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

By Lei-Get Morzmorality (James Russell)

Table of Contents

Section 1:Project Overview....3
Chapter 1:Quick Intro....3
Chapter 2:Structure of Project....4
Project Directory Structure....4
Godot Project Editor Structure....6
Input Map....6
Extra Information....7
Chapter 3:Design of Main.tscn....8
Scripts Inside Main.tscn....9
Debug Scripts....10
Chapter 4:Design of Player.tscn Scene....12
Player Scene Overview....12
Design of Player.tscn....13
Scripts Inside Player.tscn....14
Chapter 5:Design of Base.tscn Scene....18
Overview of Base.tscn Geometry....18
Scripted Child Objects of Base.tscn (inside Main.tscn)....20
Chapter 6:Debug Scenes....22
Coll_Sphere.tscn....22
Pos_Visual_Axis.tscn....22
Section 2:Player.gd Script....23
Chapter 1:Summary of Contents of Player.gd Script....23
Global Variables and Constants....23
Functions....24
Ready....24
Unhandled Input....24
Physics Process....25
Chapter 2:Global Variables In-Depth....26
Explanation (at the Top of the Script)....26
Notes....26
Extends....27
Settings....27
Signals....32
Nodes....32
States....33
Global Variables....34
Input....34
Movement....35
Rotation....36
Falling....37
Jumping....38
Slope Speed....39
Steps....39
Camera Step Smoothing....41
Ray Casting....42
Interaction....43
Chapter 3:Functions Explained In-Depth....44
Quick Reference of the Functions....44
Step_Player_Up()....46
InterpolateCamera()....61
Slope_AffectSpeed()....64
Touch_CheckAndExecute()....69
Chapter 4:ready()....74
Chapter 5:_unhandled_input()....76
Chapter 6:_physics_process()....79
Section 3:Math Terms and Functions Explained 101
Chapter 1:lerp()....101
Chapter 2:pow()....102
Section 4:Normals Explained....107
Chapter 1:Coordinate System....108
Chapter 2:Vectors....117
Chapter 3: Transforms....122
Chapter 4:Normalization....127
Pythagorean Theorem....127
Chapter 5:Dot Product....130
3D Dot Product Formula....133
Understanding the Dot Product....134

Click a section to go to it.

Section 1: Project Overview

Chapter 1: Quick Intro

This project, called "Godot FPC Base", is a Godot 3.0.x project that can be used as a base for creating a first-person controller for a video game, architectural visualization, or more.

This manual provides an in-depth explanation of how this project works and its structure. It's not terribly complicated, but nevertheless I've covered it from top to bottom for newcomers.

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. The 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 collision shapes.
    - "**invisible_walls.shape**"
      - 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.
    - "**Shape_Capsule.shape**"
      - The capsule collision shape for the player.
  - **dae/**
    - The exported dae files for level geometry and other things.
  - **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 assets 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 Editor 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 “Debug Scenes” section.

Chapter 3: Design of Main.tscn

This chapter explains the main scene ("**Main.tscn**") and its scripts. Each item has the name first and its node type in parentheses.

- **Main (Spatial)**
  - "**Init_Script.gd**" is attached to this node.
  - **Env_and_Lights (Spatial)**
    - **WorldEnvironment (WorldEnvironment)**
      - Uses the "**Main_Env.tres**" file in the "**environment**" folder.
    - **Sun (DirectionalLight)**
      - Just the main light.
  - **Base (Spatial)**
    - A instance of "**Base.tscn**".
    - **Platform_Low (KinematicBody)**
      - "**elevator.gd**" is attached to this node.
      - Has a "**AnimationPlayer**" node which has a "**Lift**" animation.
    - **Platform_High (KinematicBody)**
      - "**elevator.gd**" is attached to this node.
      - Has a "**AnimationPlayer**" node which has a "**Lift**" animation.
    - **Big_Sphere (RigidBody)**
      - "**Sphere.gd**" is attached to this node.
    - **Blue_Sphere (RigidBody)**
      - "**Sphere.gd**" is attached to this node.
    - **ZipBox (KinematicBody)**
      - "**elevator.gd**" is attached to this node.
      - Has a "**AnimationPlayer**" node which has a "**Lift**" animation.
    - **FloorPlatform (KinematicBody)**
      - "**elevator.gd**" is attached to this node.
      - Has a "**AnimationPlayer**" node which has a "**Lift**" animation.
    - **SmallStep_01 (StaticBody)**
    - **Smallstep_02 (StaticBody)**
    - **Smallstep_03 (StaticBody)**
    - **BigStairs (StaticBody)**
    - **BigStairsPlatform (StaticBody)**
  - **GIProbe (GIProbe)**
  - **Player (KinematicBody)**
    - And instance of "**Player.tscn**".

Scripts Inside Main.tscn

There is only one not explained elsewhere.

Init_Script.gd

There is only one global variable in this scene: a bool that says if we want to have collision slide visualizations. If you set this to true make sure you also put in "**emit_signal("Coll_Sphere_Show", slide, get_slide_collision(slide).position)**" somewhere in "**Player.gd**". Check the bottom of this script for a little more information.

After that we have a "**_ready()**" function. It sets the window title, mouse mode, and unhandled input process.

Then we have a "if" statement checking if we want to have collision spheres. If we do then it sets up a variable for pointing to the player node, a collision sphere array to hold the instance pointers to the collision spheres, and an integer that holds whatever "**MaxSlides**" is inside "**Player.gd**". After that it resizes the collision sphere array to whatever "**MaxSlides**" is. By default that's four, so our array would be four elements long. We also have a variable that holds the collision sphere scene so that we can instance it however many times we need.

After this there is a loop that creates instances of the collision sphere scene ("**Coll_Sphere.tscn**") according to the amount of slides there are in the player's script.

In this loop it first creates the instance that we need. Whenever we "instance" a scene, what we are doing is basically just copying it and making a new, independent object of whatever is in that scene. As I said it's completely independent and has it's own variables, settings, and whatever else.

The reason that is important is because our next step in this loop is to set the "**Array_Element_Number**" variable inside the "**Coll_Sphere.gd**" script that is linked to the instanced scene. Each when setting these variables, each scene has it's own "**Array_Element_Number**" variable. It doesn't copy over from one sphere to the next. We set the number of "**Array_Element_Number**" according to whatever iteration the loop is currently in. By default, the variables would number from 0 to 4.

After this, we add the instanced sphere to be a child of the current scene, which would be "**Main.tscn**" in this case, as it is the one that has this script attached to it. Here's the code:

```
get_tree().get_root().get_child(0).call_deferred("add_child", Coll_Sphere_Array[x])
```

Click here to go to the online Godot manual to see how scene trees work.

If that link doesn't work, then look in the Godot manual for "**Scene Tree**". If that doesn't work, then look for "**get_tree()**" or "**get_root()**" and go from there.

One important aspect of this line of code is the "**called_deferred()**" function. What this does is simply defer the execution of the function named in the first argument to until the current scene is completely loaded, so as to keep the engine from trying to load something that isn't ready yet.

The first argument of "**call_deferred()**" is the function you want to execute when everything is done loading, which in this case is "**add_child()**" without the parentheses. It must be a string. The

second argument is the what you want the first argument of "**add_child()**" to be. If you have another function with multiple arguments just add them in a comma separated list. An example:

```
call_deferred("SomeFunc", 1.2, SomeVar, 5.0)
```

After that, we finally connect the "**Coll_Sphere_Show**" signal of the player's node to the "**Coll_Sphere_Show()**" function of the collision sphere's script. Make sure to emit that signal in "**Player.gd**" somewhere.

The following code, which is an initialization of a temporary variable and a for loop, just sets the initial y position of these sphere instances in 50cm increments above each other, so that the programmer can see them and make sure that things are initializing correctly. It's not necessary.

After the "**_ready()**" function we have an unhandled input function. All it does it quit the program when the player presses whatever the "**ui_cancel**" action is (the "Escape" key by default), and toggles the window into or out of fullscreen whenever the player presses the "**Toggle_Fullscreen**" action (F4 by default). Note that "**is_action_just_pressed()**" only gets input *once* when the button is initially pressed. This works well here as we only want the window to change whenever the player presses the key, but not when it is held down, as that would cause the window to toggle between fullscreen and windowed mode way too much.

Lastly, at the bottom we have the example code you can put into "**Player.gd**" to show the collision spheres.

Debug Scripts

Here I explain how the two debug scripts work.

Coll_Sphere.gd

This is to be used with an "**ImmediateGeometry**" node. At the top it has a single global variable called "**Array_Element_Number**" to be used to say what element index the current instance of this script's collision sphere inside the collision sphere array is. So if you have a "**MaxSlides**" of 4 inside the player's script, then the "**Array_Element_Number**" at the top of the file represents the slide number to be visualized. Also check the "**Init_Script.gd**" section on the previous page for more information.

