Contents

1.Assets Notice

This asset contains assets from the open-source project freedoom, which is public domain. This asset is designed so that you can easily switch out the graphics used.

Getting started:

When you import the asset for the first time, you may notice your entities have no graphics! Due to unity's request, the user is intended to find these graphics themselves, though everything can be done with freedoom and a program called Slade. All graphics must be in a folder called "Resources" in order for the code to fetch them correctly. If you are having troubles or questions about this process, email me at derrickcodewrassler@gmail.com. I try to check my email whenever I can, but I may take a bit to get back to you.

2.Sprite System Info

A. Animating sprites

There are a variety of ways to do this, I encourage you to find your own, but I animated them through a numbering variable, which set the sprite index of the script. I'm sure there are much more efficient ways, but this seemed like the easiest to do for now.

B. Getting the sprite shadow material working

If you are applying this on a new sprite renderer, it won't work until you go
into debug mode, the three dots at the top of the inspector, and enable it. From there, on the sprite renderer component, enable "cast shadows" and "receive shadows."

C. Importing your own sprites

Make sure any sprites you import go into the resources folder. If you want to organize your sprites
by folder, create folders for them in the resources folder, and make sure you include the folder in the sprite name variable.

D. Parenting rule

Make sure that any object with the sprite system component has a parent that dictates it's motion and or orientation. Otherwise, the script will not work!! If you want to add movement to a gameobject, do it through the parent, not the sprite itself. The sprite will always be facing the camera.

3.Monster Info

(It is heavily recommended that you start off using another monster prefab as a base before creating your new monsters from scratch.)

Script variables, Advanced AI:

Sound Settings

- `AudioClip[] sightSounds`: Sounds played when the AI spots a target.
- `AudioClip[] idleSounds`: Sounds played while the AI is idle.
- `AudioClip[] attackSounds`: Sounds played during an attack.
- `AudioClip[] painSounds`: Sounds played when AI takes damage.
- `AudioClip[] deathSounds`: Sounds played when AI dies.

Behavior Settings

- `float alertRange`: Defines how far AI can detect and alert allies. **(Keep this low for indoor areas, larger for outdoor areas.)**
- `bool hasRangedAttack`: Determines if AI can perform ranged attacks. **(Raycast or projectile attack is defined via animation.)**
- `bool hasMelee`: Determines if AI can perform melee attacks.
- `float raycastAccuracy`: Accuracy level for raycast-based attacks. **(0-10, includes melee attack.)**
- `int raycastDamage`: Damage dealt by raycast attacks. **(Includes melee attack.)**
- `bool rapidFire`: Defines if AI continuously fires.
- `bool canReviveOtherAI`: Defines if AI can revive teammates.
- `bool flying`: Determines if AI has flight capabilities.
- `GameObject bossHealthBar`: Health bar to enable when the monster sees player
- `float missileChance`: Probability of launching a missile attack. **(0-10.)**
- `GameObject mainProjectile`: Prefab for AI's primary projectile attack. **(If you want the monster to have multiple types of fireballs, there is a custom fireball attack in the animation event.)**

Targeting & Movement Settings

- `string enemyTag`: Defines the tag used to identify enemies.
- `Transform target`: Current target AI is pursuing.
- `bool targetSighted`: Determines if AI has spotted a target.
- `bool stoppedWalking`: Indicates if AI has stopped moving.
- `bool stillFiring`: Tracks if AI is continuously firing.

Internal Components

- `NavMeshAgent nav`: Handles AI navigation.
- `Animator anim`: Manages AI animations.
- `AudioSource audio`: Plays AI-related sounds.
- `float lostTargetTime`: Tracks time since AI lost its target.
- `float fireRate`: Defines delay between attacks.
- `float nextFireTime`: Stores the next allowed fire time.
- `bool charging`: Says if monster is doing a charge attack

Script variables, Health

- `int health`: Current health value of the entity.
- `int maxHealth`: Maximum health the entity can have.
- `bool dead`: Tracks if the entity is dead.

