Project Settings

At first when opening the Unity editor I was a bit overwhelmed by the many options available, and it can be hard to get going without at least knowing how to configure the most basic of settings for a Unity project. In the sections below, I'll cover some simple settings that are worthwhile to consider when creating a new project in Unity.

Playmode Tint

This option is not found in `Project Settings`, but I think it is something everyone entering into Unity for the first time should consider. Navigate to the menu bar at the top of your editor and select `Edit->Preferences->Colors` and adjust the `Playmode Tint` to something very noticable. This will avoid forgetting you are in play mode and making some changes, only to be forced to revert them all once exiting play mode.

For the rest of these sections, we will be working in the `Project Settings` panel opened with `Edit->Project Settings...` in the menu bar of the Unity editor.

[Official Unity Project Settings Documentation](#)

Project Name

It is not to be assumed that Unity will distribute builds of your game with your local Unity project name as you defined it when creating your project initially. In fact, Unity requires us to specify these details within the `Player` section of `Project Settings`. See the section below is adjusted to suit the needs of your project.

Game / Application Icons

It's important to change things like this from the default settings, otherwise even a finished project can end up looking incomplete. Navigate to the `Player` section and scroll down to adjust icon

settings. It's important to be consistent across all platforms, and this can easily be done by checking the Override for PC, Max & Linux Standalone tick box at the top of the panel. This will apply your Icon settings on all platforms.
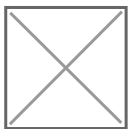

Splash Screen

This is for Unity Professional Licenses only

Within the Player panel we can find the below settings for modifying the splash screen of a game or application created with Unity.

Quality Settings

It's important to adjust quality settings to suit your development environment so you aren't running your game within the Unity editor in max settings.

You can rename quality levels, add new, and adjust platform-specific modes as well. It's important to note that clicking the name of the quality setting in this table (just left of the check-marks) will apply the settings within your editor for testing. The Default drop-down arrors correspond with each platform at the top level of the table.

Graphics Settings

This is where you'll define the preconfigured graphics settings available to the player. Its important to adjust these to suit the platform the build will be running on. As an example, this feature could be useful when trying to distribute a test build of a Unity game with WebGL. We could reduce the settings to improve performance within the browser to make the game much less demanding. This allows us to build more efficiently to WebGL and not create an unecessarily demanding or slow performing game given this basic platform of WebGL.

Curious what WebGL is or looks like in use? I host some archived examples on my website, <u>click here</u> to check out some Unity WebGL games that I've already built and hosted online for playing.

Input Manager

This section is very useful in configuring controls for your game that can then be used for scripting. For example, the section below I have defined a button for `Fire`, which is triggered when the player clicks the left mouse button

By using a custom script that defines global constants, we can reduce the task of changing these values later on. Below, I'll cover an example of using the `Input Manager` paired with a few C# scripts to define controls in global variables which can be easily modified in a central location. This avoids a scenario where we have built a complex game and want to change controls later in development, requiring us to change static values across numerous scripts. This is not only tedious but also makes the project more prone to errors.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Controls : MonoBehaviour
{
    // Constants used within the game to handle passing control settings to builtin unity functions with string parameters
    // Unity parses these strings against settings in Edit->ProjectSettings->InputSettings

    // Character walking controls for joystick / keyboard
    // Keyboard has boolean movement, joystick has variable 0.0f - 1.0f
    public const string c_MoveStrafe = "Horizontal";
    public const string c_MoveWalk = "Vertical";

    // Character look controls for mouse / joystick
    public const string c_LookMouseVertical = "Mouse Y";
    public const string c_LookMouseHorizontal = "Mouse X";
```

```csharp
    public const string c_LookGamePadVertical = "Look Y";
    public const string c_LookGamePadHorizontal = "Look X";


    // Character movement modifiers
    public const string c_ModJump              = "Jump";
    public const string c_ModSprint            = "Sprint";
    public const string c_ModCrouch             = "Crouch";


    // Character weapon controls
    public const string c_PrimaryAim               = "Aim";
    public const string c_PrimaryGamepadAim            = "Gamepad Aim";
    public const string c_PrimaryFire             = "Fire";
    public const string c_PrimaryGamepadFire           = "Gamepad Fire";
    public const string c_PrimarySwitchWeapon          = "Mouse ScrollWheel";
    public const string c_PrimaryGamepadSwitchWeapon     = "Gamepad Switch";
    public const string c_PrimaryHide            = "Primary Hide";
    public const string c_PrimaryNextWeapon             = "NextWeapon";


    // UI controls
    public const string c_UIPauseMenu            = "Pause Menu";
    public const string c_UISubmit            = "Submit";
    public const string c_UICancel            = "Cancel";
}
```

These constants can then be used in a relative `PlayerInput` class, which can handle recieving input from the player at a higher level so we won't need to refactor all of our scripts in the scenario that we want to modify our controls.

