Lei-Get’s First-Person Camera Controller for Godot 3.x

By Lei-Get Morzmorality (James Russell)

Table of Contents

Click a section to go to it.

Section 1: Project Overview

Chapter 1: Quick Intro

This is a relatively comprehensive FPC (First-Person Camera) controller project for Godot 3.x. It includes:

- Mouse-look.
- Keyboard movement (no joystick).
- Jumping
- Stepping up stairs and platform.
  - Camera interpolation when stepping up.
- Context sensitive “Use” functionality (like opening doors, pushing balls, etc.).
- Slope angle sensitive acceleration/deceleration.
  - Can be attenuated or turned on or off.

The main purpose of this project is to help people (including myself) learn how to program a player character in a video game. These principles in this project can be applied to any kind of game. After understanding the basics, it’s a simple matter of going forward and programming, little by little, step by step, any kind of character you need. For instance, jumping in a 3D FPS can be the same as jumping in a 2D game.

I hope I explain well enough how this project works and how to implement it into your own project.

Chapter 2: Structure of Project

This section details the structure and use of the files inside the project.

Project Directory Structure

The project folder structure looks like this:

- **Godot_FPC_Base**
  - **blends/**
    - Contains the Blender files of imported meshes:
      - “**base.blend**” contains the level geometry.
      - “**FPC_Base_Mesh**” contains the player character’s basic mesh geometry.
  - **coll_shapes/**
    - Contains any collsion shapes.
    - Only contains the invisible walls around the level geometry. This is here because I don’t want any visible geometry, so I just made the walls in Blender, exported them as “.dae” files, created a collision shape in Godot from it, saved it, and deleted the mesh node.
  - **dae/**
    - The exported dae files for level geometry.
  - **documents/**
    - Contains documents pertaining to credits of assets used, asset licenses, and this manual.
  - **environment/**
    - Contains the default environment for the project.
      - It has been set as such in the “Project Settings” in the editor, so any new scene created in this project will have it, by default.
      - Can be overridden by a “**WorldEnvironment**” node, of course.
  - **fonts/**
    - Contains fonts used in this project.
    - Only contains the “Almanack” font.
  - **scenes/**
    - Contains all the scenes used in the project.
    - Has the base level scene, the Godette model, the player scene, and the main scene that everything is run from.
    - **DEBUG/**
      - Contains scenes useful for debugging if you are experiencing problems or are coding new features.
      - Contains the collision sphere and position visualization axis scenes.
  - **scripts/**
    - Contains all scripts for things like the player, elevator platforms, scene initialization, the spheres that can be pushed by the player, GUI and DEBUG elements.

        - **DEBUG_Elements/**
            - Scripts for the collision spheres and position visualization axis.
        - **GUI_Elements/**
            - Scripts for the GUI elements, which are really just the cross hair, it’s red circle, and the debug label.
    - **textures/**
        - **dark_metal_01/**
            - A metal texture I made.
            - You can use it however you want.
        - **godette**
            - A model from SirRichard94 of Godette, the unofficial Godot mascot.
        - **hdr**
            - Contains a free HDRi from HDRI Haven.
        - **HUD**
            - HUD textures.
                - Just the cross hair and its red circle.
        - **Imperfections**
            - An imperfection texture I made with GIMP.
            - You can use it however you want.
        - **reference_textures**
            - Contains any textures to be used as reference.
                - Just a image that has two lines intersecting at the center of the screen.
        - **xcf**
            - The GIMP project files, just in case someone wants them.
    - **CREDITS**
        - Just credits some people who made assests that are used in this file.
    - **export_presets.cfg**
        - Godot-generated presets for exporting the project.
    - **icon.png**
        - The custom icon I made for the project.
    - **LICENSE**
        - The license of this project, which is the MIT license.
    - **project.godot**
        - The godot project file thing.
    - **README.md**
        - The GitHub readme.

Godot Project Structure

This is how the project itself, when opened in Godot, is structured. I’m not going to completely spell it out as you can just open the project and see for yourself.

- **Main.tscn**
  - **Env_and_Lights**
    - Contains the world environment, which uses the “**environment/Main_Env.tres**” resource.
    - Also contains the sun lamp.
  - **Base**
    - The scene that contains the base level geometry.
    - Also, it is the parent of the platform, sphere, and miscellaneous static body nodes.
  - **GIProbe**
    - Just the GI probe.
  - **Player**
    - The player scene node, which is “**scenes/Player.tscn**”.

Input Map

The input of the project is simple. All the used actions look like this:

- **ui_cancel**
  - Key: Escape
  - Exits the program.
- **Player_FW**
  - Key: W
  - Moves play forward.
- **Player_BW**
  - Key: S
  - Moves player backward.
- **Player_Left**
  - Key: A
  - Moves player left.
- **Player_Right**
  - Key: D
  - Moves player right.

- **Player_Jump**
  - Mouse: Right Button
  - Key: Space
  - Makes player jump.
- **Player_Use**
  - Key: E
  - Uses any usable object that is being pointed at.
- **Player_Shift**
  - Key: Shift
  - Causes the player to move faster when held.
- **Toggle_Fullscreen**
  - Key: F4
  - Toggles fullscreen.
- **Player_ToggleDebug**
  - Key: F3
  - Toggles the visibility of the debug label in the player scene tree.

Extra Information

How to use the debug scenes and scripts is explained in the “Use of Debug Scripts and Scenes” section.

Chapter 3: Design of Player.tscn Scene

The main player scene file is “**Player.tscn**”. It is simply linked inside any scene you want to use it in. The “**Player.gd**” script is linked to it inside the player scene file, but you could link another script to it inside the referencing scene, if you wanted to.

Player Scene Overview

Here’s the overview of the player’s scene file:

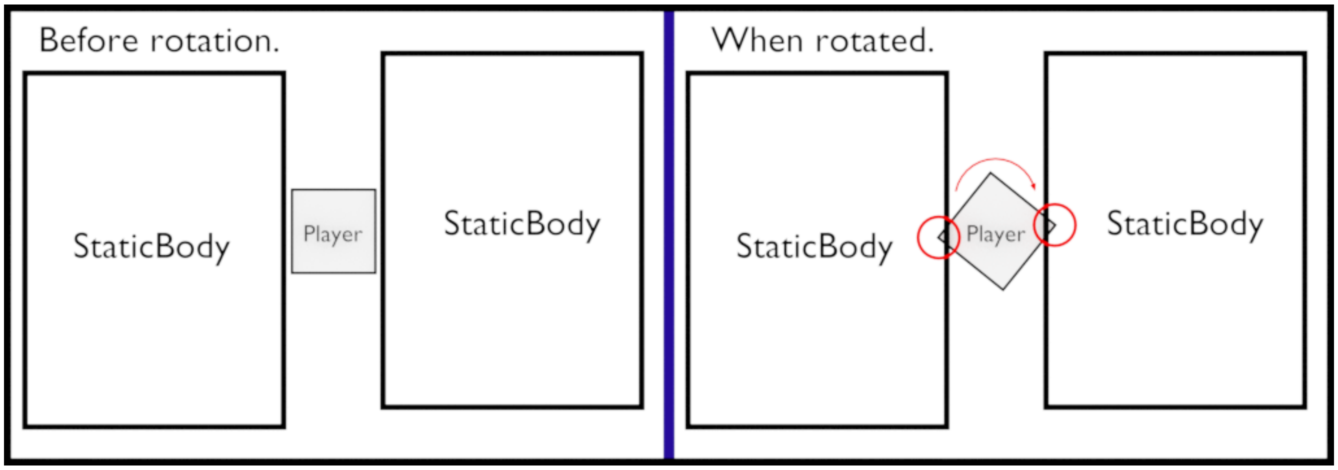
- **Player (KinematicBody)**
        - The main part (obviously) of the scene. Is linked against the **“Player.gd”** script.
    - **FPC_Base_Mesh (MeshInstance)**
        - The mesh.
        - Uses “**FPC_Base_Mesh.dae**”.
    - **CollisionShape (CollisionShape)**
        - The collision shape of the player.
        - It is a capsule.
    - **Camera_Main (Camera)**
        - The 3D camera that is the eyes of the player.
    - **Camera2D (Camera2D)**
        - The 2D camera used for the HUD and debug information.
        - **Crosshair (TextureRect)**
            - The cross hair texture node.
            - **Crosshair_Useable (TextureRect)**
                - The red circle that shows up when the player is looking at something he can use.
        - **DEBUG (Control)**
            - The parent node for all debug information.
            - **Debug_Label (Label)**
                - The text that can be used for debug information, like showing the player’s velocity or direction.
                - Can be edited in the player script using “**Debug_Label_String = str(Whatever)**”.
                - Can be set in the player script using “**Debug_Label.set_text(Whatever)**”.
                    - This already exist at the end of the file, so you really don’t need to set it twice.

Design of Player.tscn

The design is simple.

The "**Player**" kinematic body serves as the base node. This will do all the movement and physics. The visible 3D part of the player scene is just "**FPC_Base_Mesh**". Note that in this node's settings, this mesh is only visible on layer 20. In the "**Camera_Main**" node, the cull mask allows every layer but layer 20 to be shown, therefore culling out the player mesh. This is done so the player mesh doesn't get in the way of the camera and look weird.

The "**CollisionShape**" node is a capsule shape. This is because if a player had a box collision shape and where to turn while in a confined space, the collision shape would get stuck inside the bodies that make up the confined space.



Also, if the character were standing in a crevice with two angled walls and no floor, rotation will cause him to move up and down as the corners of the box push against the walls.

The "**Camera_Main**" node is simply the main node through which the player views the world. It's local position in this scene is interpolated when moving up steps.

The "**Camera2D**" node is the main view of whatever 2D elements are on the screen, which in this case are the cross hair and debug information.

Scripts Inside Player.tscn

There are 4 scripts attached to node inside the player scene.

- **Player.gd**
- **Crosshair.gd**
- **DEBUG.gd**
- **Debug_Label.gd**

Player.gd

“**Player.gd**” is further detailed in "Section 2: Player.gd Script", but the basic idea is this:

- There are a bunch of variables at the top.
- It has 4 custom functions for use within the script.
- It has the “**_ready()**” function.
- It has the “**_physics_process()**” function, of which the overview is:
    - Check for input.
    - Set states according to input.
    - Check if the player is looking at something usable and react accordingly.
    - Calculate horizontal movement.
    - Calculate vertical movement according to if the player is:
        - On a floor.
        - Jumping.
        - Falling
        - On a slope.
    - Apply movement using “**move_and_slide()**”.
    - Get states once again.
    - Execute “**Step_Player_Up()**” function.
    - Check for camera interpolation and react accordingly.
    - Execute “**Touch_CheckAndExecute()**” function.
    - Set the debug label text.

Crosshair.gd

“**Crosshair.gd**” has a “**_ready()**” function and a “**ViewportSizeChanged()**” function. In the “**_ready()**” function it calls “**ViewportSizeChanged()**” and then connects it to the main viewport node’s “**sized_changed**” function, so that whenever the size of the game window changes, so it can update the size and position of the crosshair and the red circle that denotes when a usable object is pointed at.

“**ViewportSizeChanged()**” changes the size of the crosshair and its child according to the size of the game window. It first set size ratio to be applied to the cross hair.

“**Y_Size_Ratio**” (inside the script) is “**25.0/1080.0**”. This means on a 1080p screen, I want the size of the crosshair to be 25 pixels wide. So lets say that the viewport size is not 1080p but instead is 1024x768. How will this work? Well, “**Y_Size_Ratio**” equals “**25.0/1080.0**”, which ultimately equals “**0.023148148**”. So, we take that number and multiply the X-axis size of the viewport to get our crosshair size number: “**Crosshair_Size = int(0.023148148 * 1024) = 23**”. This way, we can have the size we want and not have to worry about the window size.

Now we have a problem. In order to have a perfectly centered crosshair it needs to have a even numbered size, and 23 is odd. So what do we do?

First, we must check to see if "**Crosshair_Size**" is odd in the first place. Remember that many times you need to check a variable or condition before you modify. If you weren't to check things you may cause an error or modify something that doesn't need to be modified.

We use the modulo operator for this, which looks like this "**%**". What this does is simply divide the first number against the second, and then returns the remainder. So in our code we have this:

```
	if(Crosshair_Size % 2 != 0):
```

What this does is this: It takes "**Crosshair_Size**", which is "**23"** in this case, and divides it by two. Then, it returns the remainder, which is "**1**". What this is basically telling us is that it isn't even, or else the remainder would be "**0**" because dividing an even number by 2 doesn't produce any remainders.

Since we've found out that our proposed crosshair size isn't even, we make it even by simply adding "**1**" to it. Easy!

After this, what we now do is simply set the size and position of both the crosshair and it's red circle child. We set the size by simply calling:

```
	set_size(Vector2(Crosshair_Size, Crosshair_Size))
```

For each node. Then, we set the position of these nodes to half their size, like this:

```
	set_position( Vector2( (Viewport_Size.x/2 - Crosshair_Size/2) , (Viewport_Size.y/2 - 
Crosshair_Size/2) ) )
```

The reason we set the position to the *center* of the node instead of to the upper right corner is because in Godot I set the anchor of the "**TextureRect"** to the center of whatever picture I use. Check the Godot manual for information on "**Control**" node anchors. In this case, I simply selected the crosshair node and clicked on the "**Layout**" menu inside main viewport, went down to "**Anchors only**" and selected "**Center**". You can also manually set the anchor of the node manually in the node inspector. It's simply called "**Anchor**" and it's in the range of "**0.0 – 1.0**".