"**_ready()**" makes a sphere and a line going through it. Check the Godot manual or online for tutorials and information about **ImmediateGeometry**.

The only other thing in here is a function called "**Coll_Sphere_Show()**" which sets the position of the sphere according to the position of the collision slide referenced in the player script.

What it does is take a slide number and a 3D vector. It checks to see if the current slide number (inside the player script) matches the "**Array_Element_Number**" set while being instanced in "**Init_Script.gd**". If they're the same it sets the position of the sphere to the current slide collision's position using an signal.

Pos_Visual_Axis.gd

This is also to be attached to an "**ImmediateGeometry**" node.

In the "**_ready()**" section it creates a line primitive that has three lines all intersecting each other in the center. After that it connects "**Set_Position()**" in this script to the "**Render_Pos**" signal in "**Player.gd**".

In the "**Set_Position()**" function it takes a 3D vector given when emitting the signal from "**Player.gd**", and simply sets the location of the geometry according to the vector given. That's it.

Chapter 4: 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**".
  - **Shape_Capsule (CollisionShape)**
    - The collision shape of the player.
    - It is a capsule primitive shape.
      - I've tried making a convex collision shape based on a cylinder, but it didn't work very well. The character would keep going through walls.
        - Here's a quote from the main programmer of Godot about this when someone asked for Cone/Pyramid/Cylinder primitive types for collision/physics bounds:
          - "**Pyramid is not a problem, but the others are not common at all on physics engines, given the SAT algorithm can't be used. They only exist in Bullet, which uses a different collision algorithm called GJK+EPA. I don't like this algorithm much because for games it's a little imprecise and requires users to set-up margins manually to all convex shapes. As such, this can't be done. Even PhysX does not support cylinders.**" - Juan Linietsky, January 2016.
        - So, basically, the algorithm used in the physics engine (I'm guessing Bullet, too, as this was written in January of 2016 and things may have changed since) best supports kinematic bodies with primitive shapes, as they fit better with the way the physics engine works. It's usually best to use whatever the engine gives you, in general.
  - **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.
  - Gets shown/hidden when the user presses F3.
  - **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.
  - **Instruction_Label**
    - Shows the player all the actions and what they do.

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 "**Shape_Capsule**" node is a capsule primitive 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 (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 5: Design of Base.tscn Scene

In this section I'll quickly explain how the "**Base.tscn**" scene is organized and also how all of it's children in "**Main.tscn**" works as well, such as the spheres that you can push around and the elevating platforms.

Overview of Base.tscn Geometry

Text Above Ramps

The text above the ramps are simply text objects from Blender converted to meshes. They tell the angle of the ramp below it in degrees.

Ramps With Varying Slopes

These are here for the testing of the attenuation of walking on slopes and walking on slopes that are considered walls ("**MaxFloorAngleRad**" in "**Player.gd**").

The Big Curved Ramp Inside the Hemisphere

This is for testing the movement of a somewhat low-res terrain. This is important when doing slope speed attenuation as on a ramp like this as one polygon on the ramp may have quite a bit different slope angle to the next polygon, causing a weird effect wherein the player randomly slows down and speeds up while walking up the ramp.

This may be mitigated by modifying the exponent of the "**pow()**" function at the end of "**Slope_AffectSpeed()**" function to give a smoother angle to speed falloff.

Big Hemisphere

It's just a big hemisphere that looks cool

Big Steps Inside Hemisphere

For testing jumping and walking in confined space that angle into a impassable space between them at one end. To see if there are collision errors.

Two Curved Walls

There are two curved walls, one on the floor and one in the air near the floating bridge. The one on the floor has a dark metal material.

This ramp is for testing walking up a slope that gradually curves into a wall and seeing if there are any errors. The only problem is that the character tries to walk up the slope that is too steep even if he can't walk on it. It isn't a huge problem though, and it's not unrealistic. If you were to try walking up a steep slope quickly you would get up a little ways and start sliding back down.

Floor Boxes

The floor is actually 4 boxes. This is for testing walking on a floor that is made of several elements. When passing over from one floor section to another there are no errors, so that's a good sign of well designed collision detection.

Group 01 Boxes and Objects in NE Corner

This group of boxes and objects is just for testing random geometry, stairs, a ramp, and a couple of tiny bridges between some of them. It's also for testing jumping in a confined space, as when you walk under the little bridges you can try to jump but you'll fall right back down.

Group 02 Boxes and Objects in NW Corner

This exist for the same reason that the objects and boxes in the NE corner do.

The big curved floating object is for testing how the player walks on in confined curved space. The only problem I've had is when the player in under the overhang near the end of the object. He tried to step up onto some of the geometry there, for some reason. It's not game-breaking, though.

High Curved Road

This is so that the an elevator that goes up can have a destination.

Tall Cubes on Edges of Map

These are here for testing the shadows.

Big 30 Degree Ramp on South End

This is for testing a big ramp. The big problem with this is that since the ramp is a little ways away from the invisible wall, the player character can fall between the ramp and the wall. When this happens, the player doesn't really fall between them as the space is only about 10 cm. Even so, when this happens the physics engine doesn't count the player as being on a floor, but instead as being on a couple of walls, not allowing the player to get back on the ramp. I might work on this, I might not. It seems important.

Invisible Walls

These are here to keep the player from falling off the sides and to also test pushing a quick moving object through them, such as the blue sphere with Godette trapped inside.

I've had a problem with the blue sphere being pushed through it at somewhat moderate velocities. Then I made the walls thicker and the problem persisted. I also tried using primitive collision shapes provided by Godot and they didn't work either. I was also enabling and disabling continuous collision detection on each example. I guess CCD doesn't work right now, as of Godot 3.0.2. Oh well.

Small Step Platforms

These are for testing platforms that have tiny steps on them and seeing if the character tries to walk on the tiny steps instead of the bigger step. The player actually does walk up the tiny step, but it doesn't cause any issues such as getting stuck on the tiny step or what not.

Scripted Child Objects of Base.tscn (inside Main.tscn)

Platform Low, High, ZipBox, and FloorPlatform

These objects have "**elevator.gd**" attached to them.

This script simply has one functions in it, "**TouchFunction()**".

It first checks to see if the animation player child node is playing any animation. If it's not then it plays the "Lift" animation of that node.

This is used by simply touching the node with the player character, and all the activation is taken care of in "**player.gd**".

Blue Sphere and Big Red Sphere

These objects have "**Sphere.gd**" attached to them. This script has just one function in it, "**UseFunction()**", which pushes the sphere around according to where on the sphere the player is looking at, and pushes with a force according to it's mass set in the node properties.

It first set's up two Vector3 variables, "**Pos**" and "**Impulse**". Then it set's up a floating point variable called "**Impulse_Mul**" and set's the impulse amount according to the object's mass as set in the node's settings. "**Pos**" is the position on the sphere that the impulse will be applied to. "**Impulse**" is the direction and magnitude of force of the applied impulse. "**Impulse**" is multiplied by "**Impulse_Mul**".

It then gets the player node and also the player's camera node and puts them into a variable.

Now it makes variables "**x**", "**y**", and "**z**" and puts the direction normal that the player is facing inside them. "**x**" and "**z**" say how much the *player node* is looking at the x and z axis and in what direction. "**y**" says how much the *camera node* is looking up and down, basically. If I tried to use the camera's normals I couldn't get the "**x**" axis is register as anything other than "**0**". I'm not sure why.

After this we set up a Vector3 variable that says in what direction the player is looking.

Here's the final part where we set everything up to do our "**apply_impulse()**" on the sphere. We set the "**pos**" variable according to the player's "**Use**" ray cast intersection position. Now here's the confusing part: we do this in global space but relative to the sphere. So we take the ray cast intersection point and subtract the sphere's global position from it to get the relative position of the ray intersection.

The picture show the basic reason why we use global space. If the we used local space and the object rotated, like the sphere, whenever we tried to “use” the sphere the point would be incorrect even if we converted it to global space using “**to_global()**”.

Now the impulse variable needs to be set. It just gets our previous Vector3 variable that holds the direction the player camera is looking and multiplies that by the force multiplier (“**Impulse_Mul**”).

Then we simply apply the impulse with “**apply_impulse()**”.

Chapter 6: Debug Scenes

Here are quick explanations of the debug scenes inside the "**scenes/DEBUG**" folder.

Coll_Sphere.tscn

This is just a single node, a "**ImmediateGeometry**" node with "**Coll_Sphere.gd**" attached to it. This scene is meant to be instanced through code. Be sure to emit the "**Coll_Sphere_Show**" signal inside "**Player.gd**" somewhere.

Check these links to sections of this manual for more information.

Init_Script.gd

Coll_Sphere.gd

Player.gd Signals

Pos_Visual_Axis.tscn

This is a single "**ImmediateGeometry**" node with "**Pos_Visual_Axis.gd**" attached to it. This scene is meant to be instanced inside the editor. You do this by simply instancing it to the root node and emitting the "**Render_Pos**" signal inside "**Player.gd**". You can attach it to whatever node you want, really. But you'll need to edit the line inside "**Pos_Visual_Axis.gd**" that connects the "**Set_Position()**" function to whatever node you need it to connect to.

