MegaMan Engine for Unity

Made by MegaChibisX

Player Creation

Creating a new player might not be within your interest in the scope of making a MegaMan game. It is a *MegaMan* game after all. However, to learn how to make new weapons, you will need some knowledge of the player mechanics. You should be able to simply reference the **Manual** to get a basic understanding of what each variable and method does.

In this tutorial we will make the **Power Adaptor**, which will act as a new player. That means the Adaptor can do things like use weapons and have its own abilities, separate from MegaMan's. For example, the Power Adaptor is unable to slide. It's punch buster won't be made here, as it is a weapon, and those will be handled in the next tutorial.

Sprite Sheet:

The first thing your player will need is a sprite sheet. The engine already comes with one for MegaMan, which you can use as a reference. You can duplicate it and replace each frame with your own, or you can just use your own. MegaMan's sprite sheet is found under "**Assets/Resources/Sprites/Players/MegaMan**". In any case, it's recommended to have sprites in your sprite sheet for the following:

- Teleporting
- Idle
- Running
- Jumping
- Climbing (Standing on the top of the ladder included)
- Sliding
- Damage normal
- Damage fallen

Also, **shooting** and **throwing** states, both having animations for the following:

- Idle
- Running
- Jumping
- Climbing

You can also add any other animations you are going to need yourself.

If you use the MegaMan sprite sheet included in the project, you will notice a few extra animations, specifically animations for the **Super Arm**, for using custom busters, for driving a ride armor and for screen-clearing attacks. Unfortunately all these features were planned to be added, and simply never were. There are currently no plans to add them to the project, but the sprites remain. If you wish to remove them from your own sprite sheet, feel free to.

Now that you have your own sprite sheet, make sure to put it in its own folder. The folder should be located inside the **Resources** folder, and it is heavily recommended that is is put under the "**Assets/Resources/Sprites/Players**" path for convenience. Rename your sprite sheet to **<PlayerName>_Full**.

If you used the engine's player sheet as a reference, you can also copy the **Meta File** that includes "**Full**" in its name. Put the **Meta** in your folder as well, and give it the same name as your Sprite Sheet's (with "**.meta**" at the end if you have file extensions showing in your computer). The file's format also needs to be included. Ex: "**MegaMan_Full.png**" or "**MegaMan_Full.png.meta**".
Copying the Meta will likely save you some time from **slicing** the sprites in the Editor, but extra time will spent making the Sprite Sheet anyway, so its your choice.

Setting up the image:

When you import your sprite sheet into the Unity Editor, you will need to

Sprite Slicing:

To separate your sprite sheet into sprites, they need to be separated in the editor first. This is explained much better earlier in the **Boss_Tutorial (2)**, so you can check that out if you're not familiar with this.

Open the **Sprite Editor** and split the sheet into single sprites. You can use the **Automatic Slice** or **do it yourself**. In either case, apply. You may also want to set the origins now, but those can be more easily set during the animation phase.

Animation:

The next step is making the animations. We will make the Stand animation first, and simultaneously have an **Animator** made by Unity.

Go to the **Project** window and find your Sprite Sheet. Open it up by pressing the arrow to the right of the image. It should now should every individual sprite, next to each other.

To make the Stand animation, select all the frames of the animation with **Ctrl** or **Shift**, and drag them into the **Scene** window. This should make a window popup, asking you to name it. The default location for animations is under "**Assets/Sprites/MegaMan/**", and you should make your own folder in there too. You can just leave them in the **Resources** for simplicity though.

Name your animation "**<PlayerName>Stand**" and select a folder. This should have made two different files, the **Animation** itself, as well as the **Animator**. Rename the animator to "**<PlayerName>Animator**". The animation should already have an appropriate name. If there is no prefix in the name you game, one should be added in the **Project** window for convenience. If there is one, you should go to the **Animator Window** and remove the prefix from the name inside the **Animator**, since these can only be accessed by script.

Go to the **Animation** window and make the rest of your animations. This process has also been explained in the **Boss_Tutorial (2)**, so you can check it out, under the **Animation** section, for details.