We do this to the red circle node, as well.

DEBUG.gd

This script simply toggles the visibility of the debug nodes on or off. It's pretty straight forward. All it does is set the "**_unhandled_process()**" to true, and whenever the "**Player_ToggleDebug**" action is activated, it checks to see if the debugging information is visible or not, and it changes it to whatever it currently is not.

One note is that I use "**Input.is_action_just_pressed()**" instead of "**Input.is_action_pressed()**" because the first function only activates when you initially press the button/key, and the second function keeps activating for as long as the button or key is held. This way the debug information is toggled on or off 60 times a second when you just press it once.

Debug_Label.gd

This basically does the same thing as "**Crosshair.gd**", but just a different size and position.

First, in the "**_ready()**" function, it runs "**ViewportSizeChanged()**", and then it connects that to "**size_changed()**" inside the main viewport of the game, the one that is the senior-most node, so that it is activated whenever the game window changes size, either by going fullscreen or by dragging the sides of the window.

In "**ViewportSizeChanged()**", it sets the font size and the rectangle size of the label.

First it gets the size that I want the font to be relative to the screen.

```
Font_Size_Rel = 50.0/1080.0
```

So that makes "**Font_Size_Rel**" equal "**0.046296296**", or ~4.6% of the horizontal screen size. Then, we set the relative size of the label's rectangle to six times the width of the relative size and 3 times the height of the previous variable. That will end up being about "**(0.27, 0.14)**". So what that means is that the labels total rectangle size will be about 27% of the screen horizontally and 14% of the screen vertically. That will be the area that the text inside the label will be shown in.

After that we make a variable which will hold the size of the viewport. We use that to set the size of the font itself and then set the size of the label rectangle according to the variable we made above, which looks like this:

(continued on next page)

```
get_font("font").set("size", Viewport_Size.y*Font_Size_Rel)
rect_size = Vector2( Viewport_Size.x * Font_RectSize_Rel.x   ,   Viewport_Size.y * Font_RectSize_Rel.y )
```

Chapter 4: Design of Base.tscn Scene

Section 2: Player.gd Script

Chapter 1: Summary of Contents of Player.gd Script

This chapter will detail, in order from top of the script to the bottom, what each thing does. Global variables, functions, unhandled input, "**_ready()**", and the main loop will be explained in further detail in their own chapters for easier consumption.

Global Variables and Constants

The top of the file, before the functions, contains notes, global variables, and constants, some of which are settings. The ones that are not don't need to be edited, as they are just keep variables needed for functioning, such as "**SlideCount**" inside the "**MOVEMENT**" section, which just holds is result of "**get_slide_count()**".

For more detail check the "Global Variables and Constants" chapter.

- Explanation of script and reference to this manual and the FPC GitHub project page.
- Notes on how to use the script correctly.
- The "**extends**" that extends the current script's functionality.
- Settings for things like the player's movement speed, mouse look toggling, etc.
- Definition of the crest factor in a handy constant.
- Signals for debugging.
- Variables that hold pointers to nodes.
- Variables that hold states, such as if the player is on a floor and what not.
- Global variables that don't really have any other place to be.
- Input bools and the strings that identify them by their action names set in the project's settings.
- Movement variables like the final walk velocity.
- Rotation variables like the relative movement of the cursor.
- Variables pertaining to the character's falling, such as the bool which states if the player is falling or not.
- Variables pertaining to jumping, such as the jump velocity.
- Variables pertaining to the speed of the character when moving on a slope, such as the slopes normal before being normalized into a 2D vector.
- Variables pertaining to stepping up platforms and steps, such as the distance the player must move up to get on the step.

- Variables pertaining to the camera smooth movement when walking up stairs and steps, such as how much time has elapsed since the interpolation started.
- Variables pertaining to ray casting, which is done several times through the script, such as the position that the ray cast should be shot from.
- Variables pertaining to interaction via the “Use” button or touch, such as the array that holds all the touched objects that had a “touch” function in their attached script.

Functions

The custom functions for this script, to make things a little easier to understand and read. These will be explained further in-depth in the “Functions Explained In-Depth” section.

- **InterpolateCamera()**
  - Interpolates the camera position when moving up a step or staircase for smooth camera movement going up them.
- **Step_Player_Up()**
  - Checks to see if there is a step to be stepped upon, and moves the character up if there is.
- **Touch_CheckAndExecute()**
  - Checks for any “touch” functions inside the currently colliding objects, and activates them if there are.
- **Slope_AffectSpeed()**
  - Checks to see if the player is on a slope of some kind and affects speed accordingly.

Ready

Initializes several things. The general overview:

- Starts the processes.
- Prepares states.
- Gets and sets the player’s position and the max height of the step, if too high.
- Set’s some defaults.
- Sets size of the “touched objects” array.

Unhandled Input

Takes care of camera and player rotation using the mouse. The general overview:

- Checks if mouse look is enabled.
  - Check if input is a mouse motion.
    - Get the relative mouse motion in comparison to where it was the last frame.
    - Set limits to how high or low the player can look.

        - Apply the rotation.

Physics Process

Moves the character around; jumps, steps up, applies gravity, etc.

The code overview:

- Check for input.
- Set states according to input.
- Check if the player is looking at something usable and react accordingly.
- Calculate horizontal movement.
- Calculate vertical movement according to if the player is:
    - On a floor.
    - Jumping.
    - Falling
    - On a slope.
- Apply movement using “**move_and_slide()**”.
- Get states once again.
- Execute “**Step_Player_Up()**” function.
- Check for camera interpolation and react accordingly.
- Execute “**Touch_CheckAndExecute()**” function.
- Set the debug label text.

Chapter 2: Global Variables In-Depth

This section explains, in detail, what each variable does at the top of the file, and where they are used. It also shows the default values of each one. The reason for these global variables is I don't think it's optimal to keep creating and freeing variables inside code. Let's say the following code is inside "**_physics_process()**":

```
	var SomeVar = self.translation.y
```

If this variable is used over and over again inside this script, I like to just create it at the top of the file once and always have it. I'm not sure if the variable is freed from memory after a single loop of "**_physics_process()**" or after the script is dereferenced.