Check these links for more information.

Pos_Visual_Axis.gd

Player.gd Signals

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.
- 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 each frame, 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, but it makes me feel better.

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 around 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 = 0.3**
  - The sensitivity of the mouse movement applied to the character’s rotation.
  - Used in the “**_unhandled_input()**” function.

Global Variables

- **Player_Height = 1.8**
  - The total height of the player.
  - Used in several places that will be explained in their appropriate sections.

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 > LINEAR MOVEMENT**" section.
- **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 > LINEAR MOVEMENT > 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 > LINEAR MOVEMENT**" section.
- **SlopeStopMinVel = 0.05**
  - Used in "**move_and_slide()**" in the "**_physics_process > FINAL MOVEMENT APPLICATION > LINEAR MOVEMENT**" 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 > LINEAR MOVEMENT**" 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_Dividend = 0.02**
  - The dividend for the step safety margin variable, below.
  - Used in:
    - "**Step_SafetyMargin**" variable right below this one.
    - "**_physics_process() > INPUT > SPEED SHIFT**" to alter the step safety margin based on the speed of the character.

- **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_Multiplicand = 125.0**
  - The multiplicand for the "CamInterpo_Length_Secs" variable below.
  - Lower this number to make the camera interpolation speed slower.
  - Used in:
    - "**CamInterpo_Length_Secs**" variable, below.
    - The "**_physics_process() > INPUT > SPEED SHIFT**" section to alter the speed of the camera interpolation based on the player's speed.

Slope Speed

- **Slope_EffectMultiplier_ClimbingUp = 0.3**
  - 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.

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_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.

- **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.**
- **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 > LINEAR MOVEMENT**” section.
  - Used in several places that will be explained in-depth in their appropriate sections.

Rotation

For rotating the player and his camera.

- **Mouse_Moved = false**
  - The bool that says to rotate the character because the mouse has been moved.

- **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_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.

- **Step_SafetyMargin = Step_SafetyMargin_Dividend/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.**

- **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_Length_Secs = BaseWalkVelocity / (CamInterpo_Length_Secs_Multiplicand * (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.
- **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 be the location to which it will be interpolated.

- **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. I'm going to go a bit out of order when explaining how the functions work. I'm first going to explain "**Step_Player_Up(),**" then "**InterpolateCamera()**", "**Touch_CheckAndExecute(),**" and "**Slope_AffectSpeed()**".

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.

Step_Player_Up()

This function simply steps the player up when encountering a step. This function should generally be used after a "**move_and_slide()**" is done so that the wall, floor, and ceiling states can be updated.

```
if(SlideCount > 1):
```

The first thing it does is see if there are more than one slide collisions. This way the function only executes when encountering more than one slide, like if the player is running against a wall that would constitute 2 to 3 slides.

```
if(State_OnFloor):
```

It then checks to see if the player is even on the floor. How can you step up something without being on the ground?

```
if(State_OnWalls):
```

Then it checks to see if the player is on a wall. This is way if there are multiple slides, like going up a ramp, it won't try to step up the ramp. There can also be multiple slides when simply walking on a level floor. Usually there are 2 slides when doing so. I'm not totally sure why this is.

```
for Slide in range(SlideCount):
```

If the player is on a wall, floor, and has more than 1 slide we start a "**for**" loop that goes through each slide.

```
if(get_slide_collision(Slide).normal.y <= MaxFloorAngleNor_Y):
```

It first checks to see if the slide collision in question is a wall by checking it against the max floor angle normal ("**MaxFloorAngleNor_Y**").

```
Step_CollPos = get_slide_collision(Slide).position
```

If it is, it gets the position of the collision in global space.

```
Player_GlobalFeetPos_Y = Player_Position.y - (Player_Height / 2) - get("collision/safe_margin")
```

Then it gets the global position of the player's "feet", the very bottom of the player character's collision shape. By default this is 0.9 meters below the player character's origin, since the default character height is 1.8 meters. But this "0.9" does not include the safety margin of the kinematic body, so we include that too.

```
if(Player_GlobalFeetPos_Y < Step_CollPos.y):
```

It then checks to see if the global position of the collision slide is even above the player's "feet" in the first place.

```
Step_CollPos_RelToPlayer = to_local(Step_CollPos)
```

If it is it gets the local position of the collision position by using "**to_local(Step_CollPos)**". Since this is called inside "**Player.gd**" that is attached to "**Player.tscn**", it returns the collision position in the player's local space.

```
Step_PlayerVel_Global_Norm = Vector3( Slope_PlayerVelVec2D.x ,
                                      0.0,
                                      Slope_PlayerVelVec2D.y )
```

It now gets the player's global horizontal velocity and puts it into a variable. This is so we can compare it between the step's angle and the player's velocity later.

What we want is the horizontal velocity of the player in a normalized vector.

The elements of this vector are from "**Slope_PlayerVelVec2D**", which is set inside "**Slope_AffectSpeed()**" whenever the player is walking on the ground, so there's no need to worry about errors due to "**Slope_PlayerVelVec2D**" not being set, as it will always be whenever the player moves on the ground.

Also, "**Slope_PlayerVelVec2D**" has already been normalized inside of "**Slope_AffectSpeed()**", so there is no need to do it here.

Note that it is a 3D vector, but the middle element (the Y axis) is "**0.0**". This is because we are going to be comparing the this against another 3D vector. More on that when we get there.

```
Step_CollPos_Global_RelToPlayer = Vector3(Step_CollPos.x, Player_Position.y,
Step_CollPos.z) - Player_Position
```

What we do here is get the relative position of the collision compared to the player's position *in global space*. We do this by simply subtracting the player's global position from the collision's global position.

The Y axis, or the second element, of this vector will always equal "**0**". This is because the second element of the first vector in this code *is* the player's y position. That's so we can just compare the horizontal axis positions without having to worry about the vertical position, which isn't important in this case.

I could have converted the X and Z axis of the collision and player positions into a 2D vector, but that would take extra work on the computer's part and so we are just going to compare two vectors in the upcoming code in which both vector's Y axis are "**0**".

```
Step_CollPos_Global_RelToPlayer = Step_CollPos_Global_RelToPlayer.normalized()
```

Now this just normalizes the collision position relative to the player, so we can use it in a dot product in the next line of code. Check the "Normals Explained: Normalization" section of this manual for information on vector normalization.

Let's recap what has happened so far, first by looking at this picture:

This view is from the top, looking down.

In this picture, we have the player moving north and a little bit to the east. Let's say that it's about "(**0.24 , 0.0, 0.96)**" (remember that it has been normalized). And then the collision position is exactly to the east of the player. So let's say that it is "**(1.0, 0.0, 0.0)**" (this is also normalized).

So now what we want to do is find out the angle between these two vectors. We'll find out how why we need it later.

```
        Step_CollPos_AngleToPlayer =
abs(  asin( Step_CollPos_Global_RelToPlayer.dot(Step_PlayerVel_Global_Norm) )  )
```

In this line of code, we have a lot going on.

The first thing we need is the dot product of the "**Step_CollPos_Global_RelToPlayer**" vector and the "**Step_PlayerVel_Global_Norm**" vector, so in this line of code we do:

```
        Step_CollPos_Global_RelToPlayer.dot(Step_PlayerVel_Global_Norm)
```

To save time, I'm going to tell you that the dot product is "**0.246385**". Now, what we want to do is find the arcsine ("**asin()**" in Godot) of that number to get the angle in radians between the direction that the player is walking and the direction of the collision position.

The This is trigonometry, so let's take this:

And look at it like this:

You see why we need trigonometry? In the end, vectors and relations between them are all just triangles, even in 3D. But we will just stick to 2D for now, as it is simpler and all we care about right now is the horizontal plane, anyway.

Like I said, we need the angle between the collision position and the player's velocity. We do this by using "**asin()**" on the dot product we just got. So the code looks like this:

```
asin( Step_CollPos_Global_RelToPlayer.dot(Step_PlayerVel_Global_Norm) )
```

And if we were to use real numbers:

```
asin( 0.246385 ) = 0.2489484 radians = 14.263692637 degrees
```

So now we see that the angle between the player's velocity vector and the collision position is "**1.015985294**" radians, which is about "**14.26**" degrees.

Okay, that all great, but what's the point? Let's look at the next line:

```
if(Step_CollPos_AngleToPlayer > 0.200712864):
```

It checks to see if the angle between then player's velocity vector and the step collision vector is less than about "**0.2007**" radians, or "**11.5**" degrees.

The reason is that when the player is running along a step within ~11.5 degrees, the script attempts to step the player up, but since the player is still moving almost parallel along the step, the player slides along the side of the step and isn't able to get completely on the step, and therefore falls back down.