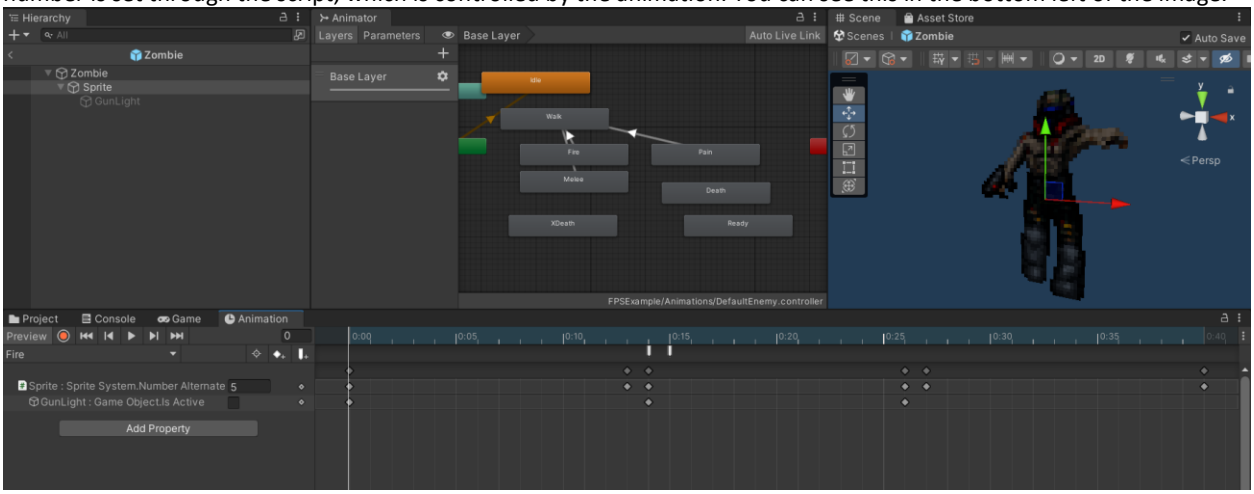
- `float painChance`: Chance that the entity will play a pain animation when taking damage.
- `GameObject dropItem`: Item dropped upon death.
- `Slider bossHealthBar`: UI element to display boss health.

Things to note:

- A monster won't infight or target another until it plays a pain animation.
- If health script collides with tags `"DamagingFloor"` or `"Explosion"` it immediately takes damage. DamagingFloors by default instant kill most things.

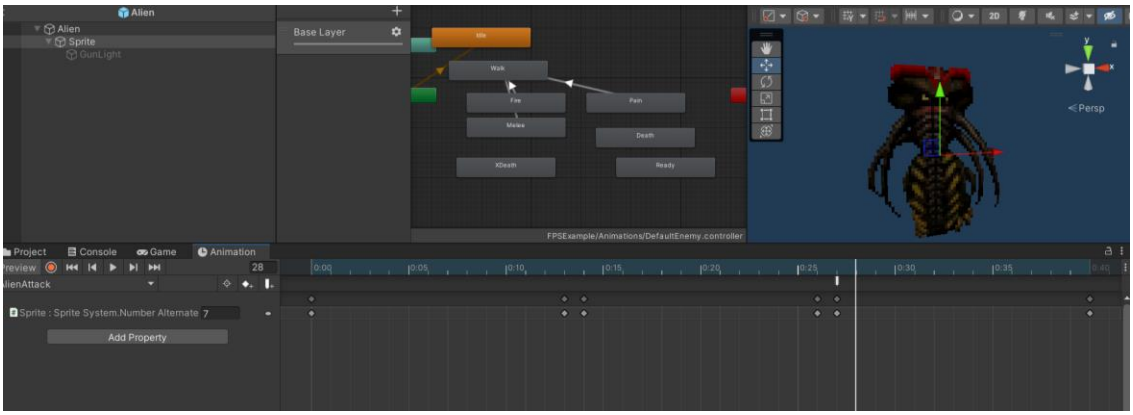
Animator Info: (Not as complicated as it looks)

The animations the monster use depends on how the monsters sprites are ordered. Typically, with the zombie, it has 4 sprites for walking, 2 for firing, 1 for pain and 5 for death, in that order, built off each other. This means any monster with this same sequential sprite layout as the zombie can reuse the same animation clips, such as the human. The sprite number is set through the script, which is controlled by the animation. You can see this in the bottom left of the image.