Explanation (at the Top of the Script)

This explains what the script is and references this manual as well as the wiki on the GitHub project page, which is just a copy of this manual. It also shows the URL of this project's GitHub page.

https://github.com/leiget/Godot_FPC_Base

Notes

Says how the script must be used for correct functionality. They are:

- **"The collision safety margin must be set to at least 0.01 for the player's kinematic body, or else things like walking up steps and such will not work correctly."**
  - **"You'll need to test it and find out what works whenever you change the size of the player's collision shape."**
  - I'm not sure why the player character's margin must be at least 0.01, but it does.
- **"This script is made with the intent that the player's collision shape is a capsule."**
  - If you use something else, you will need to change the code.
  - Also, a capsule shape is best because if you where to use a cube and tried to turn the character while in a confined space, the corners of the cube would go inside the objects that are confining you.
- **"Note that in Godot 3.0.2 the "move_and_slide()" function has 5 arguments, but in the latest GitHub version (as of March 07, 2018) it has 6, with the added argument being in the 3rd position and is "bool infinite_inertia=true". If this option is true, what it means is that no other object can rotate the character. If false, it can if enough force is applied."**
  - "**bool infinite_inertia=true"** means that the character can't be rotated, because the amount of inertia needed to do so is infinite.
  - Look up "Infinite Inertia Tensor" on the net or in a book, if you want. It's not explained in this document.

Extends

- **extends KinematicBody**

This extends the current script's functionality to include the named class type and all its ancestors.

For instance, the "**KinematicBody**" inheritance tree looks like this:

KinematicBody < PhysicsBody < CollisionObject < Spatial < Node < Object

As you can see, "**KinematicBody**" inherits all these classes. But what does that mean? It means you can use all these other classes' functions. For instance, the "**Spatial**" class has the funtion "**get_global_transform()**", which gets the global transform matrix of the current node.

When you type "**extend KinematicBody**", it *extends* the "**KinematicBody"** functionality to include all the inherited ancestors, and in my example you can therefore use "**get_global_transform()**", even though "**KinematicBody**" itself doesn't have that function.

Settings

This is the big part that has all the variables that can be changed, whether on-the-fly or before running the script, to fit the needs of the project. It's organized into several parts according to their use. Here is a breakdown, including default values:

Mouse Look

- **MouseLook = true**
  - This allows you to turn mouse look on or off, for whatever reason you need to.
  - Used in the "**_unhandled_input()**" function.
- **Cam_RotateSens = 0.25**
  - The sensitivity of the mouse movement applied to the character's rotation.
  - Used in the "**_unhandled_input()**" function.

Movement

- **BaseWalkVelocity = 10**
  - The walk velocity of the character, in m/s, when a movement action is pressed.
  - Used in:
    - The "**SETTINGS**" section to calculate "**Step_SafetyMargin**".
    - The "**MOVEMENT**" section in the top of file to calculate the initial "**FinalWalkVelocity**".
    - **The "_physics_process > INPUT > SPEED SHIFT" to calculate "FinalWalkVelocity" when the "Pressed_Shift" action is pressed or when it isn't.**
- **ShiftWalkVelocity_Multiplier = 2**

    - The amount to multiply the “**BaseWalkVelocity**” when the “Player_Shift” action is held, to make the character move faster.
    - Used in:
        - **“_physics_process > INPUT > SPEED SHIFT” to calculate “FinalWalkVelocity” when the “Pressed_Shift” action is pressed.**
- **MaxFloorAngleRad = 0.7**
    - This is the max floor angle the character is able to walk on without it being considered a wall.
    - Used in:
        - “**MaxFloorAngleNor_Y**” : See explanation below.
        - “**move_and_slide()**” in the “**_physics_process() > FINAL MOVEMENT APPLICATION**” section.
- **MaxFloorAngleNor_Y = cos(MaxFloorAngleRad)**
    - This is the max floor angle, but converted to what it is as a normal, so that the floor normal can be compared to this without having to constantly convert it to radians.
    - Used in:
        - “**Step_Player_Up()**”
            - To see if a slide collision is a wall.
            - To see if the result from the raycast downward is a floor.
        - “**Slope_AffectSpeed()**” : To compare the current slide normal to the maximum, to get a ratio of how slow the character should move relative to the “**BaseWalkVelocity**”.
        - In the “**_physics_process > VERTICAL MOVEMENT > FALLING**” section to see if a current slide collision is a floor or not.
- **FloorNormal = Vector3(0,1,0)**
    - The 3D vector which defines which was is “up”, relative to the player.
    - Used in:
        - “**_ready() > move_and_slide()**” to initialize the state of the character.
        - “**_physics_process() > FINAL MOVEMENT APPLICATION > move_and_slide()**” to move the character.
- **MaxSlides = 4**
    - This is the maximum number of slides to be calculated when the character is moved using “**move_and_slide()**”.
    - Used in:
        - “**Touch_CheckAndExecure()**” : To set the size of the “**SlideCollisions**” array.
        - “**_ready()**” : To set the size of the “**Touch_ObjectsTouched**” array.
        - “**move_and_slide()**” in the “**_physics_process() > FINAL MOVEMENT APPLICATION**” section.
- **SlopeStopMinVel = 0.05**

  - Used in “**move_and_slide()**” in the “**_physics_process > FINAL MOVEMENT APPLICATION**” section.
  - The maximum horizontal velocity the character can have on a slope and remain still.
  - That is, when a character is on a slope, and the slope isn’t very steep, the engine will check to see how far the character would move if gravity is applied, and thus slide down the slope. If the character were to only have a velocity of, say, **0.02** while on the slope and gravity is applied, then the engine will not move the character.
    - If the body is standing on a slope and the horizontal speed (relative to the floor’s speed) goes below “**SlopeStopMinVel”**, the body will stop completely.
  - However, this doesn’t seem to work. I’ve tried very high values and the character would still slide on slopes. This may change in the future (written on March 07, 2018).
  - Used in “**move_and_slide()**” in the “**_physics_process > FINAL MOVEMENT APPLICATION**” section.

Falling

- **Falling_Gravity = 9.8**
  - The gravity to be used on the character.
  - It is converted to a negative number later, when needed.
  - Used in:
    - The “**_physics_process() > VERTICAL MOVEMENT > JUMPING**” section:
      - To check if the player’s jumping velocity is above the gravity velocity before attempting a jump.
      - To subtract it from the jump velocity when the jump action is pressed.
        - That is, “**Falling_Gravity**” is subtracted from the jump’s velocity so the character’s jumping velocity is affected by gravity.
- **Falling_Speed_Multiplier_Default = 0.25**
  - The default value of the above variable.
  - 0.25 is a good starting place for this character script, making the final gravity when standing on a floor or slope 2.45 m/s. As stated above, a gravity of 2.45 m/s keeps the player from being pulled down too hard when on slopes, causing him to slide down it.
  - Used in:
    - “**Slope_AffectSpeed()**”, at the end, to be the “**from**” argument in the “**lerp()**” (linearly interpolate) function so that the falling speed of the character when on the floor is never below “**Falling_Speed_Multiplier_Default**”.
- **Falling_TerminalVel = 54**
  - The terminal velocity of the character is set here, in m/s
  - For Earth in average conditions, this is about 54 m/s.