```csharp
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
// A script to pass player input to a relative GameObject's Controller script


public class PlayerInput : MonoBehaviour
{


    PlayerController playerController;


    // Start is called before the first frame update
    void Start()
```

```csharp
    {
        playerController = GetComponent<PlayerController>();
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }


    // Update is called once per frame
    void Update()
    {
        // Always check if the player wants to unbind their cursor lock state
        UpdateLockState();
    }


    public bool CanProcessInput()
    {
        // If the cursor is locked within the game, return true
        return Cursor.lockState == CursorLockMode.Locked;
    }


    public Vector3 GetMovement()
    {
        // If the cursor is not locked within the game, do nothing
        if (!CanProcessInput()) return Vector3.zero;


        // Create a file / class (Controls.cs) to hold strings which can be passed to functions in all scripts
        // Allows for easy control customization without reefactoring a lot of code
        // GetAxis returns 0.0f-1.0f strength of movement (allows joystick variable movement, keyboard movement
is 1.0f if key pressed)
        // Horizontal = a,d    Vertical = w,s
        Vector3 move;
        switch(playerController.targetPov)
        {
            case Kamera.pov.Mounted:
            move = transform.forward * Input.GetAxis(Controls.c_MoveStrafe) * -1.0f  + transform.right *
Input.GetAxis(Controls.c_MoveWalk) * 1.0f ;
                break;


            default:
            move = transform.right * Input.GetAxis(Controls.c_MoveStrafe) + transform.forward *
Input.GetAxis(Controls.c_MoveWalk);
```

```csharp
                break;
        }
        // Return the clamped amnount of movement to apply to a GameObject within some relative Controller
script
        return move;
    }


    public float GetLookHorizontal()
    {
        return GetLookAxis(Controls.c_LookMouseHorizontal, Controls.c_LookGamePadHorizontal);
    }


    public float GetLookVertical()
    {
        return GetLookAxis(Controls.c_LookMouseVertical, Controls.c_LookGamePadVertical);
    }


    // If the player presses the UICancel key, the cursor is unlocked.
    void UpdateLockState()
    {
        if (Input.GetButton(Controls.c_UICancel)) Cursor.lockState = CursorLockMode.None;
        else if (Input.GetMouseButton(0)) Cursor.lockState = CursorLockMode.Locked;
    }


    // Checks whether the look input is via mouse or gamepad and returns a float 0.0f-1.0f of the strength
    float GetLookAxis(string mouseLook, string stickLook)
    {
        if (CanProcessInput())
        {

            // Check if there is any input from a gamepad controller on the given axis
            bool isGamePad = Input.GetAxis(stickLook) != 0.0f;
            // If we are using a gamepad use stickLook's strength, otherwise use mouse input
            float str = isGamePad ? Input.GetAxis(stickLook) : Input.GetAxis(mouseLook);

            if (isGamePad)
            {
                // since mouse input is already deltaTime-dependant, only scale input with frame time if it's coming
from sticks
                str *= Time.deltaTime;
```

```
            }
            else
            {
                // reduce mouse input amount to be equivalent to stick movement
                str *= 0.01f;
    #if UNITY_WEBGL
    // Mouse tends to be even more sensitive in WebGL due to mouse acceleration, so reduce it even more
                // str *= webglLookSensitivityMultiplier;
    #endif
            }

            return str;
        }
        else return 0.0f;