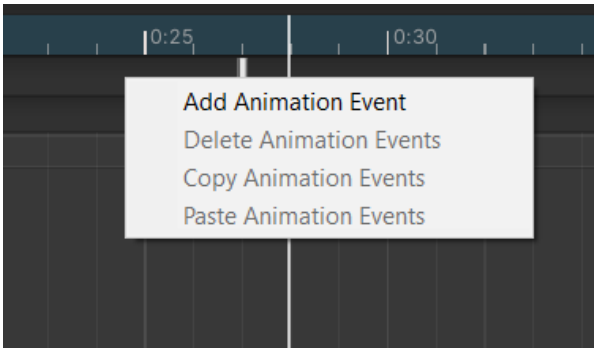
(NOTE: Make sure that you set the previous sprite again the frame before the new one in the animation, otherwise the timing of your sprite animation will be off due to blending!)

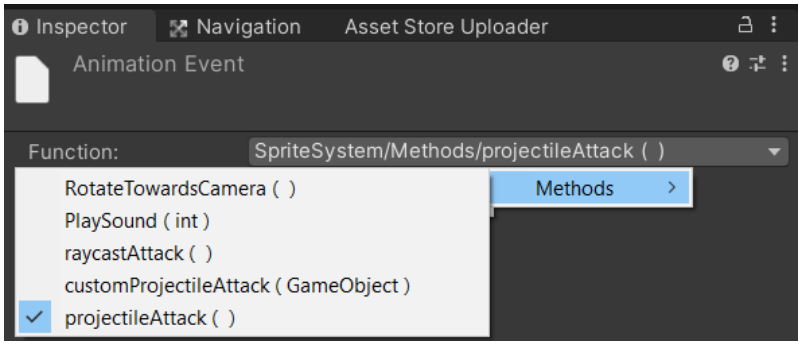
If however you look at the alien, you'll notice that when it fires, it takes up 3 sprites. This means that every other sprite index is offset by 1, so those animations afterwards will have to be remade to fit the alien. However, the walking can be reused, because it still uses the first 4 frames, same as the zombie.



Animation Events:

However, the animations get more interesting than just sprite layouts. How a monster attacks is entirely dependent on the animation events within the animation. To create an animation event, right click the black area right above the keyframes and below the timeline.





Here you're given a couple of functions. Ignore RotateTowardsCamera();

PlaySound(int index): Plays sound from clips: `0 sight, 1 idle, 2 attack, 3 pain, 4 death`

RaycastAttack(): Performs a Raycast attack. Used for shooting bullets and melee

RangedRaycastAttack(int range): Performs a Raycast attack within a certain range

ProjectileAttack(): Performs a Projectile attack using the default projectile

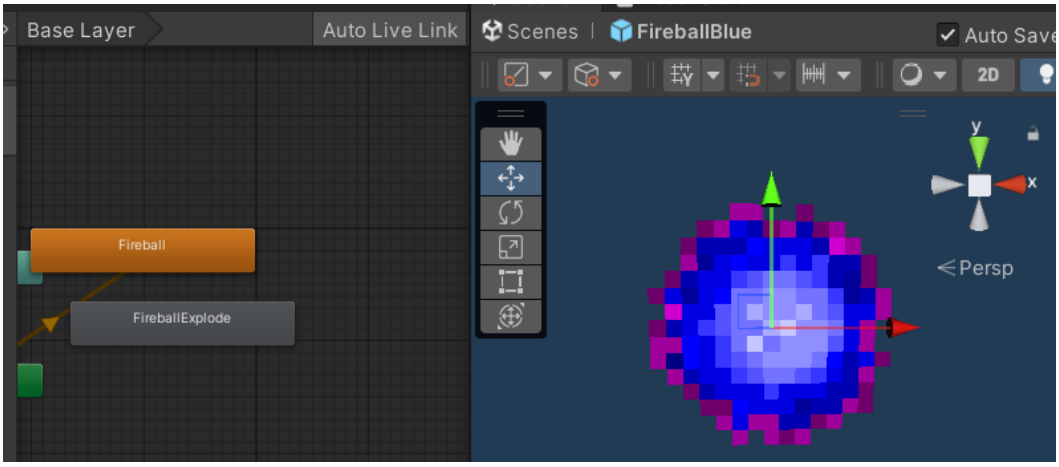
CustomProjectileAttack(GameObject projectile): Performs a Custom Projectile attack , useful if you want a monster with multiple projectile types.

