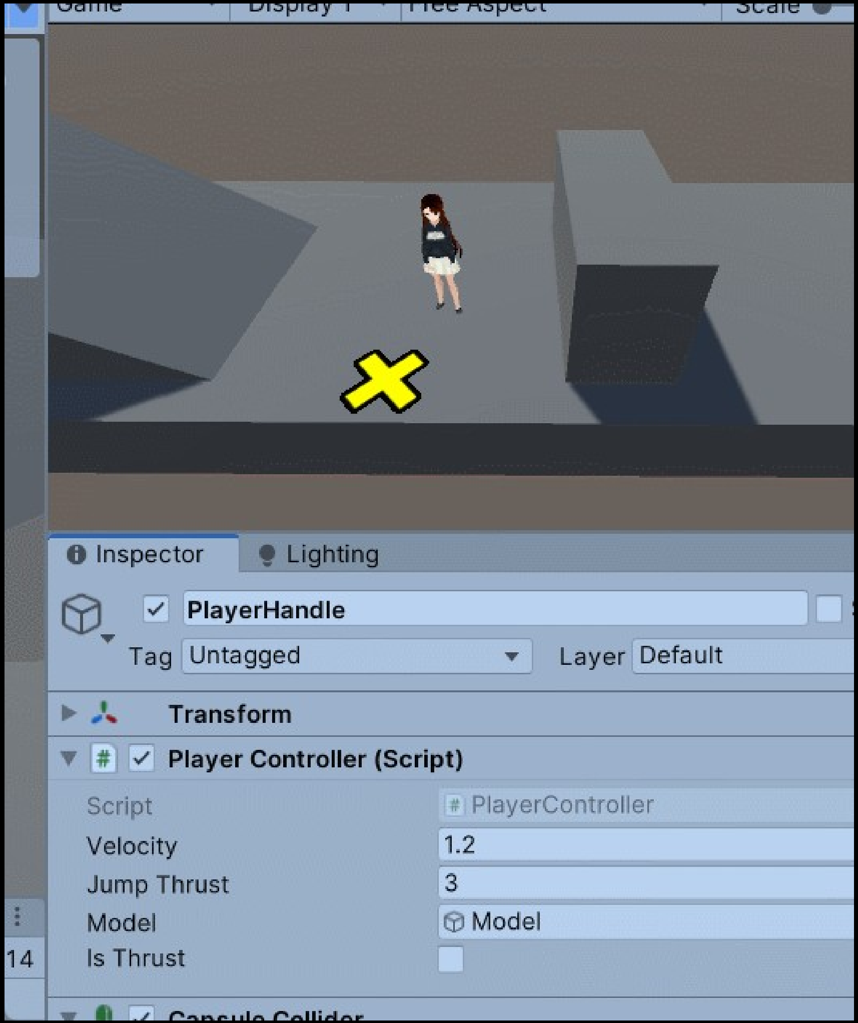


```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float velocity = 3.0f;
    public float jumpThrust = 3.0f;
    public GameObject model;

    private Vector3 movingVec;
    private Animator anim;
    private Rigidbody rigid;
    [SerializeField]
    private bool isThrust = false;

    void Awake()
    {
        anim = model.GetComponent<Animator>();
        rigid = GetComponent<Rigidbody>();
    }

    void Update()
    {
        // Using Transform to Translate.
        float movingVecH = Vector3.Dot(movingVec, Vector3.right);
        float movingVecV = Vector3.Dot(movingVec, Vector3.forward);

        if (Mathf.Abs(movingVecH) >= Mathf.Abs(movingVecV))
        {
            movingVec = movingVecH * Vector3.right;
        }
```



```
        else
        {
            movingVec = movingVecV * Vector3.forward;
        }

        if (movingVec.magnitude > 0.1f)
        {
            model.transform.forward = Vector3.Slerp(
                model.transform.forward, movingVec, 0.05f);
            anim.Play("walking");
        }
        else
        {
            anim.Play("idle");
        }

        Vector3 newVelocity = movingVec * velocity;
        newVelocity.y = rigid.velocity.y + (isThrust ? 1.0f : 0) * jumpThrust;
        rigid.velocity = newVelocity;
        isThrust = false;
    }

    public void Move(Vector3 vector)
    {
        movingVec = vector;
    }

    public void Jump(bool _isTrhust)
    {
        isThrust= _isTrhust;
    }
}
```

- Try it.