In the final product, you should have animations for the following:

- Intro
- Outro
- Stand
- Run
- Jump
- Climb
- Slide
- Hurt
- Fallen
- StandShoot
- RunShoot
- JumpShoot
- ClimbShoot
- StandThrow
- RunThrow
- JumpThrow
- ClimbThrow

You can simply copy the animations from MegaMan and change the sprites.

Once the animations are made, it is a good idea to try them out, and make sure they look good in the game itself.

We will start by simply duplicating the MegaMan **Prefab**, and replacing the Animator with our own. This can be done simply by following these steps:

- Make sure you are working on an already existing stage with features like ladders and, important, a player, and preferably MegaMan, already on it.
- Duplicate the MegaMan **Prefab** by pressing **Ctrl+D**, or by right clicking on the **MegaMan** GameObject and pressing "**Duplicate**".
- Rename the new GameObject to the name of your player.
- If you are using Unity 2018.2, with your new Player selected, to to the **GameObject** menu on top of the Editor, and select "**Break Prefab Instance**".
- If you are using Unity 2018.3 or above, right click on the new Player in the **Hierarchy** window, and click the "**Unpack Prefab Completely**" button.
- Disable the original Player GameObject.
- Go to the **Camera** GameObject, find the **Camera Ctrl** script, and assign your new Player into the **Player** field.

To replace the sprites with your own, you need parent the **Sprite** GameObject you created before to the new Player and delete the old one. However, as the Colliders of the Player are held within the **Sprite** object, they need to be transfered before the original **Sprite** object is deleted.

- Drag your **Sprite** GameObject from before into the new Player.
- Reset the position to (0, 0, 0), then align the feet of your Player with the original sprite's in the Y direction. (Ex: (0, 1.5, 0).)
- In the **Hierarchy** window, drag the **Colliders** GameObject from the old Sprite GameObject into the new one.
- Reset its position to (0, 0, 0).
- Delete the old Sprite GameObject.
- Go to the Parent GameObject that holds the **Player** script.
- Drag the **Sprite** GameObject into the **Anim** and **Sprite** fields on top of the **Player** script, which should now be empty and marked as "Missing".
- To the very bottom of the script, go to the **center** variable of the **Player** script, and subtract by the same amount you moved your sprite before. (Ex: With Sprite being at (0, 1.5, 0), from (0, -2.32, 0) the center should move to (-0, -3.82, 0).)
- Copy the **center** position variables into the **offset** field of the **Box Collider 2D** Component.
- If you want to edit the **size** variable of the **Box Collider 2D**, be sure to also edit the **width** and **height** variables in the **Player** script accordingly.
- Go to the **Colliders** GameObject, and similarly, move its Y position to the negative of its parent's value. (Ex: If the **Sprite**'s position is (0, 1.5, 0), move the **Colliders** to (0, -1.5, 0).)
- Go to the **Sprite**, and in the **Animator** change the **Update Mode** field to **Unscaled Time**.

This should now work. The player won't change colors while charging their buster, and they will only use the Mega Buster, but that's okay. This will be fixed now.

Similar to the **Enemy_Tutorial (3)**, to allow the player to change colors, we will duplicate the sprites, cut each color we want to change out, turn it to white, then play it on top of the original sprite.

Hopefully you placed your sprite sheet in a folder by itself, otherwise things may get cluttered.

Start by duplicating the sprite from **inside the Unity Editor** 5 times. This can be done easily with **Ctrl+D**. Rename the new files to the following, replacing the <PlayerName> with your own player's name:

- <PlayerName>_Blank
- <PlayerName>_Dark
- <PlayerName>_Light
- <PlayerName>_Outline
- <PlayerName>_Face

(If you didn't do it from the Editor, make sure you also duplicate and rename the **Meta Files** appropriately.)

Each of these files will have a different use in the game:

- **Blank** will simply be the original sprite, but all white. This is used when Speed Gear is used, to leave a trail of the player behind them.
- **Dark** will replace the darker of the player's armor's colors with the darker color the weapon of the weapon. For example, MegaMan's blue color will be used.
- **Light** is the opposite of **Dark**, it will replace the lighter color of their armor. For example, MegaMan's cyan parts.
- **Outline** should contain the outline of the player's sprite.
- **Face** should contain all the parts that weren't covered by the 3 sprite sheets above.