ChargeAttack(): Performs a Charging attack, requires a rigidbody though

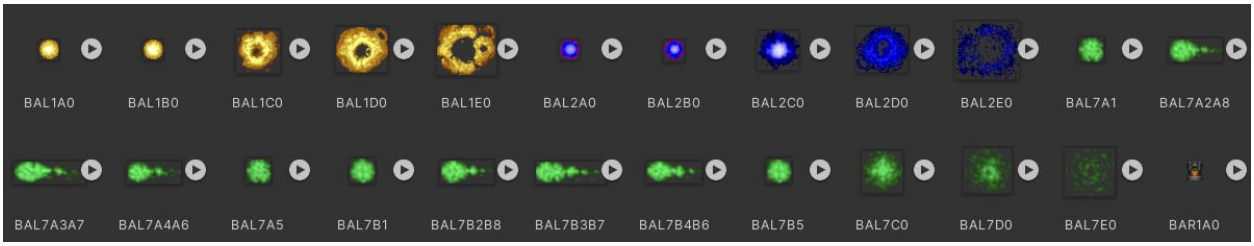
Obviously, a monster with a raycast attack can't use the same animation as one with a projectile attack. However, if two monsters have the same sprite layout but have different projectiles, they can use the same animations and the projectile can be set on their script.

4.Projectile Info

A good example of animations borrowing off each other are monster projectiles.



Since all these projectiles all have two sprites for flying, and three for exploding, they all can reuse the same animations! Yes, even the green projectile, which changes angles, can reuse animations because the angle is automatically set via script. All that matters it that the letters before the angle index match up. A=A, B=B, etc. This design drastically decreases the amount of work you would have to do, rather than having to manually set each sprite for each type in the animation.



As far as variables go, the projectile script is pretty simple. When spawned the projectile travels in the direction it is facing at the speed it's given. The source variable is used for infighting. When the projectile hits an enemy, it gives the source to the other monster. That monster then decides to consider that source as a target or not.

- `GameObject source`: The entity that fired the projectile. (used for infighting)
- `int damage`: The amount of damage dealt upon impact.
- `float speed`: The speed at which the projectile moves.

For the rocket projectile, it has an additional collider when it explodes of tag explosion that is enabled during the explosion animation. This allows it to do extra splash damage to those around it.

5.Player Info

FPSController Variables & Their Purpose

Movement Settings

- `float moveSpeed`: Defines how fast the player moves.
- `float jumpForce`: Defines the height of the player's jump.
- `float gravity`: Defines the gravity affecting the player.
- `bool hasBlueKey`: Tracks if the player has collected a blue key.
- `bool hasRedKey`: Tracks if the player has collected a red key.
- `bool hasYellowKey`: Tracks if the player has collected a yellow key.

Mouse Look Settings

- `float mouseSensitivity`: Determines how sensitive mouse movement is.
- `Transform cameraTransform`: The camera object controlling the player's view.
- `float verticalRotation`: Tracks the vertical rotation of the camera.

Shooting Settings

- `Camera playerCamera`: Reference to the player's camera for aiming.
- `GameObject hitEffect`: Visual effect for hitting objects.
- `GameObject bloodEffect`: Visual effect for hitting enemies.
- `Animator gunMovementAnimator`: Controls the movement animations of the weapon.
- `Animator gunAnimator`: Controls the shooting animations of the weapon.
- `string currentWeapon`: Stores the currently equipped weapon.
- `string[] weapon`: Stores the player's available weapons.
- `int[] ammo`: Stores the ammunition count for different weapon types. **(0 = bullets, 1 = shells, 2 = rockets.)**
- `TMP_Text ammoText`: Displays current ammo count.
- `TMP_Text[] ammoDisplays`: Displays ammo counts for different weapon types.
- `int ammoIndex`: Tracks which ammo type is currently in use.
- `GameObject[] projectiles`: Array of projectile prefabs available to the player.

Sound Settings

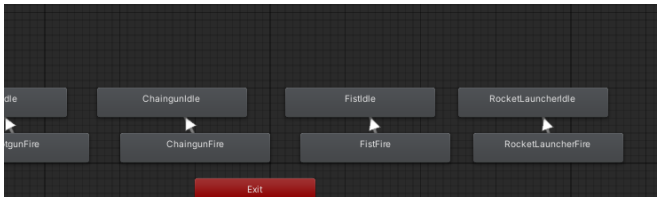
- `AudioClip[] landSound`: Sounds played when the player lands.
- `AudioClip[] pistolSound`: Sounds played when firing the pistol.
- `AudioClip[] shotgunSound`: Sounds played when firing the shotgun.
- `AudioClip[] painSounds`: Sounds played when the player takes damage.
- `AudioClip[] deathSounds`: Sounds played when the player dies.
- `AudioClip[] punchSounds`: Sounds played when the player punches.
- `AudioClip[] rocketSounds`: Sounds played when firing rockets.