    - Used in the “**_physics_process() > VERTICAL MOVEMENT > FALLING**” section to see if the player is falling at terminal velocity and to keep him from going past it.
- **Falling_TimeToHitTerminalVelSec = 14**
    - The time it takes the character to hit terminal velocity, in seconds.
    - For Earth in average conditions, this is about 14 seconds.
    - Used in the “**_physics_process() > VERTICAL MOVEMENT > FALLING**” section to set how fast the player hits terminal velocity.

Jumping

- **Jump_Vel_RelativeToGrav = 1.55**
    - The jump velocity relative to gravity.
        - This means if the force gravity is **9.8**, the initial vertical velocity of the character will be “**15.19**” when the jump button is pressed.
    - Used in “**_physics_process() > VERTICAL MOVEMENT > JUMPING**” section at top of script to calculate “**Jump_Vel**”.
- **Jump_Length = 0.75**
    - How long the jump will last until the character starts falling, in seconds.
    - Used in the “**_physics_process() > VERTICAL MOVEMENT > JUMPING**” section:
        - To calculate if the jump has finished.
        - To calculate the jump velocity as time passes, so as to smoothly taper off the jump as it reaches its peak.

Stepping Up

- **Step_MaxHeight = 0.5**
    - The maximum height, in meters, that the player can step up a platform.
    - This must be less than half the size of the player.
        - This is because when stepping on a step, a ray cast is shot down upon the collision position to see if there is anything in the way of the player stepping up the step. If this height is more than half the height of the character, it will be set to half the height of the character. This happens in the "**_ready()**" function.
    - Used in:
        - The “**Step_Player_Up()**” function to set the vertical position of the step raycast.
        - “**_ready()**” to check if “**Step_MaxHeight**” is more than half the players height and set it to that.
- **Step_SafetyMargin = 0.2/BaseWalkVelocity**
    - The additional amount that character has to move up when stepping up a step.

    - This helps keep the character from getting stuck when moving up and down steps/platforms because he can't get quite enough height to get over the step, and therefore slides back down off the edge of it.
    - It's set according to how fast the player is moving, so that the stepping of stairs is more correct.
    - Used in:
        - "**Step_Player_Up()**"; to add to the "**SteppingUp_SteppingDistance**" to make sure the player goes up far enough to get fully on the step.
        - The "**_physics_process > INPUT > SPEED SHIFT**" section to to set "**Step_SafetyMargin**" according to how fast the player character is moving.
- **Step_RaycastDistMultiplier = 1.1**
    - The amount to move the ray cast away from the player to help detect steps more accurately.
    - When the character hits a potential step, a ray is cast from above that position, level to the player's global origin. It shoots straight down, level to where the player's "feet" are, checking for a collision.
        - This variable is the multiplied distance from the player the ray should be cast. This is to ensure that the ray cast origin is completely over the step, and also to ensure tiny steps are ignored.
    - Used in the "**Step_Player_Up()**" function.

Camera Interpolation

- **CamInterpo_Length_Secs = BaseWalkVelocity / (125.0 * (BaseWalkVelocity/10.0))**
    - How long it takes to interpolate the camera, in seconds.
    - Dependant on speed of character.
    - The default length is 0.08 seconds, if BaseWalkVelocity is 10.
    - Used in:
        - "**InterpolateCamera()**" to see if the camera interpolation is over and to calculate how far the interpolation is, at the current frame.
        - The "**_physics_process() > INPUT > SPEED SHIFT**" section to calculate how long the camera interpolation should be when the player is pressing the shift key, or when he's not.

Slope Speed

- **Slope_EffectMultiplier_ClimbingUp = 0.2**
    - The multiplier that sets how much a slope affects the character's applied gravity when climbing up one.
    - Used in "**Slope_AffectSpeed()**" to say how much slopes affect walking speed when moving up one.

- **Slope_EffectMultiplier_ClimbingDown = 1.5**
  - The multiplier that sets how much a slope affects the character's applied gravity when climbing up one.
    - This is more than 1.0 because I want the player to "stick" to the slope when walking down it, so as to keep the character from raising off the platform and falling on it, over and over again.
    - It also gives a sense of gravity as a person tends to walk down slopes faster than up them.
  - Used in "**Slope_AffectSpeed()**" to say how much slopes affect walking speed when moving down one.

Use Action

- **Ray_UseDist = 2.0**
  - The distance (in meters) the use button ray can go; the ray that looks for things that can be used.
  - Used in "**_physics_process > INPUT > CROSSHAIR: USABLE ITEM AND USE BUTTON**" section.

Defines

This section is for any constant expression that isn't native to GDScript.

- **CrestFactor = 1.414213562373095**
  - Used to make sure the player character's velocity is correct. See the "Crest Factor Explained" section.
  - Used in:
    - The "**_physics_process() > HORIZONTAL MOVEMENT > ADD X AND Z AXIS**" section.
    - "**Slope_AffectSpeed()**" to get the correct horizontal velocity of the player.

Signals

See the respective sections for in-depth explanations of each signal and how to use them.

- **Render_Pos(Coll_Vec3)**
  - Signal for rendering any 3D position.
  - Arguments:
    - **Coll_Vec3**
      - Type: Vector3
      - Description: The 3D position that you want to represent.

    - Not used in the default script.
        - You can put this wherever you want to represent any position.
- **Coll_Sphere_Show(SlideNumber, Pos_Vec3)**
    - Signal for showing collision slides from "move_and_slide()".
    - Arguments:
        - **SlideNumber**
            - Type: int
            - Description: The slide number you want to represent.
        - **Pos_Vec3**
            - Type: Vector3
            - Description: The position of the slide number to be represented.
    - Not used in the default script.
    - Can be put anywhere a “**for Slide in range(SlideCount):”** exist.

Nodes

These are simply node pointers to make accessing them easier.

- **Node_Camera3D = get_node("Camera_Main")**
    - Simply the player’s 3D camera node representing the player’s vision.
    - Used in:
        - The “**CAMERA STEP SMOOTHING**” section at the top of the file to get the default local position of the camera.
        - In the “**InterpolateCamera()**” function to move the camera.
        - In the “**Step_Player_Up()**” function to get the global position of the camera before being moved up the step and to move it there after the whole player is moved up the step.
        - In “**_input()**” to get and set the vertical rotation of the camera.
        - In the “**_physics_process > INPUT > CROSSHAIR: USABLE ITEM AND USE BUTTON**” section to see if the player is pointing at anything within range that can be used.
- **Node_Crosshair_Useable = get_node("Camera2D/Crosshair/Crosshair_Useable")**
    - The red circle that shows up when the player is within reach of and looking at a usable object.
    - Used in the “**_physics_process > INPUT > CROSSHAIR: USABLE ITEM AND USE BUTTON**” section to show and hide itself.
- **Debug_Label = get_node("Camera2D/DEBUG/Debug_Label")**
    - The debug label in the player’s tree that shows whatever the programmer wants it to show.