PlayerHandle

Tag Untagged

Layer Default

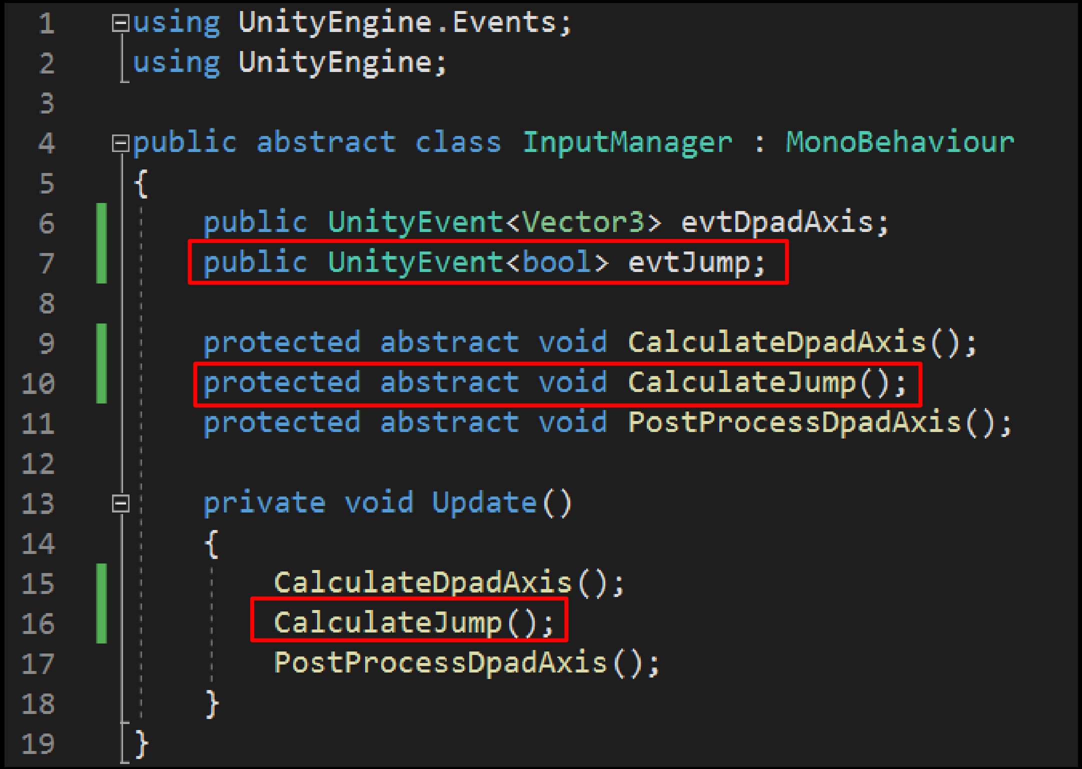
Transform

Player Controller (Script)

Script	PlayerController
Velocity	1.2
Jump Thrust	3
Model	Model
Is Thrust	□

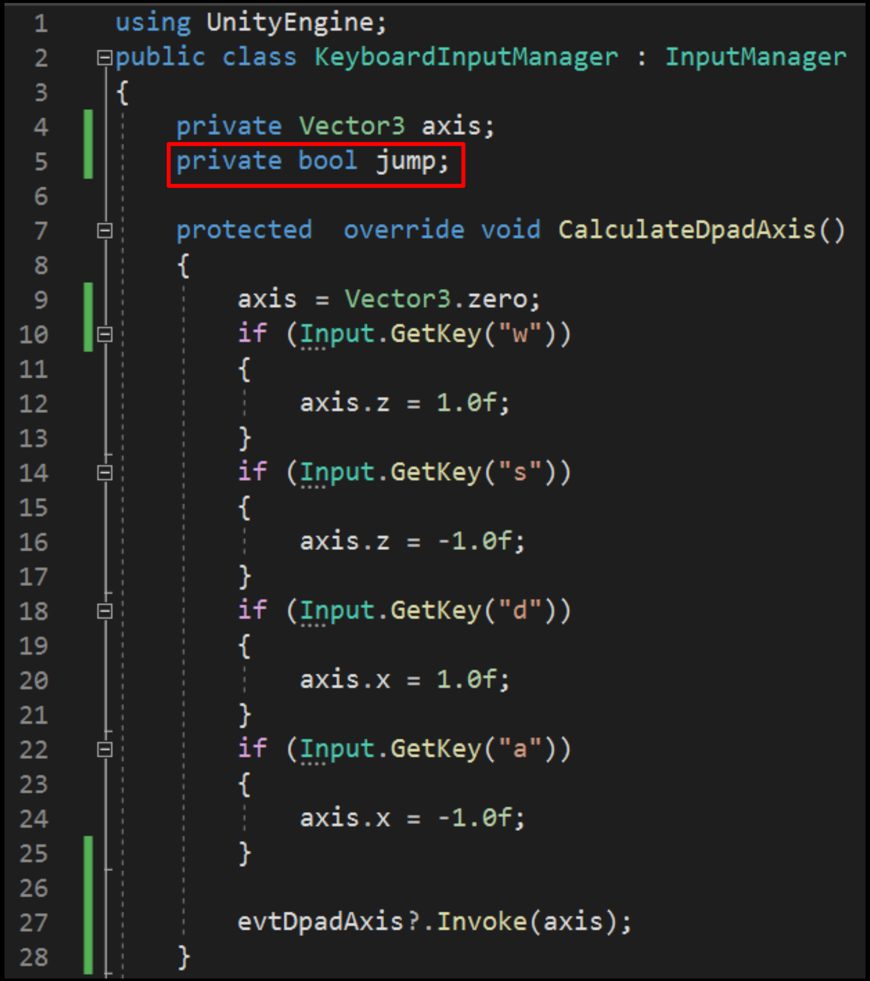
EX: Press Space to Jump

- Refactor the ImputManager to support a boolean type action event ”jump”.

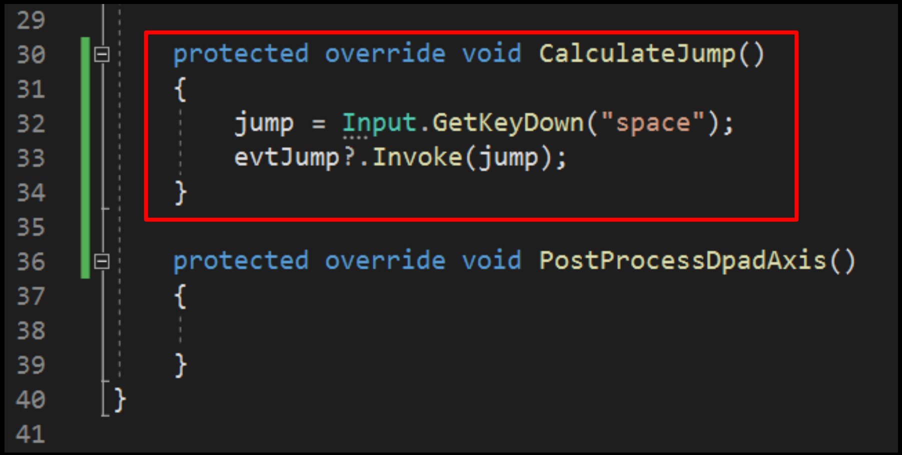


```
using UnityEngine.Events;
using UnityEngine;

public abstract class InputManager : MonoBehaviour
{
    public UnityEvent<Vector3> evtDpadAxis;
    public UnityEvent<bool> evtJump;

    protected abstract void CalculateDpadAxis();
    protected abstract void CalculateJump();
    protected abstract void PostProcessDpadAxis();

    private void Update()
    {
        CalculateDpadAxis();
        CalculateJump();
        PostProcessDpadAxis();
    }
}
```