And example picture:

Notice that X velocity of the player isn't very much? In fact, it's so little that it isn't able to move the player completely onto the step, and instead slides along the side of the step. And when the player is moved up in order for the player to get on the step, the player has already been slid on the side of the step, and is therefore not on the step itself when moving.

If the player is moving at *more* than a 11.5 degree angle, there is enough velocity of the player towards the step that he is able to move onto it.

It can be difficult to visualize what's going on in your code, but it's essential to practice doing so as it is a needed step in problem solving. If you have a hard time doing so I suggest putting your head down, eyes closed, in a quiet room and give yourself time to think about it.

The next lines of code are:

```
Step_CollPos.x = to_global(Step_CollPos_RelToPlayer * Step_RaycastDistMultiplier).x
Step_CollPos.y = Player_GlobalFeetPos_Y + Step_MaxHeight
Step_CollPos.z = to_global(Step_CollPos_RelToPlayer * Step_RaycastDistMultiplier).z
```

These lines of code push the collision position vector out a little bit so that it is over the step geometry instead of being on the edge, as shown here:

So if our relative collision position was something like "**(0.5 , 0.0 , 0.0)**" after multiplying it by "**Step_RaycastDistMultiplier**" (which is set in the "**SETTINGS**" section as "**1.1**") we get "**(0.55 , 0.0, 0.0)**". We convert it the X and Z axis of these lines to global space, which is needed for the ray cast, as it takes Vectors in global space.

Notice that we set the Y axis position of the collision position to be the vertical position of the player's "feet" plus the maximum height that a step can be, which by default is "**0.5**".

The reason we are doing all this is because we are going to use "**Step_CollPos**" as the starting point for several ray casts.

The first being the next few lines of code:

```
Ray_SpaceState = get_world().get_direct_space_state()
Ray_From = Vector3(Player_Position.x , Step_CollPos.y , Player_Position.z)
Ray_To = Step_CollPos
Ray_Result = Ray_SpaceState.intersect_ray(Ray_From,Ray_To,[self])
```

"**Ray_SpaceState**" get the state of the physics "world", that is it get the state of everything in the scene that has anything to do with physics, like collision shapes, locations, etc. We call the ray tracing function from this space state.

"**Ray_From**" sets the position that the ray cast is going to start from. In this case, it is going to be at the player's horizontal position, and the step collision's vertical position.

"**Ray_To**" sets the position that the ray is going to be cast to. It's going to be what we set it as, before.

"**Ray_Result**" cast the ray by using "**Ray_SpaceState.intersect_ray()**" and keeps the result of that ray cast, which we can look at in the next line of code.

So let's get a visualization of the ray cast to see what we are doing (on the next page):

So to start off with, "**Ray_To**" is the collision position's X and Z position. And it's Y position is the position of the player's "feet" plus "**0.5**", which is the maximum height that a step can be, according to the default "**Step_MaxHeight**" setting at the top of the script.

This ray cast is needed because imagine that there are two separate steps, with their own independent collision bodies, on top of each other. If the player were to walk into these steps, one of the slide collisions will be on top of the bottom step. I don't know why this is, that's just how the physics in Godot work. And the ray cast done above makes sure there isn't anything in the way of the player being able to step up onto the step being collided with. Here's a visualization:

The red dot represents a collision slide that would be used for stepping up the player. The ray cast we are currently talking about is represented here, and shows why it is necessary.

```
if(Ray_Result.empty()):
```

It first checks to see if the ray cast just done is empty. If it isn't, then nothing happens as the player isn't able to step up with something in the way, like it would be in the above picture. And if it isn't the last slide collision, it goes to the next slide.

But, if it *is* empty, we move on with the code.

```
Ray_From = Step_CollPos
Ray_To = Vector3( Step_CollPos.x , Player_Position.y + (Player_Height * 0.5) +
( to_global(Step_CollPos_RelToPlayer).y - Player_GlobalFeetPos_Y) , Step_CollPos.z )
Ray_Result = Ray_SpaceState.intersect_ray(Ray_From,Ray_To,[self])
```

Now we are going to setup and execute another ray cast. This time, "**Ray_From**" is simply the modified step collision position. "**Ray_To**" is the collision position, also. But, the vertical position is the player's origin position, plus half of the player's height, plus the height of the step. It then casts a ray. Here's a visualization:

The reason we need this ray cast is to check and see if there is a floating platform above the collision position that could block the player from being able to get onto it. Here's a second visualization (on the next page):

So what we are doing is casting a ray up to see if there is anything above the step that could obstruct the player from moving onto the step. In this picture, there is a floating platform in the way.

The reason that the ray cast goes so high is that it shoots all the way up to where the top of player's collision shape would be if he were on top of the step.

```
if(Ray_Result.empty()):
```

Then it checks to see if the ray cast hit anything. If it did, nothing happens, and if it isn't the last slide collision, it goes to the next slide. If it didn't then we move on.

```
Ray_From = Step_CollPos
Ray_To = Vector3(Step_CollPos.x , Player_GlobalFeetPos_Y , Step_CollPos.z)
Ray_Result = Ray_SpaceState.intersect_ray(Ray_From,Ray_To,[self])
```

We setup and execute one last ray cast. This ray cast shoots down onto the step.

"**Ray_From**" is the same as it was last time, being the modified step collision position. "**Ray_To**" is the step collision position, but with the Y position being equal with the player's "feet". Here's a visualization:

And from the side:

And then we have the next line of code:

```
if(not Ray_Result.empty()):
```

And yet again we check to see if the ray cast has a result. But this time we are checking to see if it has something in it instead of checking to see if it's empty. If it has a result then we continue on with our code. If it doesn't, it goes to the next slide, or in case that was the last slide it exits out of the "**for**" loop.

```
if(SteppingUp_SteppingDistance == 0):
```

Then we check to see if the stepping distance has been set from a previous loop iteration, to avoid having it set twice.

```
Step_Cam_PosBefore_Global = to_global(Node_Camera3D.translation)
```

Here we get the current global location of the camera (before it's moved up with the player), and put it into a variable.

If the camera is being interpolated from a previous step and hasn't finished doing so, this variable becomes the new starting point for the interpolation.

```
SteppingUp_SteppingDistance = (Ray_Result.position.y - Player_GlobalFeetPos_Y +
Step_SafetyMargin)
```

Then we find out the distance to move the player up by getting the vertical position of the ray cast result and subtracting the player's "feet" vertical position from it, so we can get can get the height of the step.

As an example, say that the player hits a step. The player's origin on the Y axis is "**2.5**". To get the vertical position of the player's feet we subtract half of the player's height from "**2.5**". The player's height is "**1.8**", so half that is "**0.9**". So we subtract "**0.9**" from "**2.5**" to get "**1.6**", the Y axis position of the bottom of the player.

Then we get the Y axis position of the step. Let's say it's "**2.0**". So let's take that and subtract "**1.6**" from it. We get "**0.4**", which is the height of the step.

After we get the height of the step we add the step safety margin to it all to make sure the character fully rises above the step.

```
CamInterpo_StartingPos_Local_Y = to_local(Step_Cam_PosBefore_Global).y -
SteppingUp_SteppingDistance
```

Now we need to get the starting position of the camera in the player's local space. We do this by getting "**Step_Cam_PosBefore_Global**" (which was set two lines above), converting it to local space with "**to_local()**", and then getting the Y axis component of it.

After we get the local Y axis component of the the camera position, we subtract from it the distance that we are going to move the character up, so that the camera will have the same vertical position before and after the player is moved up a step. To illustrate:

In this picture, it shows how after the player is moved up, the global vertical position of the camera is still the same. We do this by subtracting the *local* position of the camera by the height of the step, because remember that whenever we move the player we move the camera. So any changes to the camera will need to be done in local space.

```
if(CamInterpo_StartingPos_Local_Y < 0.0):
        CamInterpo_StartingPos_Local_Y = 0.0
```

The next couple of lines of code. First we check to see if the starting position of the camera (in local space) is below "**0.0**". This is because a local vertical position below "**0.0**" means that the camera is more than half way down the player! That's too low. I would rather have the camera "jump" in such circumstances than to have the player all the way down to the player feet, if he happens to be moving fast enough.

So if the camera's starting position happens to be below "**0.0**", we just set it to "**0.0**" to keep it from going too far down.

```
global_translate(Vector3(0.0, SteppingUp_SteppingDistance, 0.0))
```

Okay, here we actually DO something, instead of just calculations and crap. This line actually moves the player up over the step. Note that we are not using "**move_and_slide()**". This is because I don't need any slides to take place. I just need to move the player up. Also be aware that "**global_translate()**" uses the physics engine, which means that you can't clip through objects with it. At least not with the Bullet physics engine. I haven't tested with Godot's built-in physics engine.

So if we accidentally move the player up into something that is hanging over the step, the player won't get stuck inside of it, so that's cool.

```
Node_Camera3D.translation.y = CamInterpo_StartingPos_Local_Y
```

This line also actually does something. After moving the whole player up, the camera now needs to come back down to what it was before, so that the camera will be at its starting position for the vertical camera interpolation.