Internal Components

- `string formerWeapon`: Stores the previously equipped weapon.
- `CharacterController characterController`: Manages player movement and collisions.
- `Vector3 moveDirection`: Tracks the player's movement direction.
- `AudioSource audio`: Plays various player-related sounds.

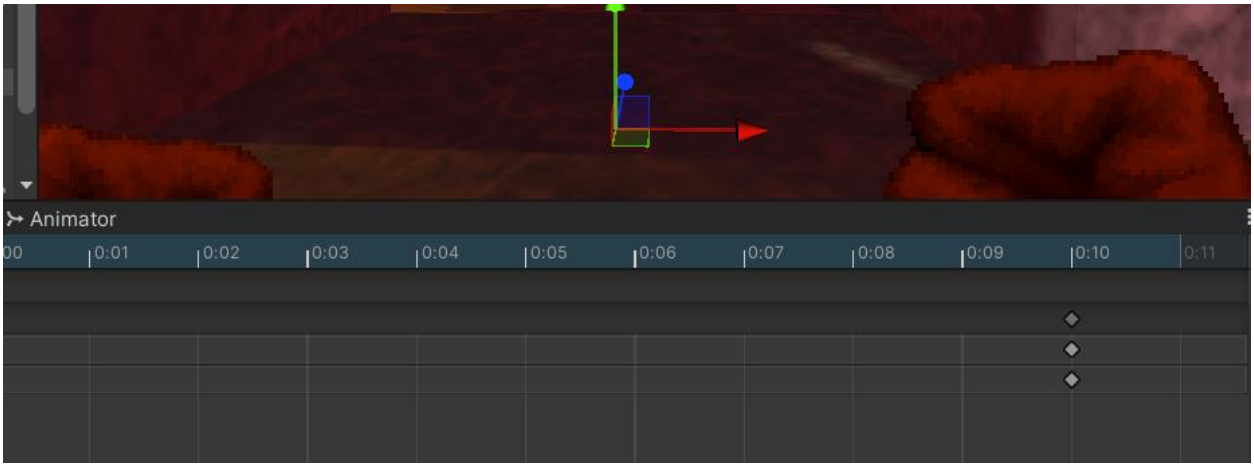
How to Add Custom Weapons, Part 1 (Animations)

So, let's take a break from the script and head into the scene. You'll want to go to the Weapon GameObject and open up the animator and animation tabs.