•

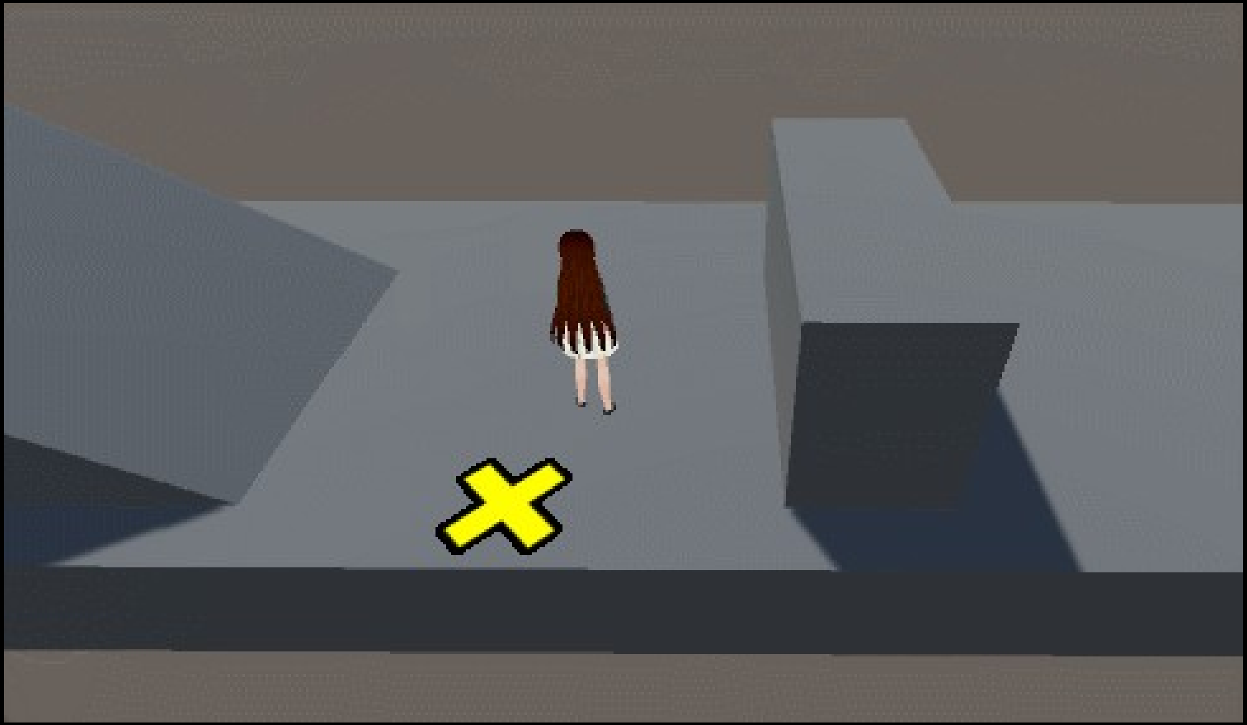


```csharp
using UnityEngine;
public class KeyboardInputManager : InputManager
{
    private Vector3 axis;
    private bool jump;

    protected  override void CalculateDpadAxis()
    {
        axis = Vector3.zero;
        if (Input.GetKey("w"))
        {
            axis.z = 1.0f;
        }
        if (Input.GetKey("s"))
        {
            axis.z = -1.0f;
        }
        if (Input.GetKey("d"))
        {
            axis.x = 1.0f;
        }
        if (Input.GetKey("a"))
        {
            axis.x = -1.0f;
        }

        evtDpadAxis?.Invoke(axis);
    }
```



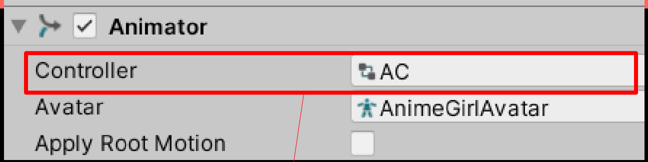
```csharp

    protected override void CalculateJump()
    {
        jump = Input.GetKeyDown("space");
        evtJump?.Invoke(jump);
    }

    protected override void PostProcessDpadAxis()
    {

    }
}

```

- Try it.
  - When we press space for several times in a short period, the player get thrusted like a rocket and fly into the sky.
  - We will fix this bug by using FSM in the future.