We have already calculated where to move the camera and placed the result in "**CamInterpo_StartingPos_Local_Y**". So all we do have to do is make the camera's vertical position whatever is in "**CamInterpo_StartingPos_Local_Y**".

Note that this line of code isn't actually "moving" the camera like it would if we used "**global_translate()**". Instead what it does is simply change the camera's position, regardless of what it was before.

And, of course, this changes the camera's *local* position. (I hope I'm not being too confusing).

```
CamInterpo_DoInterpolation = true
```

This line sets the flag that tells the script to vertically interpolate the camera. Check the next section, "**InterpolateCamera()**", for more information.

```
CamInterpo_CurrentTime_Secs = 0
```

This resets the camera interpolation timer back to 0 since we have a new step.

Okay, so after this we only have one more line of code:

```
SteppingUp_SteppingDistance = 0
```

This line is executed after the "**for**" loop. It set's the stepping distance back down to "**0.0**" so that the line that checks the stepping distance ("**if(SteppingUp_SteppingDistance == 0.0):**") will allow the code following it to be executed.

InterpolateCamera()

So this function interpolates the camera's position whenever a player takes a step, to allow smoothing of the camera's vertical position, so that movement up stairs and steps look smooth from the player's perspective.

There are most likely better ways of doing this, but since this is just a beginner project I decided that this way will work fine.

Here are some pictures that show the basic idea of how this works:

In this this picture, the player hits a step. It goes through "**Step_Player_Up()**", finds out how high the step is, records the current vertical position of the camera, and then steps the player up.

After that, the camera is moved back down to what it was before the player moved up the step. Then (on the right side of the picture) the camera is moved back up to its default position over time, which is defined by "**CamInterpo_DefaultPosition_Local_Y**". The time it takes for the camera to get to its default position is defined by the "**CamInterpo_Length_Secs**" setting at the top of the script.

Arguments

"**InterpolateCamera()**" has 3 arguments: "**Prev_Pos_Local_Y**", "**Time_Current**", and "**Time_Delta**".

"**Prev_Pos_Local_Y**" is the position that the camera was at before the player was moved up the step. This is set inside "**Step_Player_Up()**" whenever the player is moved up the step. If the player steps up another step while the camera is still being interpolated the current local position of the camera will be the new "**Prev_Pos_Local_Y**".

"**Time_Current**" is the current time of the interpolation, in seconds. So if the length of the interpolation was "**0.25**" seconds, "**Time_Current**" will be anything between "**0.0**" to "**0.25**".

"**Time_Delta**" is the delta of the frame time, in seconds. The argument specified will increment "**Time_Current**". Simply put "**delta**" here when calling from "**_process()**" or "**_physics_process()**". So if the delta of the frame was "**0.016**" and "**Time_Current**" was "**0.05**", "**Time_Current**" would then be "**0.066**".

Example Usage

Let's say that we are going to call this function "**_physics_process()**". It would look like this:

```
Current_Interpo_Time = InterpolateCamera( Cam_Start_Pos, Current_Interpo_Time , delta )
```

Note that "**InterpolateCamera**" returns a value, and we are putting it into a variable called "**Current_Interpo_Time**". This is because the function adds "**delta**" to the current interpolation timer inside of the function and returns it back to the timer variable so that it can be used in the next frame.

We also have "**Current_Interpo_Time**" being used as an argument. This is because we need it to be able to tell what position the camera will be and to increment the timer.

The Code

The first line of code is:

```
if(Time_Current < CamInterpo_Length_Secs):
```

It checks to see if the current time of the interpolation is less than the length of the interpolation process as set in the "**SETTINGS**" section. Remember that "**Time_Current**" isn't a global variable, but is one of the arguments specified when calling this function.

If the camera interpolation isn't over yet….

```
Node_Camera3D.translation.y = lerp(Prev_Pos_Local_Y, CamInterpo_DefaultPosition_Local_Y, Time_Current / CamInterpo_Length_Secs)
```

We start off by changing the camera's position by using a "**lerp()**" function. Check the "lerp()" chapter for an explanation.

After that:

```
Time_Current += Time_Delta
```

We simply increment the timer according to the delta specified.

```
return Time_Current
```

And we return the current time after being incremented so we can use it in the next frame loop.

Now, if the interpolation is over and the camera has reached its final position:

```
# Otherwise, if time is up and the camera is in it's final position...
else:
	# Make sure to manually set the final Y position of the camera, just in case.
	Node_Camera3D.translation.y = CamInterpo_DefaultPosition_Local_Y
```

I included the comments from the code for clarity.

As you can see, in the first line of code in the "**else**" statement we simply move the vertical position of the camera to its default position. This is done to make sure that the camera is *exactly* where it's supposed to be when the interpolation is over, as sometimes the timer can be off a bit when it goes over the time limit.

An example: let's say that the current timer is "**0.225**" and that the interpolation length is "**0.25**". So the camera position is interpolated in the "**if**" statement above as being 90% of the way though interpolation. And then let's say the delta is "**0.03**", so the timer is incremented to "**0.255**".

Since "**0.255**" is over the time limit, which is "**0.25**", the "**if**" statement above is not executed, so that means it will go to the "**else**" statement. That being so, we will have to set the final position of the camera ourselves by simply stating that the position of the camera will now be the default one, "**CamInterpo_DefaultPosition_Local_Y**", which is set whenever the script is loaded.

After that:

```
CamInterpo_DoInterpolation = false
```

We set the state that says to stop the interpolation.

```
return 0.0
```

And then we return 0, which then resets the counter to 0. And that's it.

Slope_AffectSpeed()

This function finds out whether the player is walking up, down, or sideways on a inclined surface and adjusts the speed accordingly.

For instance, if the player is walking up a slight incline, his speed will be slightly slower. But if the player is walking up a steep incline, his speed will be relatively much slower.

Or let's say the player is walking down the slope. In video games one of the basic problems with player character's is that when they walk down a slope, they tend to "jump" down the slope. Here's an example picture:

In this case, what I did was make gravity pull the player down *more* when walking down a slope. This resulted in two things: the player walking down the slope faster, and the player staying on the slope.

Both these effects are realistic. Imagine you can run very fast and you are standing on the side of a steep slope. What would happen if you just started running at top speed down the slope? At the onset of your running you would jump off the ground a little bit because you are moving horizontally faster than gravity can pull you down. But then gravity catches up and starts pulling you down, and as you keep running you stay on the slope.

There are better ways of doing this, but this is a basic but effective way and is good for teaching as it's not too complicated.

In another situation let's say that this slope is very wide, and the player is walking along it, not going up or down. Since he is moving along the slope his velocity isn't slowed down or sped up, as he isn't fighting against or going with gravity.

This is how this function is set up. So now that we have an overview let's get to the code:

```
if(SlideCount > 0):
```

First we check if there even any slides to be bothered with. If there are we move on and if there aren't we break out of the whole function.

```
for Slide in range(SlideCount):
```

Next we go through all the slides in a loop.

```
if(get_slide_collision(Slide).normal.y > MaxFloorAngleNor_Y):
```

Now we check to see if the slide normal is a "floor" or not. If it is, we move on. If not, we go to the next slide, or if that was the last slide we break out of the function.

```
Slope_FloorNor2D = Vector2(get_slide_collision(Slide).normal.x,
get_slide_collision(Slide).normal.z).normalized()
```

We get the horizontal floor normals; the X and Z axis. We get these instead of the Y axis because we want to find out what direction the slope is facing. We put the X and Z axis normals of the floor slope and put them into a 2D vector.

And, of course, we normalize the 2D vector so we can have accurate dot products.

```
Slope_PlayerVelVec2D =
Vector2(
        (TempMoveVel_FWAndBW.x +     TempMoveVel_LeftAndRight.x) ,
        (TempMoveVel_FWAndBW.z + TempMoveVel_LeftAndRight.z)
        ).normalized()
```

Sorry if this is hard to read, but it's kind of long and complicated so I formatted it to be a little easier to read.

What we have here is the horizontal velocity of the player. This is used to dot product against the slope's normal so we can find out which direction the player is walking on the slope.

```
Slope_DotProduct = Slope_FloorNor2D.normalized().dot(Slope_PlayerVelVec2D)
```

Check the "Dot Product" section for more information on how they work.

Here we get the dot product of the normalized floor vector and the player's velocity vector.