- Used only once at the end of the script.

- **Debug_Label_String = "-------------------"**
  - The string to be printed in the debug label. Simply make the string whatever you want to be shown on the screen.
  - Example: "**Debug_Label_String = "Player Position = " + str(Player_Position)**"
  - Used, by default, at the end of the script to show the FPS. Not necessary.

States

These are state variables, that is bools that say whether a certain condition exist or not. These are here for convenience and for only accessing the GDscript's state getting function once. That is, "**is_on_floor()**" is only needed to be called after a "**move_and_slide()**" once using "**State_OnFloor = is_on_floor()**" instead of calling "**is_on_floor()**" every time you need to check if the player is on the floor.

- **State_OnFloor = false**
- **State_OnWalls = false**
- **State_OnCeiling = false**
  - Says whether the player character is hitting a wall, floor, or ceiling.
  - Used in several places that will be explained in-depth in their appropriate sections.
- **State_Falling = false**
  - Says if the player is falling.
  - Used in several places that will be explained in-depth in their appropriate sections.
- **State_Jumping = false**
  - Says if the player is in the middle of a jump.
  - After the player has reached his jump peak, this variable becomes "**false**" and the player starts falling.
  - Used in several places that will be explained in-depth in their appropriate sections.
- **State_Movement_Diagonal_Pressed = false**
  - Says if the player is pressing a forward/backward and left/right key at the same time.
  - Used in several places that will be explained in-depth in their appropriate sections.

Global Variables

These are simply variables that are used in several different sections of the code, instead of being used only in one section. Or they may be here as there is no other section that it could go.

- **Player_Position = Vector3(0,0,0)**
  - The global position of the player character's origin.
  - Used in several places that will be explained in-depth in their appropriate sections.

- **Player_Height = (get_node("CollisionShape").shape.height + get_node("CollisionShape").shape.radius * 2)**
    - The total height of the player.
    - The complete line is this:
        - “**onready var Player_Height = (get_node("CollisionShape").shape.height + get_node("CollisionShape").shape.radius * 2)**”
    - The reason it has “**onready**” is because the nodes that are referenced may only be accessed after they are loaded (for obvious reasons). So this part of the code is only executed when the entire scene is done loading and is “ready.”
    - Used in several places that will be explained in-depth in their appropriate sections.
- **Player_GlobalFeetPos_Y = Player_Position.y - Player_Height/2**
    - The global position of the very bottom part of the player character’s collision shape.
    - Simply called “feet position” for simplicity.
    - Used in:
        - The “**Step_Player_Up()**” function to set and get the position of the bottom of the player’s collision shape.
        - “**_ready()**”; it’s initialized here.

Input

Input bools and action strings (the ones that are set in the “Project Settings/Input Map” tab, at the top).

- **Pressed_FW = false**
- **Pressed_BW = false**
- **Pressed_LEFT = false**
- **Pressed_RIGHT = false**
- **Pressed_Jump = false**
- **Pressed_Shift = false**
    - The bools that say if a action is pressed/active.
    - Used in:
        - “**_physics_process() > INPUT > MOVEMENT**” to receive the action states.
        - “**_physics_process() > HORIZONTAL MOVEMENT**” to calculate movement velocity.
- **String_FW = "Player_FW"**
- **String_BW = "Player_BW"**
- **String_Left = "Player_Left"**
- **String_Right = "Player_Right"**

- **String_Jump = “Player_Jump”**
- **String_Use = “Player_Use”**
- **String_Shift = “Player_Shift”**
  - The strings that are the names of the actions in the project settings.
  - I.e: “**ui_up**” or “**Player_FW**”.
  - Used in:
    - “**_physics_process() > INPUT > MOVEMENT**” to get action states according to name.

Movement

Variables concerning the horizontal movement of the player lay here.

- **SlideCount = 0**
  - This holds the slide count. Filled with whatever “**get_slide_count()**” returns.
  - Used so “**get_slide_count()**” doesn’t have to be called over and over.
  - Used in several places that will be explained in-depth in their appropriate sections.
- **DirectionInNormalVec3_FWAndBW = Vector3(0,0,0)**
- **DirectionInNormalVec3_LeftAndRight = Vector3(0,0,0)**
  - The 3D vectors that say which way the player is facing in normalized vectors.
  - See the “Normals Explained” section.
  - Used in:
    - “**_physics_process() > GET ROTATION-DIRECTION NORMALS**” to get the direction the player is facing in normals.
    - “**_physics_process() > HORIZONTAL MOVEMENT > FORWARD AND BACKWARDS CALCULATIONS”** to set the direction the player should move.
- **TempMoveVel_FWAndBW = Vector3(0,0,0)**
- **TempMoveVel_LeftAndRight = Vector3(0,0,0)**
  - The calculated horizontal movement velocity of the character before being added together to form “**FinalMoveVel**”.
  - Used in:
    - “**_physics_process() > HORIZONTAL MOVEMENT”** to receive the temporary velocity of the player in the variables receptive direction. That is, “**TempMoveVel_FWAndBW**” holds the velocity of the player only in the forward and backwards direction, and will be added to “**TempMoveVel_LeftAndRight**” in the “**_physics_process() > HORIZONTAL MOVEMENT > ADD X AND Z AXIS**” section .
    - “**Slope_AffectSpeed()**” to find out if the player is moving up, down, or perpendicular to the slopes normal.

- **FinalWalkVelocity = BaseWalkVelocity**
  - The final walk velocity applied to the horizontal movement of the player.
  - This is used because when the "**Player_Shift**" action isn't pressed, the walk velocity needs to be "**BaseWalkVelocity**", that is, the player needs to move at his normal speed. But if the "**Player_Shift**" action is pressed, the speed of the player needs to be "**BaseWalkVelocity * ShiftWalkVelocity_Multiplier**".
  - This changed on-the-fly to cause effects such as walking slower through water or swamps, if need be.
  - Used in several places that will be explained in-depth in their appropriate sections.
- **FinalMoveVel = Vector3(0,0,0)**
  - The final velocity to be applied to the player in "**move_and_slide()**" in the "**_physics_process() > FINAL MOVEMENT APPLICATION**" section.
  - Used in several places that will be explained in-depth in their appropriate sections.

Rotation

For rotating the player and his camera.