EX: Simple Jump Animation

- Find and drag the Jump animation into the animator AC.
  - Name it ’jump’

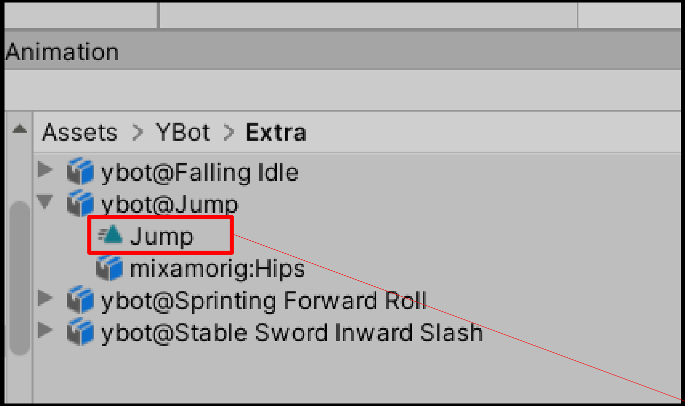


☑ Animator

Controller: AC

Avatar: AnimeGirlAvatar

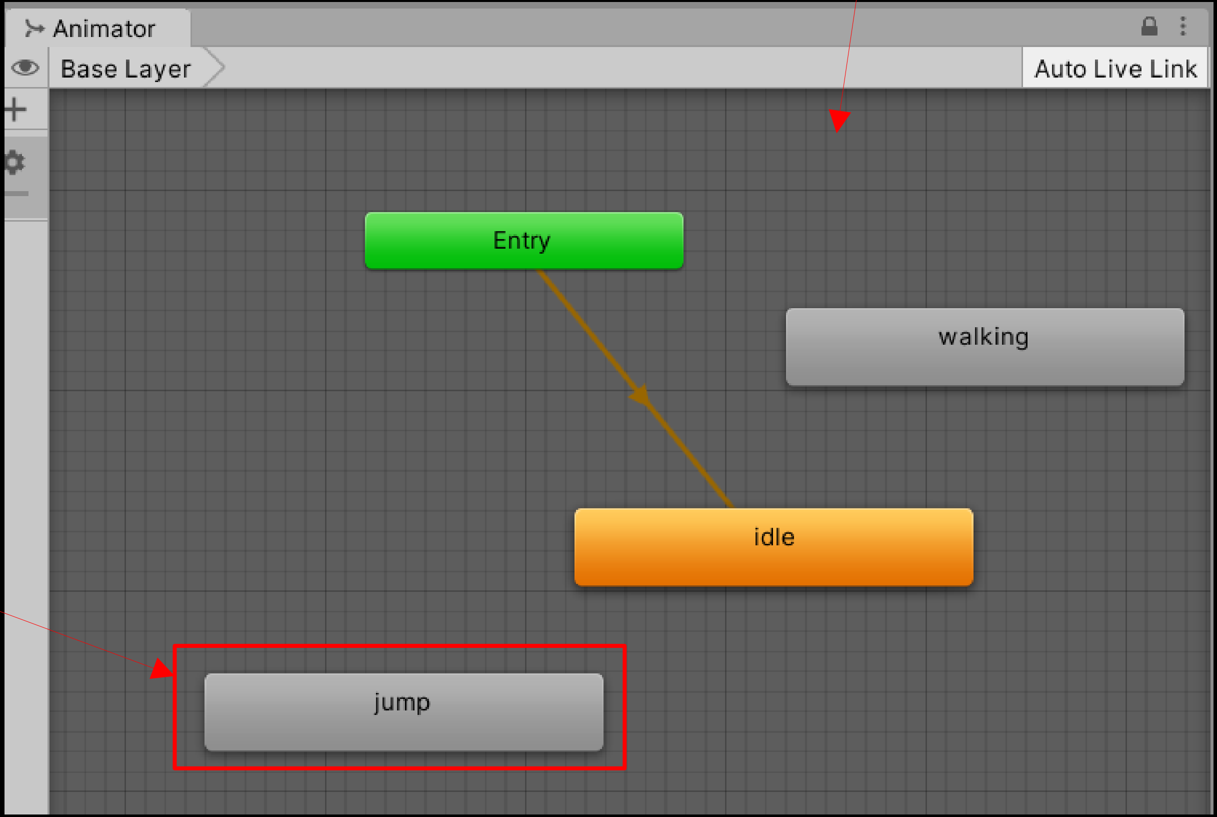
Apply Root Motion: □