Check the file "**slope_dot_pro.mp4**" in the "**documents/manual_reference_files**" folder for a visualization of what we have looked at so far. (I can't put the video file here, for some reason).

```
if(Slope_DotProduct > 0):
	Slope_DotProduct *= Slope_EffectMultiplier_ClimbingDown
else:
	Slope_DotProduct *= -Slope_EffectMultiplier_ClimbingUp
```

We then check to see if the player is going up or down the slope. If the dot product is more than 0 (that means he's moving along with the normal of the slope), the dot product is multiplied by the effect multiplier for climbing down. And if the dot product is less than 0, the opposite happens, but instead of multiplying by a positive number, we are multiplying by a negative number. This is to avoid the player's velocity from being positive, as the dot product when moving up a slope is negative and we multiply the falling speed by it, causing it to be negative. Then later when we calculate the final vertical velocity of the player we ultimately multiply the falling speed variable by a a negatived "**Falling_Speed**", as shown here:

```
Falling_Speed = Falling_Gravity * Falling_Speed_Multiplier
FinalMoveVel.y = -Falling_Speed
```

So if the slope dot product variable is negative the final velocity will be inverted and the player will jump up the slope.

Moving on, "**Slope_EffectMultiplier_ClimbingDown**" by default is "**1.5**". This means that when going down a ramp the character is pulled down harder by gravity than when going up it or walking on a flat surface. The reasons for this are outlined at the beginning of this section. The "**Slope_AffectSpeed()**" section, that is.

```
Falling_Speed_Multiplier = lerp(
Falling_Speed_Multiplier_Default ,
1.0 ,
```

```
pow(
        acos( get_slide_collision(Slide).normal.y ) / acos( MaxFloorAngleNor_Y ) *
                Slope_DotProduct ,
        2.0
    )
)
```

So we have a "**lerp()**" function. For more information on it check the last part of the "InterpolateCamera()" section, or just go back a few pages, about 4.

The first argument is the default speed multiplier. By default this is "**0.25**" and what this does is make the falling speed of the character 25% of gravity. Since gravity defined in this script is defined as "**9.8**", that means that the default falling speed would be "**2.45**".

In this script, when the player is standing still, his vertical velocity is, by default, "**-2.45**". This is because whenever the gravity is more than that and the player is standing on an incline, he will slide down instead of standing still. So that's why we have "**Falling_Speed_Multiplier_Default**".

So in this function, "**Falling_Speed_Multiplier_Default**" is our starting point for how much gravity affects the player when walking around or moving.

As the second argument we have "**1.0**". Let's say the player hit a very steep incline that is close to the limit of what the player can walk on (according to "**MaxFloorAngleNor_Y**"). And let's say that the after all the math "**Falling_Speed_Multiplier**" ends up bing about "**0.9**". That means that whenever we get to the point of calculating the final falling speed of the character in the "**_physics_process() > VERTICAL MOVEMENT > FALLING**" section, the character's vertical velocity would be "**-8.82**". And this only happens when the player is moving.

And the final argument is the weight of the linear interpolation. Let's take a look and then break it all down:

```
pow(
        acos( get_slide_collision(Slide).normal.y ) / acos( MaxFloorAngleNor_Y ) *
                Slope_DotProduct ,
        2.0
    )
```

And the first part we are going to look at is this:

```
acos( get_slide_collision(Slide).normal.y ) / acos( MaxFloorAngleNor_Y )
```

So the first thing this "**lerp()**" line does is get the angle, in radians, of the floor normal and divide it by the angle, in radians, of the maximum floor angle the player can walk on. Here an example:

```
acos( get_slide_collision(Slide).normal.y ) = 0.436317612 (which is about 25 degrees)
acos( MaxFloorAngleNor_Y ) = 0.700000291 (which is about 40 degrees)
```

0.436317612 / 0.700000291 = 0.623310615

So what this does is find out, in percentage, how steep the floor is in relation to the maximum floor angle the player can walk on. It gives us about "**63%**", which is basically saying that the floor is 63% of the way to the floor angle limit.

Continuing from our example, the next part is this:

Slope_DotProduct = 0.25

0.623310615 * 0.25 = 0.155827654

"**Slope_DotProduct**" was already calculated in the last few lines of code, so now we multiply it against our ratio. This will result in the amount that our character will be affected by gravity.

But before we let this be the *final* amount to affect the player's gravity, let's make the gravity falloff smoother, as right now it's linear. So what we do is use "**pow()**" on our result like this:

pow(0.155827654 , 2.0) = 0.024282258

Check the "**Math Terms and Functions Explained > pow()**" for how it works and what it can be used for.

We now have the final weight we want for our "**lerp()**" function, and now we can get our final result:

Falling_Speed_Multiplier_Default = 0.25

Falling_Speed_Multiplier = lerp (0.25 , 1.0 , 0.024282258) = ~0.268212

So our final falling speed multiplier equals "**0.268212**". That means that when the final movement velocity is calculated, the vertical velocity will be "**-2.6284776**".

The reason this is so low is because of two reasons:

1) In our example the dot product was multiplied by "**Slope_EffectMultiplier_ClimbingUp**", which is "**0.25**" by default. This is because we don't want the character to walk *too* slowly up the slope. If you want to change how slowly the character walks up the slope, then modify "**Slope_EffectMultiplier_ClimbingUp**".
2) The "**lerp()**" weight argument was squared, so that we could have a smoother falloff. Check the "**Math Terms and Functions Explained > pow()**" section for more information.

So after everything is done "**Falling_Speed**" will be used as the final vertical velocity, but will be negated at the end of the "**_physics_process() > VERTICAL MOVEMENT > FALLING**" section, shown here:

```
# Apply final falling velocity.
FinalMoveVel.y = -Falling_Speed
```

And then "**FinalMoveVel**" is used when calling "**move_and_slide()**".

That's it.

Touch_CheckAndExecute()

This function allows the player to activate an object with a "**Touched_Function()**" method in the script attached to it when touched.

It creates a list of objects that the player has touched, and when the player is no longer touching the object, gets rid of whatever objects are no longer being touched from the list and allows the object's "**Touched_Function()**" to once again be used.

The first line:

```
if(State_OnFloor or State_OnWalls or State_OnCeiling):
```

It checks to see if the player is touching anything at all. If the player is in the air then there's no point in wasting computational time with this function.

If the player *is* on a wall, floor, or ceiling…

```
if(SlideCount > 0):
```

This is here just as a safeguard. If the player is on a ceiling, wall, or floor, then of course there's a slide collision. But just in case there isn't, we have this here.

```
for Slide in range(SlideCount):
```

We start going through the slides.

```
if(get_slide_collision(Slide).collider.has_method("Touched_Function")):
```

The first thing we do is check if the collider of the slide collision has a "**Touched_Function()**" method in a script attached to it. If it does:

```
if(not Touch_ObjectsTouched.has(get_slide_collision(Slide).collider)):
```

We then check to see if the collider is already inside the array that holds the list of colliders with said function that have already been touched. If is has already been touched, nothing happens. If it hasn't:

```
Touch_ObjectsTouched[Slide] = get_slide_collision(Slide).collider
```

We add the collider to the list of objects that have been touched.

```
Touch_ObjectsTouched[Slide].Touched_Function()
```

And then we activate the function in question.

After that for loop is finished, we go on to these lines:

```
SlideCollisions.clear()
SlideCollisions.resize(MaxSlides)
```

This clears the list that holds the current slide collisions of the player, and then resizes it to the number of maximum slides the player can have at any one time. It's resized because "**clear()**" sets the size back down to zero.

```
for Index in range(SlideCollisions.size()):
```

Now we have a new for loop, that goes through the "**SlideCollision**" array, adding all the current slide collisions of the player to it, so we can compare them to the "**Touch_ObjectsTouched**" list to see if we are still touching any of those objects.

```
if(Index < SlideCount):
```

So this checks to see if the current iteration of the for loop is less than the amount of current slide collisions, so that we don't try to access a slide collision that doesn't exist, causing a segmentation fault.

If it is less then the slide count:

```
SlideCollisions[Index] = get_slide_collision(Index).collider
```

This then adds the current slide to the slide collision array.

On the other hand, if the current loop iteration is more than the number of slides:

```
else:
	# Set the current element to null.
	SlideCollisions[Index] = null
```

We simply set the current element of the slide collision array to "**null**". Simple.

After that we start a new loop:

```
for Index in range(Touch_ObjectsTouched.size()):
	if(Touch_ObjectsTouched[Index] != null):
```

I've concatenated these two lines in this example for expediency.

The first line starts a new for loop which goes through each element of the "**Touch_ObjectsTouched**" array, checking to see if any currently touched objects are still being touched.

The second line checks to see if the current element of the array is empty, so as to not try to accessing something that isn't there.

```
if(not SlideCollisions.has(Touch_ObjectsTouched[Index])):
	Touch_ObjectsTouched[Index] = null
```

Then it checks to see if the "**SlideCollisions**" array has the current element in the "**Touch_ObjectsTouched**" array.

If it doesn't, then it nullifies the current element in the "**Touch_ObjectsTouched**" array, to basically say that object isn't being touched anymore and therefore can be activated against on the next collision with it.