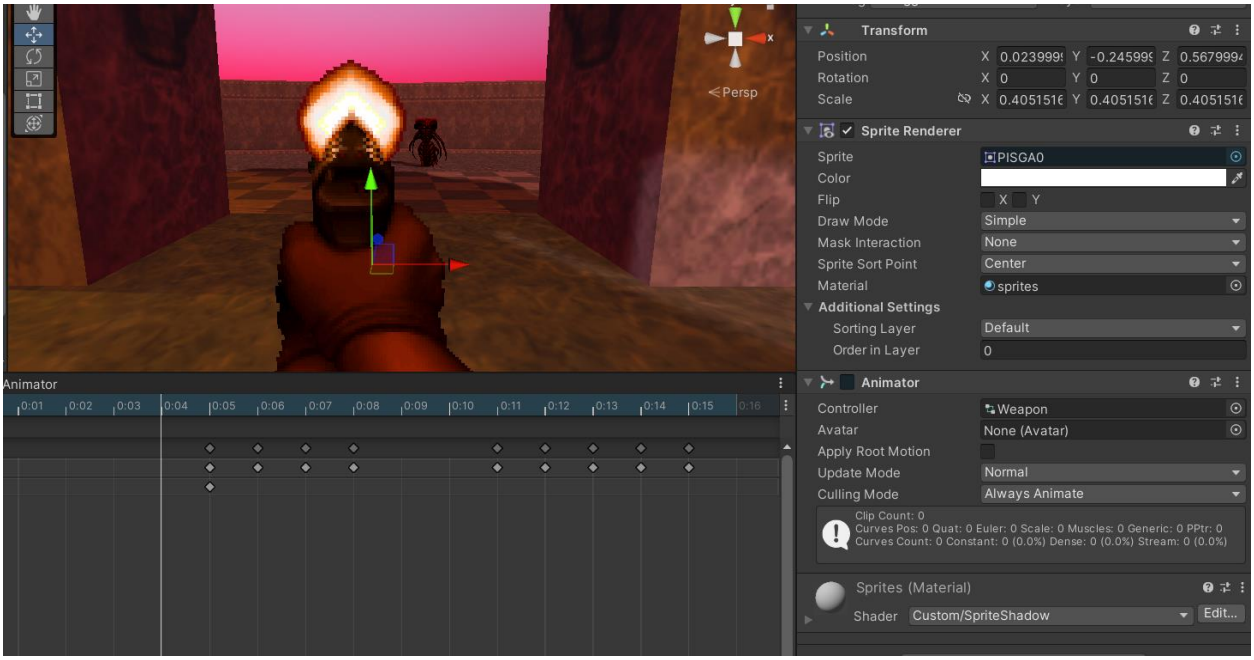


Here you'll see the predefined weapons. Create Two new animations, "(WeaponName)Idle" and "(WeaponName)Fire". Now hop into the animation tab.

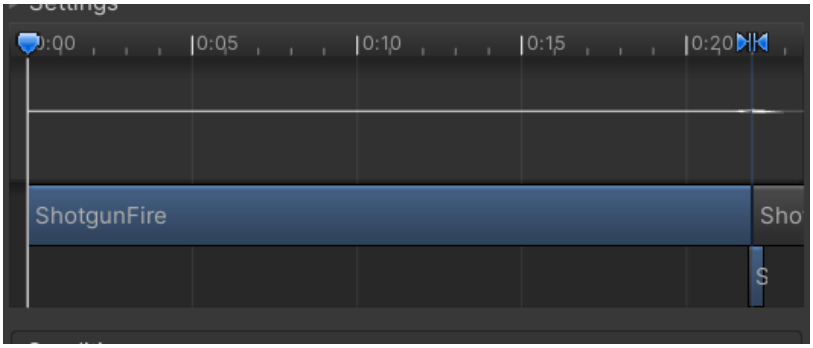
This time I decided to stick with the regular sprite approach, because most of the time weapons don't really borrow the same sprite layout and timing as each other. For the idle, set the idle sprite of your weapon for any period of time, as long as it's set. If it dosen't seem to be doing anything, make sure the record is enabled and try going back and forth in the sprite window. You can also set weapon position too.



Now onto the good bits, the fire animation. To make it simpler, there are no animation events for the weapons. If your gun flashes, which I assume it does, you can change the color of the flash to match the sprite, which is all stored in the child gameobject of weapon. You might need to move the flash to match the postion of the gun if it is initally off center. If you are struggling with this, look up a guide to animating sprites in unity.



Finally, when you're done, make sure to add a connection between the fire state and the idle state. For best results, set the exit time all the way to the very end of the fire animation.



How to Add Custom Weapons, Part 2 (Scripts)

Alright now for the second part, the scripting. Everything you'll need in the FPSController class will be in the handle shooting function, for the basics anyway. If you can hold the attack button down, AKA automatic, put it in the automatic if block. Otherwise, put it in the other one. Here's how weapon definition should look like:

```
if (currentWeapon == ("WeaponNameGoesHere") && ammo[(YourAmmoIndex)] > 0)
{
   formerWeapon = currentWeapon;//leave as is
   gunAnimator.Play(currentWeapon + "Fire"); //leave as is
   ammoIndex = (YourAmmoIndex);
   PlaySound(1);//you need to define a new sound array for your weapon, or reuse one
   bulletAttack(100, 1, 10);//performs a bullet attack, range 100, one bullet, 10 dmg
}
```