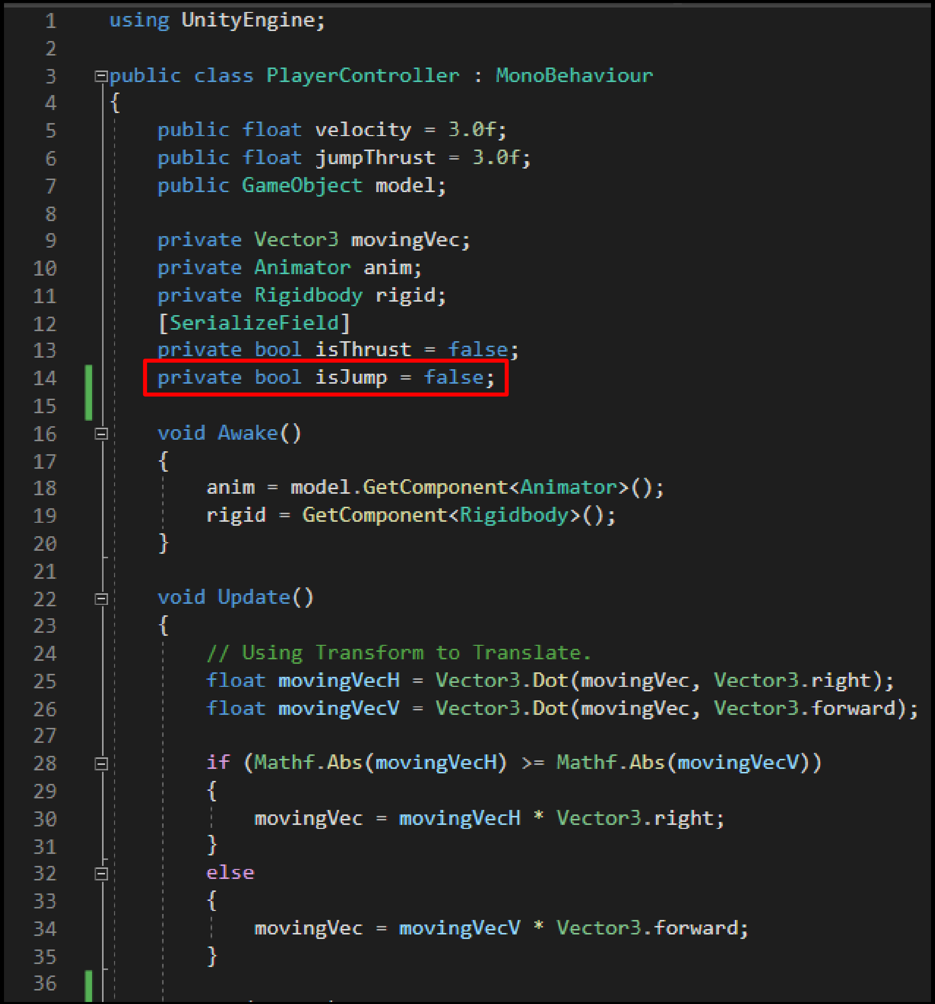
Animation

Assets > YBot > Extra

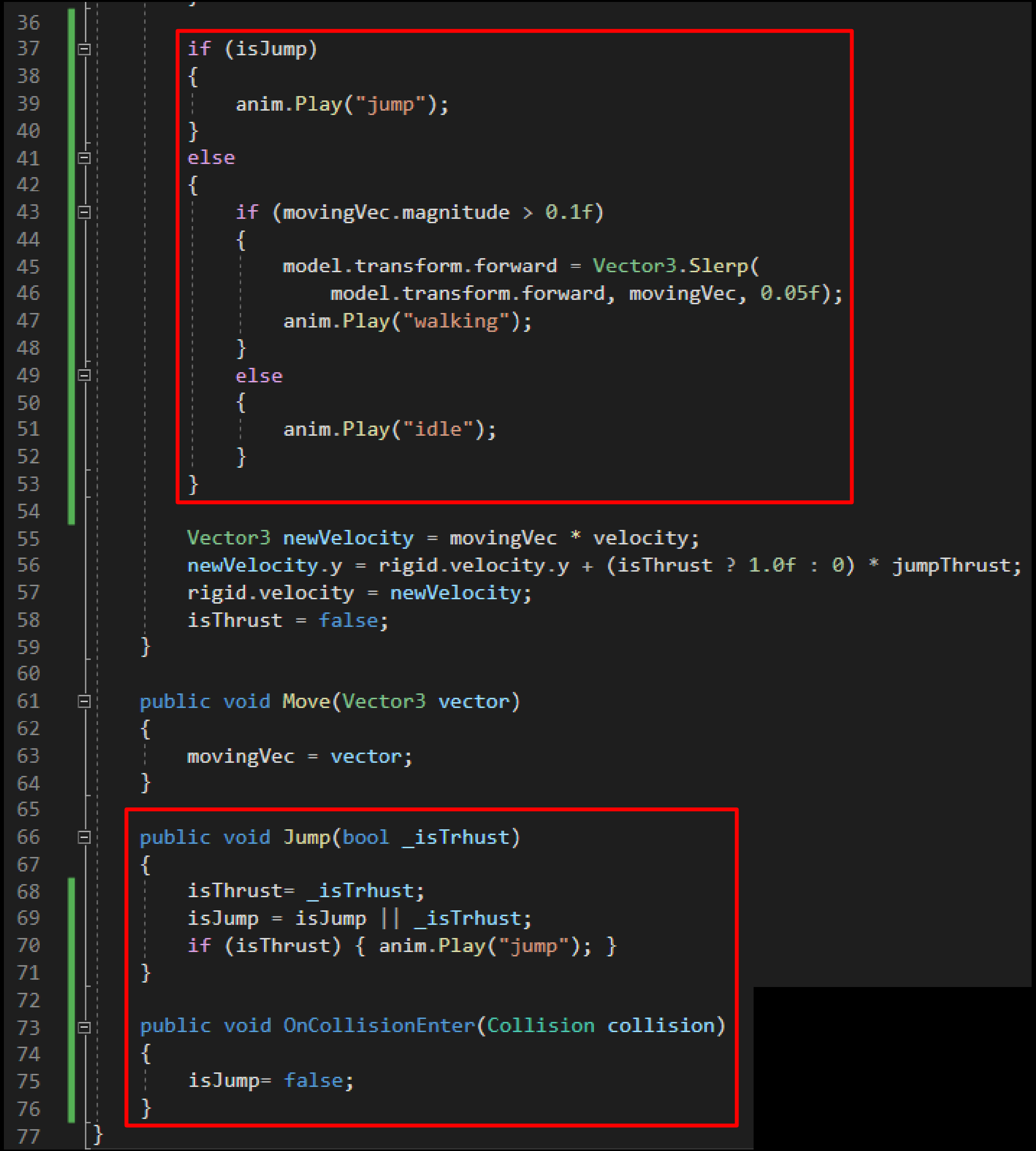
- ybot@Falling Idle
- ybot@Jump
  - Jump
  - mixamorig:Hips
- ybot@Sprinting Forward Roll
- ybot@Stable Sword Inward Slash



- Modify the player controller:

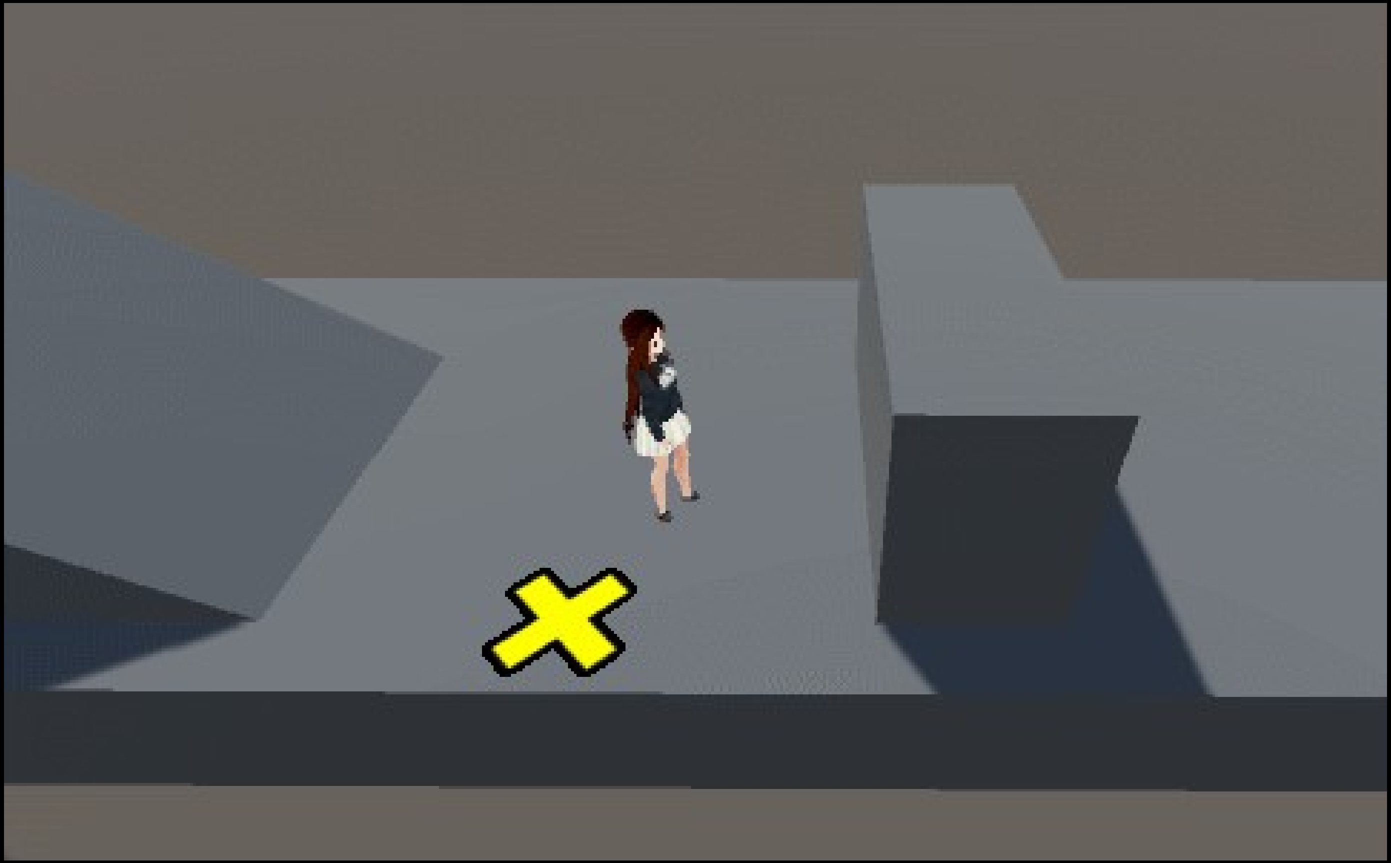


```
using UnityEngine;

public class PlayerController : MonoBehaviour
{
    public float velocity = 3.0f;
    public float jumpThrust = 3.0f;
    public GameObject model;

    private Vector3 movingVec;
    private Animator anim;
    private Rigidbody rigid;
    [SerializeField]
    private bool isThrust = false;
    private bool isJump = false;

    void Awake()
    {
        anim = model.GetComponent<Animator>();
        rigid = GetComponent<Rigidbody>();
    }

    void Update()
    {
        // Using Transform to Translate.
        float movingVecH = Vector3.Dot(movingVec, Vector3.right);
        float movingVecV = Vector3.Dot(movingVec, Vector3.forward);

        if (Mathf.Abs(movingVecH) >= Mathf.Abs(movingVecV))
        {
            movingVec = movingVecH * Vector3.right;
        }
        else
        {
            movingVec = movingVecV * Vector3.forward;
        }
```



```
        if (isJump)
        {
            anim.Play("jump");
        }
        else
        {
            if (movingVec.magnitude > 0.1f)
            {
                model.transform.forward = Vector3.Slerp(
                    model.transform.forward, movingVec, 0.05f);
                anim.Play("walking");
            }
            else
            {
                anim.Play("idle");
            }
        }

        Vector3 newVelocity = movingVec * velocity;
        newVelocity.y = rigid.velocity.y + (isThrust ? 1.0f : 0) * jumpThrust;
        rigid.velocity = newVelocity;
        isThrust = false;
    }

    public void Move(Vector3 vector)
    {
        movingVec = vector;
    }

    public void Jump(bool _isTrhust)
    {
        isThrust= _isTrhust;
        isJump = isJump || _isTrhust;
        if (isThrust) { anim.Play("jump"); }
    }

    public void OnCollisionEnter(Collision collision)
    {
        isJump= false;
    }
}
```

- Try it.
  - The player plays jump animation while flying in the air.
  - But once the player hits the wall, the jump animation ends. We can solve this problem by a detailed FSM.