Time to edit each sprite sheet. You can use your favorite Image Editor for this job. I will personally be using **Clip Studio Paint**, but most Image Editing Software should work, as long as they are able to save transparent images. In no specific order, **Krita**, **Paint.net**, **Paint Tool SAI** and **GIMP** should all work very well.

Blank:
This one is easy. Simply lock the transparency on the layer and turn every single sprite to completely white. Save the sprite sheet.

Dark:
For this one you need to select every single pixel of the one specific color you'll be using for as the dark color of your character's armor. Most programs have a way to select a specific color from all over the image. Use that tool to select your dark color. If not, it might need to be done manually, so you'll have to suffer a little. With that one color only selected, **Invert the Selection**, and delete everything else from the image. **Ctrl+X** should quickly cut the color out in most programs. Now recolor everything here to white like before.

Light:
Do the same with the lighter color of your character. There is a decent chance that by selecting the light color, some parts of the sprite that aren't supposed to change color will also be selected. For example, with the Power Adaptor, the eyes of MegaMan are also selected. These will have to be erased manually, either by removing them from the selection before you erase everything or removing them with the erase tool after.

Outline:
The outline will also work the same way. There is an even bigger chance now than before that parts of the face or other parts you don't want recolored will be selected. These will also need to be edited manually.

Face:
The face sheet will contain everything that the **Dark**, **Light** and **Outline** sheets didn't include. This can easily in most software:

- Import each of the previous sprite sheets into the **Face** one.
- Merge the 3 layers into one.
- Give the layer a color which doesn't appear in your sprite sheet.
- Merge the new layer with the original one.
- Select the unique color like before.
- Remove the color like before.
- Save.

Now the sprites should appear in the Unity Editor, and most importantly, when you look at them in the Sprite Editor, they should all be sliced the same way as the original sprite sheet. If that is not the case for some reason, open the folder outside of Unity, remove the previous Meta Files of all the sprites **except** from the Full sprite, and then duplicate and rename then appropriately.

Now it's time to show the color changing sprites within the game.

Go to your Player GameObject in the **Hierarchy** window and find the **Sprite** child. Right click on it and make a new GameObject, which should now be parented to the Sprite GameObject. Make sure its position is (0, 0, -1) and rename it to "**Colors**".

Create a new GameObject inside the **Colors** GameObject. Give it a **Sprite Renderer** Component and a **Copy Sprite** one.

Drag the **Sprite** GameObject from the Hierarchy window into the **Rend To Copy From** field.

In the **Sprite Path** field, you need to write down the path of the folder your sprites are found in, but must start from inside the **Resources** folder. For example: Let's say your sprites are found in the path "**Assets/Resources/Sprites/Players/MegaMan_Power/**". In the field, write "**Sprites/Players/MegaMan_Power/**" (Slash at the end should be included.)

Now that the components have been set, duplicate the GameObject 3 times, for 4 GameObjects total. Rename them to "**LightColor**", "**DarkColor**", "**Outline**" and "**Face**". For each one of them, add the name of the fitting **Sprite Sheet** at the end of the **Sprite Path** path.

Now go to your player's script component and assign the **Colors** GameObject to the **Sprite Container** field.

And finally, time to make the speed gear trail for your player. Navigate to "**Assets/Prefabs/Player**" and duplicate the "**MegaMan_SpeedTrail**" GameObject. Rename it appropriately, and go to to the **Sprite Over Time** component. Open the **Gradient** and change the colors as you see fit. Assign the new prefab to the **Speed Gear Trail** variable in your Player's script.

The speed gear will use MegaMan's Blank sprite for now, but that will be fixed soon.

Testing your game now, your player is probably a white blob with a face. This is because these objects also need to be assigned in the **Player** script. Go to your player script and drag the **LightColor, DarkColor** and **Outline** GameObjects into the **Body Color Light**, **Body Color Dark** and **Body Color Outline** fields. Now look for the **Default Colors** label, and open the dropdown menu. You should see 3 colors. With the assistance of your Image Editor, change them to the appropriate colors of your player.