Now that we've taken a look at what happens when there's a collision slide, now we are going to look to see what happens when there *isn't* a slide collision:

```
# Otherwise, if there aren't slide collisions...
else:
	Ray_SpaceState = get_world().get_direct_space_state()
	Ray_From = self.get_global_transform().origin
	Ray_To = Ray_From
	Ray_To.y -= ((Player_Height * 0.5) * 1.05)
	Ray_Result = Ray_SpaceState.intersect_ray(Ray_From,Ray_To,[self])
```

As you can see, what happens if there isn't a collision slide is that we cast a ray from the center of the player, straight down to a tiny bit below his feet, and see if there is anything there.

This is because if you jump on an object, even though the collision is registered between the player and the object, no actual *slide* occurs. That's because the player isn't actually sliding along the surface and is, instead, falling straight down onto it, and doesn't move left or right. The slide collisions you get with "**get_slide_collision(Slide)**" and exactly that: the *slide*, not the collision. So if you were to walk 100% straight into a wall, the wall would not be registered as a slide collision, and "**is_on_wall()**" would return false. This is a design issue with Godot and will probably be fixed someday (written as of March 31, 2018, with Godot 3.0.2 being the current stable release).

Since we can't get a slide collision from the player body, we need to shoot a ray cast down from the center of the player. This may miss an object that the player is touching but isn't directly under him, but if that is the case the chances are that it will be detected within a slide collision, as the player would slide off the side because the player's collision shape is a radius.

After we shoot the ray cast down from the center of the player:

```
if(Ray_Result):
	if(Ray_Result.collider.has_method("Touched_Function")):
		if(not Touch_ObjectsTouched.has(Ray_Result.collider)):
```

We first check if there is even a result from the ray cast.

Then we check if the ray cast hit an object with a "**Touched_Function()**" method.

If it has that function, then we check if the "**Touch_ObjectsTouched**" array already has that object. If it doesn't, then we move on to:

```
Touch_ObjectsTouched[0] = Ray_Result.collider
Touch_ObjectsTouched[0].Touched_Function()
```

We add the collider to the touched object list as the first element. As no slide collisions are happening and there is only one ray cast, we can do this.

Then we activate the function of that object.

So that shows what happens if we have a result of the ray cast, but what happens if we don't? It's exactly the same thing that happens if the slide count is greater than 0. But I'll explain it once more, for completion:

```
else:
	SlideCollisions.clear()
	SlideCollisions.resize(MaxSlides)
```

First we clear the slide collision array. Then we resize it back to what it was.

```
for Index in range(SlideCollisions.size()):
	if(Index < SlideCount):
		SlideCollisions[Index] = get_slide_collision(Index).collider
```

Then we setup a for loop that goes through the slide collision array. It checks to see if the current loop iteration is less than the slide count. If it is, it goes on to adding the current slide collision to the slide collision index.

```
else:
        SlideCollisions[Index] = null
```

However, if the for loop iteration is more than the slide count, then what we do is simply make the slide collision array element "**null**".

```
for Index in range(Touch_ObjectsTouched.size()):
        if(Touch_ObjectsTouched[Index] != null):
                if(not SlideCollisions.has(Touch_ObjectsTouched[Index])):
                        Touch_ObjectsTouched[Index] = null
```

Then we have one last for loop that goes through the "touched objects" array.

It first checks if the current element that we are looking at is null. If it is, we do nothing. If it isn't, we go on to check if current object that we are looking at, that has been previously touched, is still being touched. If it is still being touched, we leave the object inside the "**Touch_ObjectsTouched**" array. But if that object is no longer being touched, we get rid of it by nullifying the element of the array that it occupies.

And there you have it, all the functions explained, in detail. Now let's go on to the rest of the script.

Chapter 4: ready()

This part is simple. All that "**ready()**" does is initialize variables and states, just in case there is a situation to where the player starts out in the level needing these variables. They may be unnecessary as most of these variables are set before they are used inside their respective function, but it makes me feel better to initialize them, anyway, as they take almost no time to do so. And you never know, someone may change the code so as to make them needed.

The first lines:

```
set_process_input(true)
set_physics_process(true)
```

These just activate the input and physics processes.

```
move_and_slide(
                Vector3(0.0, 0.0, 0.0) ,
                FloorNormal ,
                SlopeStopMinVel ,
                MaxSlides ,
                MaxFloorAngleRad
                )
```

Then we "move" the player. Notice the player doesn't move anywhere. This is because the functions that check if the player is on the floor, ceiling, or wall are only updated when "**move_and_slide()**" is used. So that's what we are doing here.

```
State_OnFloor = is_on_floor()
State_OnWalls = is_on_wall()
State_OnCeiling = is_on_ceiling()
```

After that set the collision states.

```
State_Falling = true
```

Then we set the player's falling state to true, so that he starts falling when the level is started.

```
if(not Input.is_action_pressed(String_Jump)):
        Jump_Released = true
```

This says whether or not the player is pressing the jump button. If he is, then he can't jump until he releases it and then presses it again. This is just a safeguard and not that important.

```
        Player_Position = translation
```

This sets the initial position of the player's position variable.

```
    Player_GlobalFeetPos_Y = Player_Position.y - Player_Height * 0.5
```

Then we get the global vertical position of the player's feet.

```
    if(Step_MaxHeight > Player_Height * 0.5):
        Step_MaxHeight = Player_Height * 0.5
```

Then we check if the maximum step height is to high (above the halfway point of the player). If it is then we set it down to be half of the player's height.

```
    Debug_Label.set_text(Debug_Label_String)
```

Then we set the initial debug label text.

```
    Touch_ObjectsTouched.resize(MaxSlides)
```

And then we resize the "touched objects" array.

Chapter 5: _unhandled_input()

Here we take care of mouse movement. The reason we don't take care of any other input is because I can't figure out how. I've already tried looking online and experimenting but to no avail. So for now I just do other inputs within other functions, like the "**_physics_process()**" function.

Unhandled input, and also the "**_input()**" function only activate when there is actual input. So that means that if the player isn't moving a mouse or pressing a key, this function isn't executed.

The function line looks like this:

```
func _unhandled_input(event):
```

"event" is the event that we are going to be checking. It can be many types of events, be it key presses, mouse movement, or joystick movement.

The first lines:

```
if(event is InputEventMouseMotion):
    if(MouseLook):
```

First it checks to see if the current input event is mouse motion. Then it checks to see if the "**MouseLook**" variable is true, allowing the player to look around with the mouse.

```
Mouse_Rel_Movement = event.relative
```

This gets the relative mouse movement in a 2D vector. Relative mouse movement is how many pixels, in floating point format, that the mouse cursor has moved from the last frame. It's best used with the mouse cursor being captured by the program, naturally.

So if you, say, move the mouse 1 pixel to the left and 10 pixels up, the vector would look like this:

```
Mouse_Rel_Movement = Vector2( -1.0 , -10.0 )
```

Then you use that to find out how much to turn the player.

```
Cam_Local_Rot_X = rad2deg(Node_Camera3D.get_rotation().x)
```

This gets the local rotation of the camera on the X axis. Now, this can be confusing because when you think of "X axis" you may think of left and right. That's correct. But rotation on the X axis means looking up and down. Understand? Here are some pictures to help (on the next page):

The above picture shows the camera moving on the x axis.

And this picture shows the camera *rotating* on the x axis.

```
Cam_Temp_XRot_Var = Cam_Local_Rot_X + -Mouse_Rel_Movement.y
```

Now that we have the X axis rotation of the camera, now we will calculate the new rotation of the camera. We do that by adding the negated relative Y mouse movement to the X axis rotation value of the camera we just got. This will be a temporary variable.

Now we need to check that temporary X axis rotation to see if the player's camera is past looking straight up or down.

```
if(Cam_Temp_XRot_Var < -90):
        Final_Cam_Rot_Local_X = Cam_Temp_XRot_Var + 90 +
                                Mouse_Rel_Movement.y
```

This "if" statement checks if the temporary rotation variable is too far down, past looking straight down. If it is we set the rotation to be applied to camera to be as much as it takes to make the camera look straight down.

```
elif(Cam_Temp_XRot_Var > 90):
        Final_Cam_Rot_Local_X = Cam_Temp_XRot_Var - 90 + Mouse_Rel_Movement.y
```

On the other hand, if the camera is looking too far up, we do the same thing with this but in the opposite direction.

```
else:
        Final_Cam_Rot_Local_X = Mouse_Rel_Movement.y
```

And if the camera X rotation isn't going past the upper or lower bounds, we simply make the final rotation to be applied to the camera the same number as the mouse's relative Y axis movement.

```
Mouse_Moved = true
```

And then we set "**Mouse_Moved**" to true, so that when it comes time to rotate the camera and the player, it will do so. But this isn't done until the "**_physics_process() > FINAL MOVEMENT APPLICATION**" section.

Chapter 6: _physics_process()

This is the meat of the script.

FYI, just in case you don't know, "**_physics_process()**" is called a certain amount of times per second according to the project's "Physics FPS" option in "Project Settings".

Let's get into it.