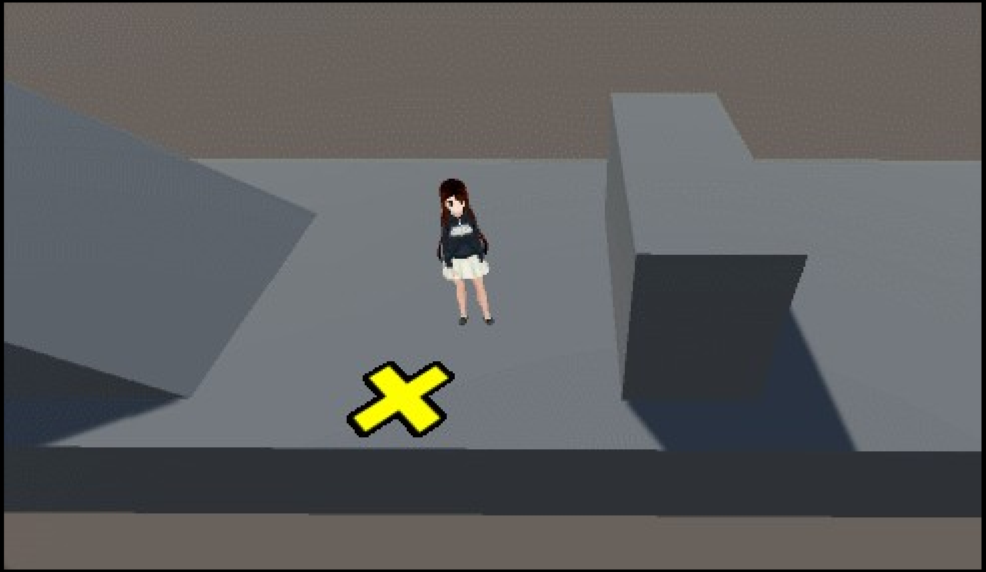
互動程式設計III

Interactive Programming Design Integration

- Player Control
  - Player Structure
  - Planar Move
  - Fixed Camera
  - Block
  - Fall
  - Jump
  - Friction

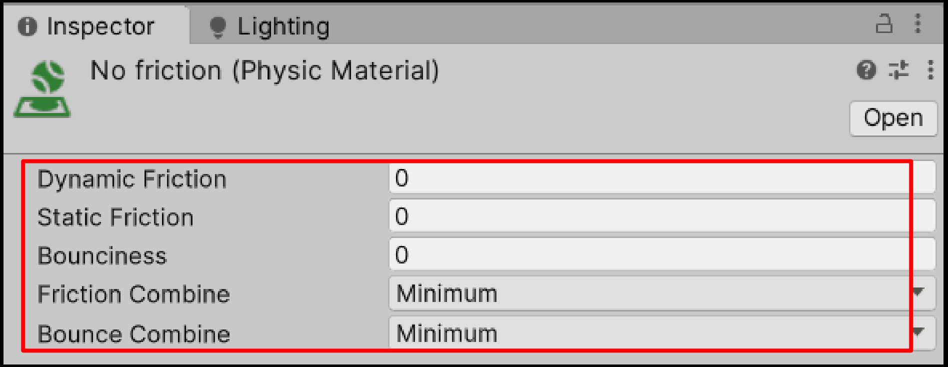
Getting Stuck on Walls

- When the player jumps onto a wall, they may become stuck or stick to the surface. This occurs due to friction between the player handle (collider) and the wall’ s surface.
- Understanding the Problem:
  - High Friction between colliding surfaces causes the player to lose momentum, preventing smooth movement or sliding off.
  - Physics Materials used in both the player’s collider and the wall can unintentionally increase friction, causing this behavior.



- Reducing Friction with Physics Materials
  - Create a Physics Material, name it <No friction>
    - Right-click in the Project window → Create → Physics Material.
    - Set both of the friction to 0.
    - Set both of the combine policies to Minimum. ❶
  - Assign the material to the capsule collider of the wall cube. ❷

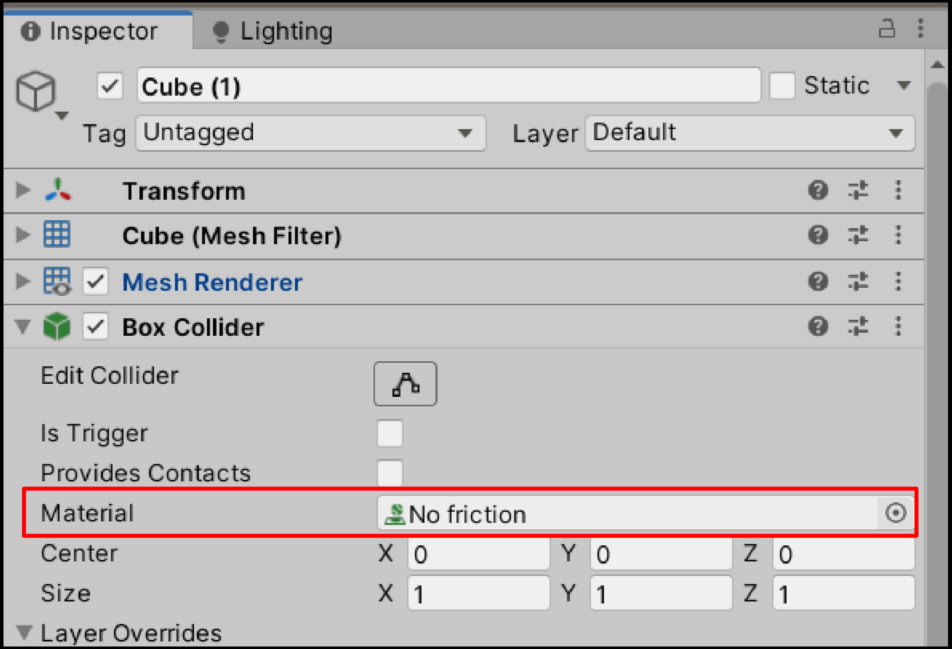
❶



No friction (Physic Material)