The player should now be fully playable. It's time to save your player as a prefab. Navigate to "**Assets/Resources/Prefabs/Players**" and drop in your player. Now it's time to make your player selectable from the pause menu.

Start by navigating to the path "**Assets/Resources/Sprites/Menus**". This is where you will find the display icons of the characters, as they are going to appear in the menu. Open the "**Characters**" file with your favorite image editor and replace one of the MegaMans in the last lines with your own character. Hit save.

Now go to to the **GameManager** script. Fairly high up you will find the **Players** enum, which lists available and possible players so far. Add your own's name to the end of the list. Now go to the bottom. Find the **GePlayerMenuSprite(Players pl)** method, and inside the **switch** statement add the following:

```
        case Players.<PlayerEnum>:
          return Resources.LoadAll<Sprite>("Sprites/Menus/Characters")[<Position>];
```

Replace the <PlayerEnum> with your player's name you gave in the enum and the position with the position you put your character into in the image from before. Count left to right then top to bottom, starting from 0, to find your player's position.

Next go to the **GetPlayerPrefabPath(Players pl)** method. Again, inside the switch statement, add the following:

```
        case Players.<PlayerEnum>:
          return (GameObject)Resources.Load<GameObject>("Prefabs/Players/<PlayerName>");
```

Again replace <PlayerEnum> with the enum you gave and <PlayerName> with the name of your player's GameObject. If your Player is not in the Players folder, make the appropriate changes. If they are not in the **Resources** folder at all, relocate them.

Now go to the **MenuPause** script. Inside the **Start(Player player)** method find where the **PlayerData** is defined, and add your own player at the end, beginning or middle of the array. Use the following line:

```
      new PlayerData(GameManager.Players.<PlayerEnum>)
```

Again, make appropriate adjustments.

Finally, in the **Player**'s inspector, change the **Cur Player** field to the enum you gave.

You may remember that we also set something aside a page ago.

Go to the **Player** script again, and look for the **MakeTrail()** method. Pretty high up you should find a **switch** statement. Inside it, add the following line:

```
          case GameManager.Players.<PlayerEnum>:
            blankSpritePath = "<Blank_Image_Path>";
            break;
```

Again, replace <PlayerEnum> and <Blank_Image_Path> appropriately.

Also go to the Player Component in the **Inspector** and change the **Cur Player** enum as well.

Now your player should be selectable from the in-game menu! Congrats!

...but your character might not be completed yet. You may want to change the speed of the player, for example. There are a few different things you can change for your player, simply by looking at their component in the **Inspector**.

Make sure you select your player's prefab, and not the one from the scene. But if you do edit the scene one, you can simply **apply** or **revert** the changes, so don't worry.

Here are the things you can change from the **inspector** to make your player act differently, from top to bottom:

- **Bars**: These include the health bar, weapon bars and other UI.
- **Deathplosion**: Your player may die differently.
- **SFX Library**: This is where all the sounds of your player are held. You can change any of them.
- **Max Health**: You can change the max health of your player. It will go off screen if it's too high, and if it's set to 0, everything will insta-kill your player.
- **Move Speed**: The normal movement speed of your player.
- **Climb Speed**: How quickly your player climbs a ladder.
- **Jump Force**: How high your player jumps.
- **Max Jumps**: How many jumps your player can make. Set to 0 to ground them.
- **Slide Type**: This checks whether your player slides, dashes or does neither. Slides and dashes act differently.

Custom Scripts:

But what if you want to make even more changes to your character? What if you want to, for example, give flight to your character, the same way the Jet Adaptor does, how would you do that? This is when it's time to make a new script! Such joy!

If you want to make a new script, you're most likely on your own. I can't predict or explain how you're going to make your unique character. What I can do, however, is tell you how to set up that character and give you some insight into the inner workings of the **Player** script, so you'll know what to use and when.

Let's start by making the script and ensuring it works. Let's set it up correctly.

Start by making the new script. Player scripts are placed in the **Assets/Scripts/Player** folder, and it is recommended that you use the same folder.