- **Mouse_Rel_Movement = Vector2(0, 0)**
  - The amount of pixels the mouse has been moved in the last frame.
  - Used in "**_input()**" to rotate the character.
- **Cam_Local_Rot_X = 0**
  - The camera's local rotation on its X-axis, which is up and down.
  - Used in "**_input()**" to rotate the character.
- **Final_Cam_Rot_Local_X = 0**
  - The amount to be applied to the camera's local X-axis.
  - Used in "**_input()**" to rotate the character.
- **Cam_Temp_XRot_Var = 0**
  - The amount the camera has moved, vertically, before being limited.
  - That is, if the player is looking straight up, and then moves his mouse up even farther, this variable will hold the angle that the camera would be before being limited to 90° up or down.
  - Used in "**_input()**" to rotate the character.

Falling

Variables used when the player is falling.

Note: The player is technically considered to be "falling" when standing still on a floor. The "**Falling_Speed_Multiplier**" is set to "**Falling_Speed_Multiplier_Default**" when standing still on a floor. By default, that would mean that "**Falling_Speed_Multiplier = 0.25**", and that "**FinalMoveVel.y**

= -2.45", if "**Falling_Gravity = 9.8"**. This is so because if the vertical velocity is -9.8 when standing on a slope, the character will slide down it because of the force of gravity.

- **Falling_IsFalling = false**
  - States if the player is falling.
  - Used in several places that will be explained in-depth in their appropriate sections.
- **Falling_Speed = 0**
  - The falling speed that is actually used to move the player.
  - Used in several places that will be explained in-depth in their appropriate sections.
    - Notably, it is used in the "**_physics_process() > VERTICAL MOVEMENT > FALLING**" section to calculate how fast the player is currently falling, in relation to how long he has been falling.
- **Falling_Speed_Multiplier = 0.0**
  - The multiplier to modify the falling speed of the character.
  - This is used when the character is moving on a slope to affect movement speed.
  - This is also used when the character is standing on the slope. When the character is on one, it is set to "**Falling_Speed_Multiplier_Default**" to keep the character from sliding down it because gravity is pulling him down too hard.
  - Used in:
    - "**Slope_AffectSpeed()**", at the end, to modify how fast or slow the character moves on an incline.
    - The "**_physics_process() > VERTICAL MOVEMENT > FALLING**" section to set the falling speed of the player to the default.
      - That is it sets "**Falling_Speed_Multiplier**" to "**Falling_Speed_Multiplier_Default**" when just standing still on a floor.
- **Falling_CurrentTime = 0**
  - The amount of time, in seconds, the player has been falling.
  - Used to calculate the vertical velocity.
  - Used in several places that will be explained in-depth in their appropriate sections.

Jumping

Variables for when the player is jumping.

- **Jump_Vel = Falling_Gravity * Jump_Vel_RelativeToGrav**
  - This value is the initial value of "Jump_CurrentVel" after the jump button is pressed.
  - Used in the "**_physics_process() > VERTICAL MOVEMENT > JUMPING**" section for several things. Will be detailed in the appropriate section of this manual.
- **Jump_CurrentVel = 0.0**

    - The current velocity of the jump, which is fed into “**FinalMoveVel**”.
    - This tapers off, over the length of the jump, until the peak of the jump has been reached. When that happens, the player is no longer considered to be jumping (“**State_Jumping=false**”), and the player starts falling (“**State_Falling=true**”).
    - Used in:
        - The “**_physics_process() > VERTICAL MOVEMENT > ON FLOOR**” to set the jump velocity to “**0**” after landing from a jump.
        - The “**_physics_process() > VERTICAL MOVEMENT > JUMPING**” section for several things. Will be detailed in the appropriate section of this manual.
- **Jump_CurrentTime = 0.0**
    - The current time of the jump, in seconds.
    - This is relative to when the jump started.
    - Used in:
        - The “**_physics_process() > VERTICAL MOVEMENT > JUMP PRESSED**” section to set the current time of the jump to “**0**” when the jump key is pressed.
        - The “**_physics_process() > VERTICAL MOVEMENT > JUMPING**” section to check if the jump is over, to calculate the jump velocity, and to progress the jump time according to the “**_physics_process()**” delta.
- **Jump_Released = true**
    - This says whether the jump action has been released or not.
    - I could of used “**is_action_just_pressed(String_Jump)**” to just get the start of the player’s pressing the jump action, but I only figured that out after I was done programming the jumping.
        - So I left this here in case someone wants to implement dynamic jumping, to where the player can alter the length of the jump by holding down the jump key longer.
    - Used in several places to see that will be explained in the appropriate section.

Slope Speed

Variables concerning the slowing down or speeding up of the player while on an incline.

- **Slope_PlayerVelVec2D = Vector2(0.0, 0.0)**
    - The player horizontal velocity.
    - Used in:
        - “**Step_Player_Up()**” to find out if the player is more than 11.5 degrees perpendicular to the step before attempting to move him up it. Explained in the “**Step_Player_Up()**” section.
        - “**Slope_AffectSpeed()**” to compare it to the slopes normal to see if the character is moving up, down, or sideways on a slope.
- **Slope_FloorNor2D = Vector2(0.0, 0.0)**

  - The floor’s normal on the horizontal normals.
  - To be compared to the player’s horizontal velocity.
  - Used in “**Slope_AffectSpeed()**” to be compared against the players normal, above, to see if the character is moving up, down, or sideways on a slope. Explained in the appropriate section.
- **Slope_Magnitude = 0.0**
  - The magnitude of the floor’s horizontal normals.
  - Used in “**Slope_AffectSpeed()**” to normalize the normal’s 3D vector into a 2D vector. Explained in the appropriate section.
- **Slope_MagnitudeRatio = 0.0**
  - The amount to divide “**FloorNor2D**” by in order to get a normalized 2D vector.
  - Check the “Slope Speed” section for more information.
  - Used in “**Slope_AffectSpeed()**” to normalize the normal’s 3D vector into a 2D vector. Explained in the appropriate section.
- **Slope_DotProduct = 0.0**
  - The dot product of the player’s horizontal velocity against the slope’s horizontal normal.
  - Used in “**Slope_AffectSpeed()**” to see if the character is moving up, down, or sideways on a slope and to calculate his velocity appropriately.

Steps

Variables concerning the stepping up of the player upon steps and raised platforms.