Dynamic Friction 0
Static Friction 0
Bounciness 0
Friction Combine Minimum
Bounce Combine Minimum

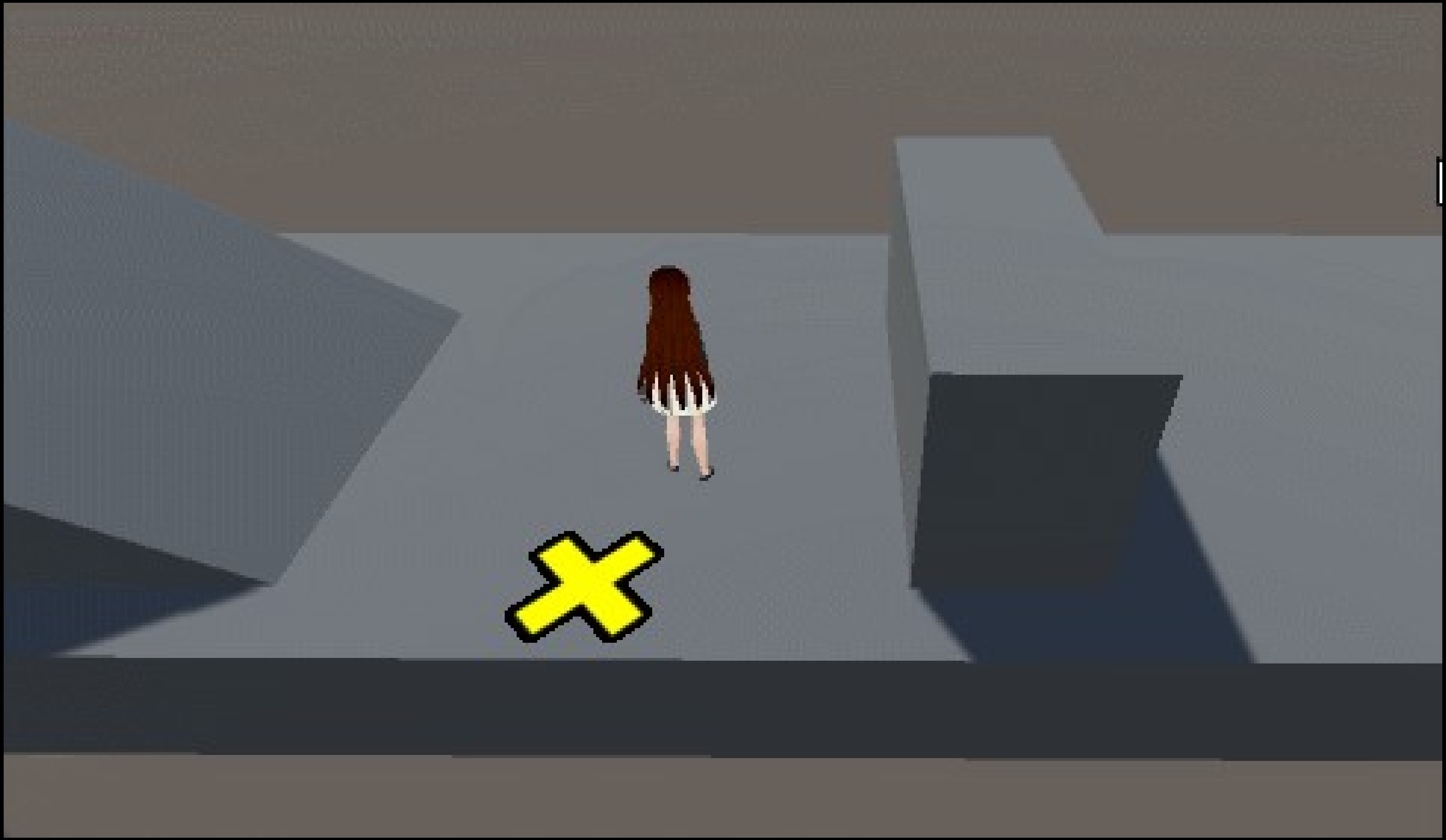
❷



☑ Cube (1) □ Static
Tag Untagged Layer Default

☑ Box Collider
Is Trigger □
Provides Contacts □
Material No friction
Center X 0 Y 0 Z 0
Size X 1 Y 1 Z 1
Layer Overrides

- Try it.



Further Imporvements

- Lock horizontal movement while jumping.
- Add more states: run, fall, hit, dead, roll, defense states.
  - We need FSM to avoid character twitching!
  - But, the animator is also an FSM. That means we have to deal the synchronization problem beteen our logic FSM and animator FSM…
    - Or, we can drop the animator FSM and make it transition-less like what we have done in previous exercises in this chapter.
      - We just call Animator.Play() whenever we need to play an animation.
- The more character states you design, the more robust the character controller should be!
  - Please aware that we did not include any FSM character controller in this chapter yet.
- Character controller plays the most important role in an 1st/3rd person game design.
  - You probably will spend weeks to month designing your character controller in a serious project.

互動程式設計III

Interactive Programming Design Integration

Q&A