Create a new **C#** script, and give it an appropriate name. Remove everything inside the class body (the **Start()** and **Update()** methods, as well as the comments), and replace "**MonoBehaviour**" with "**Player**". Your script should look something like this:

```
public class PlayerExample : Player
{

}
```

Now go back to the Unity Editor. Since your player is now a prefab, to make significant changes to them, you will have to edit them from the **Prefab Editor**. Simply select the prefab in the **Project** window and **double click** it, or **right click** and select "**Open**".
(If you are using **Unity 2018.2**, ignore this step, as the prefabs were updated in **2018.3**. You can simply apply the changes once you're done there.)

You can either remove the **Player** script now and replace it with your own script, or you can keep it around as a reference, and remove it once you've set all the values in your custom script. It is suggested you keep it around, as you can simply copy the important values from there. You will have to remove the original script by the time you try out your player.

Required Component Variables:

Here's a list of the variables that **MUST** be replaced, and what values they should have. If I forgot any, you'll likely get an error. Simply check MegaMan's prefab as a reference.

- **Anim**: This is your player's animator. The **Sprite** GameObject.
- **Sprite**: The base **Sprite Renderer** of your Player. Again, the **Sprite** GameObject.
- **Bars**: The health, weapon and gear bars. You can change those, but the defaults are the following:
  --- Basic bars ---
  - **Empty Bar: Bars_0**
  - **Health Bar: Bars_1**
  - **Rush Bar: Bars_63**

  --- Gears ---
  - **Gear Background: GearBar_0**
  - **Speed Gear 1: GearBar_5**
  - **Speed Gear 2: GearBar_6**
  - **Power Gear 1: GearBar_3**
  - **Power Gear 2: GearBar_4**
  - **Double Gear 1: GearBar_1**
  - **Double Gear 2: GearBar_2**
- **Cur Player**: Your player's **enum**.
- **Default Colors**: The normal **armor colors** of your player.
- **Normal Col**: The collider your player normally uses. It's in the **Parent GameObject**.
- **Slide Col**: The collider your player uses while sliding. It's called "**SlideCollider**" and is found under "**Parent_Player > Sprite > Colliders > SlideCollider**"
- **Sprite Container**: That is the GameObject that holds all the Color Changing sprites. This is called "**Colors**" and is found under "**Player_Parent > Sprite > Colors**".
- **Audio Weapon**: The **Audio Source** that handles weapon-related audio clips. It's in "**MegaMan_WeaponSounds**". You can rename it appropriately, but it's not necessary.
- **Audio Stage**: Another **Audio Source**, this one handles sounds related to the player's movement, like landing and damage. It's called "**MegaMan_StageSounds**", and again, can be renamed.
- **Body Color Light**: The **Sprite Renderer** that holds the Light Color sprite fragments.
- **Body Color Dark**: The Sprite Renderer that holds the Dark Color sprite fragments.
- **Body Color Outline**: The Sprite Renderer that holds the Outline Color sprite fragments.
- **Speed Gear Trail**: The GameObject that is used as a trail during the Speed Gear and Dashing. It should be a prefab found under "**Assets/Prefabs/Player**", and you likely made your own version.
- **Gear Smoke**: The smoke that gets made when the player overuses their Gear system. It's called "**ExhaustSmoke**" and is parented to the Player GameObject.
- **Gear Bar**: This is the GameObject that displays the Gear bar above the player's head when it's in use. It is called "**GearGause**" and uses a Cutout Shader. You absolutely do not need to worry about shaders in this project, and important ones have already been provided.
- **Death Explosion**: This is the effect made once the player dies. It's found under "**Assets/Prefabs/Misc**".
- **SFX Library**: This holds all the sounds your player can make. It's a class found under the **Player** script, and you can make appropriate changes to it. In the **Editor**, the Sound Effects can by found under "**Assets/SoundEffects/NES Sound Effects**" and the folders inside.
- **Width**, **Height** and **Center**: To the very bottom of the Component you can find the rest of the important variables. These need to match the **Size.x**, **Size.y** and **Center** of your **Box Collider 2D**

Be sure to remove the original Player script from your Player GameObject now.

Apply the changes to your prefab.