- **SteppingUp_SteppingDistance = 0.0**
  - The variable that says how high to move the character up to be able to get on the step.
  - Used in “**Step_Player_Up()**” to say how far to move the player up a step and how far down to locally move the camera down after moving the whole player character up.
- **Step_CollPos = Vector3(0.0, 0.0, 0.0)**
  - The global position of the collision of the player against a wall.
  - This is not the ray cast collision position. It is the collision detection’s slide collision of the player character’s kinematic body.
  - Used in “**Step_Player_Up()**” to cast rays towards and upon the step.
- **Step_CollPos_RelToPlayer = Vector3(0.0, 0.0, 0.0)**
  - The position of the collision of the step/wall relative to the player.
  - Used in “**Step_Player_Up()**” to move the ray cast away from the player a little to make sure it actually hits the step and doesn’t miss it by a little bit.
- **Step_Cam_PosBefore_Global = 0.0**
  - The global position of the player’s camera before he was moved up a step.
  - This is used for camera interpolation.

    - I put it here because it is modified in the "Step_Player_Up()" function.
    - Used in “**Step_Player_Up()**” to set the global position of the player’s camera before moving up the step.

These variables, below, are used for keeping the player from trying to move up a step if he is too parallel to it, causing him to step up and slide back down repeatedly because he is not able to fully get on the step, but only on the edge. Check the “Stepping Up” section for more information.

- **Step_PlayerVel_Global_Norm = Vector3(0,0,0)**
    - The global velocity of the player, normalized.
    - Basically, it just says what direction the player is moving (not the direction he is facing).
    - Used in “**Step_Player_Up()**” to find the angle between the step’s normal and the player’s velocity.
- **Step_CollPos_Global_RelToPlayer = Vector3(0,0,0)**
    - The step’s collision position, relative to the player, but in global space.
    - Used to find the angle between the player’s velocity and the step to see if they are perpendicular enough to allow stepping.
    - Used in “**Step_Player_Up()**” to find the angle between the step’s normal and the player’s velocity.
- **Step_CollPos_AngleToPlayer = 0.0**
    - The final angle of the player against the step collision.
    - Used in “**Step_Player_Up()**” to keep the player from trying to walk up a step that he’s walking too parallel to.

Camera Step Smoothing

Variables concerning the smoothing (interpolation) of the camera when the player walks up steps on moves up on a platform.

- **CamInterpo_DoInterpolation = false**
    - Tells script whether or not to do camera interpolation.
    - Initialized in the “**Step_Player_Up()**” function.
    - Used in:
        - “**InterpolateCamera()**” to turn off interpolation after it is done.
        - “**Step_Player_Up()**” to turn on interpolation after a step.
        - The “**_physics_process() > FINAL MOVEMENT APPLICATION > CAMERA INTERPOLATION**” section to check if camera interpolation should be done.
- **onready var CamInterpo_DefaultPosition_Local_Y = Node_Camera3D.get_transform().origin.y**
    - The default local position of the camera.

    - This value is basically whatever the camera's location is inside the "player.tscn" scene. It's relative to the player's origin.
    - This is what the camera will interpolate to when stepping up. To change this, simply change the vertical position of the player's camera inside "player.tscn".
    - Used in "**InterpolateCamera()**" to say where to interpolate to.
- **CamInterpo_StartingPos_Local_Y = 0.0**
    - The camera position, in local space, where it would be if it had not moved up with the player when encountering a step.
    - Used in:
        - "**Step_Player_Up()**" to set itself according to where it would be if it had not moved up with the player when encountering a step.
        - The "**_physics_process() > FINAL MOVEMENT APPLICATION > CAMERA INTERPOLATION**" as the first argument in the "**InterpolateCamera()**" function.
- **CamInterpo_CurrentTime_Secs = 0.0**
    - The current time of the camera interpolation that is in progress, in seconds.
    - Used in:
        - "**Step_Player_Up()**" to set the time of the interpolation to "**0**" after the player has gone up a step.
        - The "**_physics_process() > FINAL MOVEMENT APPLICATION > CAMERA INTERPOLATION**" section as the keeper of the return value of "**InterpolateCamera()**" and also as its second argument.

Ray Casting

Variables concerning the casting of rays.

- **Ray_SpaceState = null**
    - The state of the physical space in the game.
    - This holds all the information about collisions, collision boxes, where they are located, how big they are, their shape, and anything else pertaining to collision detection.
    - Used in several places.
- **Ray_From = Vector3(0.0, 0.0 ,0.0)**
    - The position that the ray cast will come from.
    - Used in several places.
- **Ray_To = Vector3(0.0, 0.0 ,0.0)**
    - The position that the ray cast will go to.
    - Used in several places.
- **Ray_Result = null**

    - The resulting dictionary of information received from “**Ray_SpaceState.intersect_ray()**”, which holds information like the collider, the ray cast collision position, or if the ray cast even hit anything in the first place.
    - Used in several places.

Interaction

Variables concerning interaction via touch and the “Use” action.

- **Use_Ray_IntersectPos = Vector3(0,0,0)**
    - The “Use” action’s ray cast collision (aka intersection) position.
    - Used in:
        - The “**_physics_process() > INPUT > CROSSHAIR: USABLE ITEM AND USE BUTTON**” section to set the position of the ray intersection.
        - In “Sphere.gd” to set the position of the applied impulse.
- **Touch_ObjectsTouched = []**
    - The list of objects that have been touched and that also contain a “Touched_Function()” method (another word for function).
    - Used in:
        - “**Touch_CheckAndExecute()**” to keep track to objects that have been touched that have a “Touched_Function()” function.
        - “**_ready()**” to set its size.
- **SlideCollisions = []**
    - The list of slide collisions to be compared against the “**Touch_ObjectsTouched[]**” array.
    - Used in “**Touch_CheckAndExecute()**” to keep track of the frame’s current slide collisions.

Chapter 3: Functions Explained In-Depth

This section explains, in detail, how each function is used and how they work.

Quick Reference of the Functions

Here is a quick reference of the functions and what they do, in case you need it.

- **InterpolateCamera(Prev_Pos_Local_Y, Time_Current, Time_Delta)**
  - InterpolateCamera(float, float, float)
  - Description: Interpolates the camera position when moving up a step or staircase for smooth camera movement going up them.
  - Arguments:
    - **Prev_Pos_Local_Y**
      - Type: float
      - Description: The previous local position of the camera before the player was moved up the step.
    - **Time_Current**
      - Type: float
      - Description: The current time of the camera interpolation.
    - **Time_Delta**
      - Type: float
      - Description: The delta, in seconds, from the last frame.
- **Step_Player_Up()**
  - Step_Player_Up(void)
  - Description: Checks to see if there is a step to be stepped upon, and moves the character up if there is.
- **Touch_CheckAndExecute()**
  - Touch_CheckAndExecute(void)
  - Description: Checks for any “touch” functions inside the currently colliding objects, and activates them if there are.
- **Slope_AffectSpeed()**
  - Slope_AffectSpeed(void)
  - Description: Checks to see if the player is on a slope of some kind and affects speed accordingly.