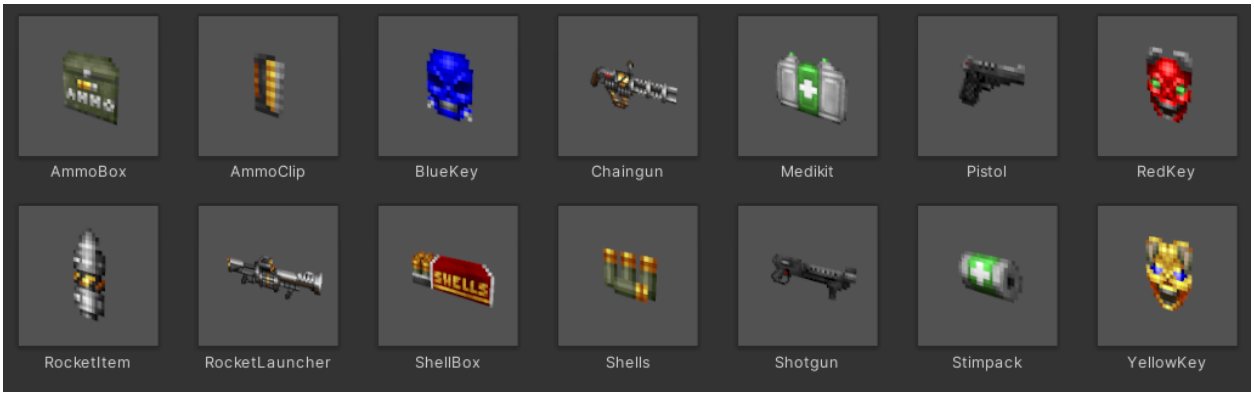
The last line, though bullet attack in this example, is important. If you want a shotgun blast, you'll need to change 1 to the number of pelts the shotgun fires, the more pelts the wider the blast. If you want a close range attack, you'll need to change the range to something like 1. If you want a projectile attack,

switch the bullet attack function for the projectile attack function. The index of the projectile is what will be in the inspector. By default 0 is rocket, but you can add more.

Additionally, if you want your weapon to play a certain sound on hit, (mostly for melee) you can add an additional if statement on raycast hit that checks if on weapon name, play a sound.

Whatever index you want your new weapon to be in, you'll need to assign that via an item, which is the section below. Remember, weapon name is super important, so make sure it's consistent across scripts and animations!

6.Item Info



Variables

- `int healthToGive`: Amount of health restored when picked up.
- `string weaponToGive`: Name of the weapon granted to the player.
- `int ammoIndexToGive`: Index of the ammo type affected. **(0 = bullets, 1 = shells, 2 = rockets.)**
- `int ammoToGive`: Amount of ammo granted when picked up.
- `int weaponSlot`: Slot in the player's inventory for the weapon.
- `bool givesBlueKey`: Determines if the item grants a blue key.
- `bool givesRedKey`: Determines if the item grants a red key.
- `bool givesYellowKey`: Determines if the item grants a yellow key.

Most of these variables should go unused. However, if for example you want to give a player the yellow key and have them get +10 health, you can do that just by editing variables. This makes custom items very easy. However, for a weapon you must specify the ammo that it gives and how much it gives, otherwise you'll just get an empty weapon with no bullets!

Making custom keys

To add a custom key, or some sort of new item for progression, you're going to have to update 3 scripts: Item, Door, and FPSController. Here's what you have to do:

FPSController.cs

Add a bool next to your other keys that stores whether player has a key or not.

Item.cs:

Create a new bool for giving a key. Then in the OnTriggerEnter function define a new if statement for your key. If true, give the player a key and set whatever element in the UI enabled to show the key to the player. The main script does this by searching the GameObject by name.

```
if (givesBlueKey) { player.hasBlueKey = true; GameObject.Find("BKEY").GetComponent<Image>().enabled = true; }
```

When doing this, make sure the key image gameObject is enabled, but the image component itself is disabled by default.

Door.cs:

Add your new requiredKey bool. Update this if statement with your new key, default line 25:

```
//if I don't require any keys
if (!(requireBlueKey || requireRedKey || requireYellowKey) || other.gameObject.CompareTag("Monster"))
```

Next, add a new if block for your new key, displaying whatever message you want to display:

```
if (requireBlueKey == true && player.hasBlueKey == false)
{
    unlocked = false;
    doorText.text = "Requires Blue Key Card";
    Invoke("fadeOutText", 1.5f);
}
```

This can be useful if you want to have npcs that get an item and open a door for you, a required status before entering a new level, or just another key.

Switches with doors:

The door class is pretty straight forward so we won't go over the script itself. However: Because the animator on the class is public, you can have a door that opens from anywhere, and doesn't require it to be face to face, AKA switches. Just make sure you have a way for monsters to open the door on the other side, otherwise they'll just phase through the door. Also feel free to add cooler door animations.