With all these variables set, you should be able to use your player as normal. Playing around with them if you want.

Other Component Variables:

There are other variables you can also change to make your player act differently:

- **Max Health**: You can change the max health of your player. It will go off screen if it's too high, and if it's set to 0, everything will insta-kill your player.
- **Move Speed**: The normal movement speed of your player.
- **Climb Speed**: How quickly your player climbs a ladder.
- **Jump Force**: How high your player jumps.
- **Max Jumps**: How many jumps your player can make. Set to 0 to ground them.
- **Slide Type**: This checks whether your player slides, dashes or does neither. Slides and dashes act differently, with the slide allowing the player to move through one-block tall gaps and the dash allowing the player to Dash Jump.

Important Script Variables:

On top of the variables you need to be know of in the Component menu, there are some more important ones for those of you wishing to write one or more scripts:

- **instance:** This is a **static** variable, meaning it can be accessed by any script without knowing which script is the player's. This is better than using **FindComponentOfType** for cases like this, where there is only one player in the stage at any time. This variable can, however, be empty. For that reason, you need to make sure it isn't empty before using it. For the player's last position, the **GameManager** can be used, as it tracks their position. You can also add more variables you want to keep track of by defining them in the **GameManager** and updating them through the **UpdateGameManager()** method.
- **timeScale:** This keeps track of the Player's local time scale. This means the player can move at an independed speed from the rest of the game, for example during the Speed Gear or when executing a screen clearing attack. This should **not** be changed by itself. Use the **SetLocalTimeScale()** method instead, as the physics will get messed up otherwise. **SetLocalTimeScale()** should be called whenever Unity's **Time.timeScale** is changed as well, if you want the player to move independedly.
- **deltaTime** and **fixedDeltaTime**: These should be used instead of Unity's **Time.deltaTime** and **Time.fixedDeltaTime**.
- **input**: This keeps track of the keyboard input. Use this instead of **Input.GetAxis()** every time, and it will also snap the input to an integer. If it didn't, the player would get compressed and move slower and faster based on the controller's input.
- **currentWeapon**: This is the player's current weapon. We'll go more in depth in the next tutorials, but this holds the important information of your weapon, like its colors, name, energy and abilities.
- **defaultWeapon**: This is your player's default weapon. It is what they will start with.
- **weaponList**: This one is a list of all the weapons your player has available. It's an enum. While weapons can be manually added, they usually have a condition that needs to be met. Check **RefreshWeaponList()** or make your own.
- **currentWeaponIndex**: This holds which weapon is the active one right now.
- **pauseMenu**: This is where the **Pause Menu** variables are held. The **Pause Menu** is a class by itself, and you can indeed make your own, but it's doubtful that a tutorial for this will be made by the writing of this tutorial.
- **paused:** Checks if the game is currently paused.
- **cutscene**: This holds the current cutscene that's played, if it needs to be played in-game. (Not fully implemented yet).
- **state**: The player can have different stages, depending on their current action, like climbing or being hurt.