    }


    public bool GetCrouchDown()
    {
        return Input.GetButtonDown(Controls.c_ModCrouch);
    }
    public bool GetCrouchUp()
    {
        return Input.GetButtonUp(Controls.c_ModCrouch);
    }

    public bool GetSprintHeld()
    {
        return Input.GetButton(Controls.c_ModSprint);
    }

    public bool GetJumpPress()
    {
        return Input.GetButtonDown(Controls.c_ModJump);
    }

    public bool GetMouseFire()
    {
        return Input.GetButtonDown(Controls.c_PrimaryFire) ||
Input.GetButtonDown(Controls.c_PrimaryGamepadFire);
```

```
    }

    public bool GetMouseAlt()
    {
        return Input.GetButtonDown(Controls.c_PrimaryAim) ||
Input.GetButtonDown(Controls.c_PrimaryGamepadAim);
    }

    public bool GetLowerPrimary()
    {
        return Input.GetButtonDown(Controls.c_PrimaryHide);
    }


    public int GetNumberPress()
    {
        if(Input.GetKeyDown(KeyCode.Alpha0)) return 9;
        else if(Input.GetKeyDown(KeyCode.Alpha1)) return 0;
        else if(Input.GetKeyDown(KeyCode.Alpha2)) return 1;
        else if(Input.GetKeyDown(KeyCode.Alpha3)) return 2;
        else if(Input.GetKeyDown(KeyCode.Alpha4)) return 3;
        else if(Input.GetKeyDown(KeyCode.Alpha5)) return 4;
        else if(Input.GetKeyDown(KeyCode.Alpha6)) return 5;
        else if(Input.GetKeyDown(KeyCode.Alpha7)) return 6;
        else if(Input.GetKeyDown(KeyCode.Alpha8)) return 7;
        else if(Input.GetKeyDown(KeyCode.Alpha9)) return 8;
        else return -1;
    }

}
```

If you want to actually be able to apply damage, we need a `Target` script. See the simple example below for a script which enables this to occur. Later, within `WeaponControl.cs`, we will check if the object we hit has this script attached, and if it does we can deal damage to the set HP amount given to the `Target`

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;


public class Target : MonoBehaviour
```

```csharp
{
    [SerializeField]
    private float health;

    [SerializeField]
    [Tooltip("If this is true we spawn the broken GameObject on destruction")]
    public bool isDestructable = false;

    [SerializeField]
    [Tooltip("The GameObject to spawn when this object is broken")]
    public GameObject broken;

    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void TakeDamage(float amount)
    {
        health -= amount;
        if (health <= 0)
        {
            if(isDestructable) Instantiate(broken, gameObject.transform, false);
            Destroy(gameObject);
        }
    }

}
```

We could then use this `PlayerInput.cs` script within a `WeaponControl.cs` script check if the player presses this button in `Update` called once per frame. If they have tried to shoot and have a weapon equipped, we can call the relative weapon's `ShootWeapon()` function. Note that you will have to attach the `playerInput` and `playerWeapons` variables to relative scripts within your editor.

```csharp
public class WeaponControl : MonoBehaviour
{
    [SerializeField]
    PlayerInput playerInput;
    [SerializeField]
    PlayerWeapons playerWeapons;
    [SerializeField]
    public Transform muzzle;
    [SerializeField]
    public GameObject projectile;
    [SerializeField]
    Camera mainCamera;
    [SerializeField]
    ParticleSystem muzzleFlash;
    [SerializeField]
    GameObject hitEffect;
    private float range = 500.0f;
    private float damage = 10.0f;


    void Update()
    {
        if (playerInput.GetMouseFire() && playerWeapons.hasWeapon) ShootWeapon();
    }

    void ShootWeapon()
    {
        fireSFX.Play();
        GameObject projectileObj = Instantiate(projectile, muzzle.transform.position +
mainCamera.transform.forward, mainCamera.transform.rotation);
        projectileObj.GetComponent<Rigidbody>().AddForce(transform.forward * 100);


        RaycastHit hit;
        if (Physics.Raycast(muzzle.transform.position, mainCamera.transform.forward, out hit, range))
        {
            Target temp = hit.transform.GetComponent<Target>();
            if (temp != null) temp.TakeDamage(damage);
```

```
            Debug.Log(hit.transform.name);

            muzzleFlash.Play();

            GameObject hitObject = Instantiate(hitEffect, hit.point, Quaternion.LookRotation(hit.normal));

            Destroy(hitObject, 1f);


            if (hit.rigidbody != null) hit.rigidbody.AddForce(-hit.normal * hitForce);


        }
    }


}
```