- **canMove**, **canBeHurt** and **canAnimate**: These variables freeze the player, make them invincible and stop their normal animations. These should all be used for special scenarios, abilities, attacks and movements.
- **health** and **maxHealth**: The player's current and maximum health.
- **knockbackTime**: This timer keeps track of how long the player gets knocked back after being hit.
- **invisTime**: This timer keeps track of how long the player will be invincible after being hit. Different from **canBeHurt**, as the player flickers while this is active.
- **gearAvailable**: This checks if the player can use the Double Gear. If the player has acquired the Double Gear, then in the **GameManager** the variable **item_DoubleGear** will be true. The Double Gear is given to the player by default right now, but you can change that easily by changing **item_DoubleGear** to false in the **GameManager**'s constructor.
- **gearGauge**: That's how much gear energy you have left.
- **gearActive_Speed** and **gearActive_Power**: These variables check which gear is active. Both can be activated at once. They shouldn't be modified directly. Instead, use the **SetGear()** method.
- **gravityInverted**: It checks if the player's gravity is inverted. Do not modify directly, use the **SetGravity()** method instead.
- **gravityScale**: The strength of the player's gravity. Do not modify directly, use the **SetGravity()** method instead.
- **gravityEnvironmentMulti**: This multiplies the player's gravity based on some objects from their environment. It resets and checks in every **Fixed Update**. The **Stage_GravityScale** script changes that variable, and other similar scripts should also modify this one, but it doesn't apply for water.
- **windVector**: The direction and strength of the wind. Used by the **Stage_WindZone** script.
- **jumpsLeft**: This checks how many jumps the player has left. Normal value is **1**, but when on the ground, it will have the same value as the **maxJumps** variable. When the player is off the ground, whether they jumped or fell, it will be lower than **maxJumps**. The player can jump again mid-air if **jumpsLeft > 0**.
- **slideType**: Since the player might be able to **slide**, **dash** or **neither**, an enum is used instead of a boolean,which would be called **canSlide**.
- **slideTime**: This timer checks for how long the player still slide.
- **dashing**: This checks if the player is currently dashing. It stays true if you chain a jump with a dash, and allows for dash jumping.
- **chargeKeyHold**: This used to measure the player's charge. Now it's just used to make sure the **Release()** method of your weapon isn't constantly triggered when the weapon button isn't held.
- **shootTime** and **throwTime**: These timers are used to keep track of the current animations. If the shooting animations are played, **shootTime > 0**, if the throwing animations are played, **throwTime > 0**, and if neither happens, then the normal animations play. Check the **Animate()** method to better understand how they work.
- **gearTrailTime**: Timer that decides when to create a trail sprite.
- **gearRecovery**: Checks if the player has exhausted their gear fully and can't use the gears now.
- **lastLookingLeft**: Checks the direction the player last looked at.

Methods:

On top of the variables, there are also methods the **Player** script comes with. Most of them are marked as **virtual**, to allow you to build on them from other scripts deriving from the **Player** script. They should be sufficiently commented, and if you are using **Visual Studio**, the **Go To Definition** and **Find All References** options should help

Examples:

Some players already use custom scripts in the engine, and it usually works as a minor addition to the base player script. Examples:

ProtoMan:

ProtoMan works the same way MegaMan does. The only difference is his shield, which destroys enemy bullets when ProtoMan is jumping, unless the jumping or throwing animations are playing.

A small addition to the **Player** script is all that's needed to implement this behavior.

The **shieldCol** variable is added, to which ProtoMan's shield is assigned. The shield is a **Kinematic** Rigidbody2D parented to the Player GameObject, with a **PlWp_Shield** script attached to it.

The shield needs to be enabled only when the jumping animation plays. Since that depends on the player's **state**, it is best handled in the **FixedUpdate()** method. Specifically, the **HandlePhysics_Movement()** method is the perfect method to override. This handles a lot of the player's states, so the shield's state should be handled after the player's. It overrides the method, runs it, and then sets the shield's status.

Bass:

Bass's script mostly depends on his weapon, which will be made in the next tutorial. The one thing his script does is handle his shooting animations. Bass can shoot in 4 directions. For that reason, an enum is used to keep track of where Bass is currently shooting.

Since we can keep track of the animations, it's time to override the **Animate()** method. The entire method is copy-pasted from the original. The section where the suffixes are added is simply expanded, and now Bass is ready to use his weapon. Other players should also be able to use his weapon, but they won't be able to shoot in multiple directions.

Jet Adaptor:

The Jet Adaptor is a bit more complex than the other two. First of all, two extra variables are defined. **shouldBeInJet** checks if the player is currently flying. **jetTime** is a timer that keeps track of how much fuel you have left. Fuel refills quickly when you are touching the ground.

The method that's mostly used by the Jet Adaptor is the **HandleInput_Movement()** method. This is where functions like the **jump** and **slide** are handled. The jet takes priority, as if the player is flying, they don't need to check for sliding and jumping.

Activating the jet takes priority. Before everything else, Jet checks if the player presses or releases the jump button mid-air. If that happens, flight is enabled or disabled.

After that, it checks if the player is touching the ground. If they are, they refuel.

Finally, if the player should be flying, the flying behavior is fired. If not, normal input is handled.