Get Info

We start off by getting info about the current state of the player, which is what you always want to do at the start of a loop, any kind of loop.

```
State_OnFloor = is_on_floor()
State_OnWalls = is_on_wall()
State_OnCeiling = is_on_ceiling()

SlideCount = get_slide_count()

Player_Position = translation
```

The first thing we need to do with this script is get information. Getting information is half of what this script does. Only a few lines actually do anything, like move the character.

In the first few lines we check the wall, floor, and ceiling states. Then we get the slide count and set the player's position variable.

```
DirectionInNormalVec3_FWAndBW = get_global_transform().basis.z
DirectionInNormalVec3_LeftAndRight = get_global_transform().basis.x
```

These variables hold the forward and sideways normals, for use with horizontal movement, including strafing.

"**DirectionInNormalVec3_FWAndBW**" gets the forward normal of the player character. That is, the normal that tells us which direction the player is facing. Note that this represents just the player's kinematic body, and not the camera. So if you are looking up or down, this normal will still be horizontal.

It just so happens that in order to find out the direction that our player is facing, all we need to do is get the player's Z axis transform normal. This can be visualized as such:

For information on how normals work go to the "Normals Explained > Normals" section.

In this picture, the thing we are needing is the blue arrow, which represents the Z axis normal, the direction the player is facing. Also note that these normals are in local space.

It's the same way with "**DirectionInNormalVec3_LeftAndRight**". All we need to get is the player's X axis transform, the red arrow.

Be sure to check out the "**Normals Explained > Transforms**" chapter for information on what transforms are. The X axis of the vector would look like this:

In this picture, the X axis is highlighted. So the "**x**" we are getting is the normal's coordinates.

The X normal that is circled here has this position:

```
X_Normal = (-0.914192 , 0.0 , 0.405281)
```

These two vectors will be used later in the "**_physics_process() > HORIZONTAL MOVEMENT**" section.

Input

Now we are moving into the "**INPUT**" section. The first few lines are:

```
Pressed_FW = Input.is_action_pressed(String_FW)
Pressed_BW = Input.is_action_pressed(String_BW)
Pressed_LEFT = Input.is_action_pressed(String_Left)
Pressed_RIGHT = Input.is_action_pressed(String_Right)
```

This simply gets the current input states of the horizontal movement actions. They are just bools.

```
if(Input.is_action_pressed(String_Jump)):
	Pressed_Jump = true
	if((State_Jumping or State_Falling) and not State_OnFloor):
		Jump_Released = false
else:
	Pressed_Jump = false
	Jump_Released = true
```

I'm going to start concatenating the code examples from now on for expediency.

This code block takes care of the jump action. It first checks to see if the jump action is pressed. If it is it sets the "**Pressed_Jump**" bool to true. Now the next part in that "if" statement checks if the player is even on the ground (that he is falling or jumping). If he's not on the floor he can't jump, so the "**Jump_Released**" bool, which allows or denies a jump based on its state, is set to false so as to not allow a jump to occur in midair.

Otherwise, if the jump key is not being pressed then we say that the jump key is not being pressed and that it has been released, so that the player can now jump again.

```
if(Input.is_action_pressed(String_Shift)):
	FinalWalkVelocity = BaseWalkVelocity * ShiftWalkVelocity_Multiplier
else:
	FinalWalkVelocity = BaseWalkVelocity
```

Here we have the speed shift action. If the "**Shift**" key or the shift speed action is being pressed, we multiply the player's walk velocity according to "**ShiftWalkVelocity_Multiplier**". By default "**ShiftWalkVelocity_Multiplier**" is "**2.0**" and the base walk velocity is "**10**". So that means if the shift action is pressed the final velocity will be "**20**", so that the player can move faster.

```
Step_SafetyMargin =
	Step_SafetyMargin_Dividend / (BaseWalkVelocity / FinalWalkVelocity)
```

Here we modify the safety margin of the player's stepping function. This makes the player move up a little past the step to make sure the player actually gets onto the step instead of just the corner, resulting in the player getting stuck on the side of the step.

We do this by dividing the base walk velocity, "**10**", by the final walk velocity, which we'll say is "**20**" in this example. The result is "**0.5**".

Then we divide"**Step_SafetyMargin_Dividend**" by that result, resulting in "**0.04**". If the final walk velocity was "**10**" the ultimate result would be "**0.02**". Evidently this is because when moving more quickly it's necessary to make sure the player goes up the step farther to keep him from hitting the edge. I'm honestly not sure, though, but it works.

```
CamInterpo_Length_Secs = BaseWalkVelocity /
		( CamInterpo_Length_Secs_Multiplicand * (FinalWalkVelocity / 10.0) )
```

Here we take care of the camera's interpolation length. This way when the player is moving faster, the camera is interpolated faster to keep up with the player.

It sets the length, in seconds, by first dividing the final walk velocity by 10. Let's say that the final walk velocity is "**20**" because the player has the shift speed action pressed.

```
FinalWalkVelocity / 10.0 = 2.0
```

Then we multiply "**CamInterpo_Length_Secs_Multiplicand**" by that result. "**CamInterpo_Length_Secs_Multiplicand**" is "**125.0**". So that means the product will equal "**250.0**".

Lastly we divide the base walk velocity by that, resulting in “**0.04**” seconds, which is our new camera interpolation length. If the final walk velocity was “**10**” then our interpolation length would equal “**0.08**” seconds. This is longer because the player is walking up the stairs slower and therefore the camera doesn’t need to interpolate as fast.

```
Ray_SpaceState = get_world().get_direct_space_state()
Ray_From = Node_Camera3D.get_global_transform().origin
Ray_To = -Node_Camera3D.get_global_transform().basis.z * Ray_UseDist + Ray_From
Ray_Result = Ray_SpaceState.intersect_ray(Ray_From,Ray_To,[self])
```

Now we are entering the section that takes care of the “Use” action and the red circle that shows up whenever the player is looking at something he can interact with.

The first thing we do is setup a ray cast and then execute it. It starts from the player’s camera and goes along the direction that it’s pointing. This goes as far as “**Ray_UseDist**” is set, which by default is “**2.0**”. “**Ray_From**” is added as the offset of the vector, giving it a global position.

As a side note, the last option in the “**intersect_ray()**” function is any objects or nodes that you want the ray cast to ignore. If you put in “**[self]**”, this not only includes the kinematic body that this script is connected to but all its child nodes.

```
if(not Ray_Result.empty()):
```

We check if the ray hit anything. If it did we go on to this code, here:

```
if(Ray_Result.collider.has_method("UseFunction")):
	Node_Crosshair_Useable.visible=true
	if(Input.is_action_just_pressed(String_Use)):
		Use_Ray_IntersectPos = Ray_Result.position
		Ray_Result.collider.UseFunction()
```

If the collider of the ray cast has a function named “**UseFunction**”, we set the red circle node, which is the “**Camera2D/Crosshair/Crosshair_Useable**” node in the player scene, to be visible.

Then we check to see if the “**Player_Use**” action is being pressed. If it is we simply execute the “**UseFunction()**” function in the collider’s script.

```
elif(Ray_Result.collider.get_parent().has_method("UseFunction")):
	Node_Crosshair_Useable.visible=true
	if(Input.is_action_just_pressed(String_Use)):
			Use_Ray_IntersectPos = Ray_Result.position
			Ray_Result.collider.get_parent().UseFunction()
```

Otherwise, if the collider doesn’t have a function, then we check to see if the collider’s immediate parent has one. This is a “just in case” scenario and may never be executed. That’s why I put the previous "if" statement first, as it’s more likely to happen and therefore won’t have to go though multiple "if" statements to check for a “**UseFunction()**” function.

In any case we do exactly the same thing here that we did in the previous "if" statement, but on the parent of the collider.

Note that the “collider” is always the physics body that holds the collider shape, not the shape itself. So if you had a rigid body that had a sphere collision shape, the collider returned would be the rigid body, not the shape.

```
# Otherwise, if there is no use function...
else:
	Node_Crosshair_Useable.visible=false
	Use_Ray_IntersectPos = Vector3(0,0,0)
```

This “else” statement is for when there isn’t any “Use” function in the collider. If that’s the case, we simply hide the red circle and set the intersection point of the “Use” ray to a 3D vector that is all 0’s. This is so no accidental actions or incorrect impulse positions occur because of a left over variable. Also this may help prevent a wrong return variable when an external script tries accessing “**Use_Ray_IntersectPos**”. This may not actually do anything, but I’m not totally sure.

```
# Else, if the ray hit nothing...
else:
	Node_Crosshair_Useable.visible = false
	Use_Ray_IntersectPos = Vector3(0,0,0)
```

This last part of this section happens if there isn’t any intersection of the ray cast. It hides the red circle and set the intersection point of the “Use” ray to a 3D vector that is all 0’s, like in the previous code block.

Horizontal Movement

Now we get information on the horizontal movement by the player and set the velocity of such.

First off let’s take a look at the “if” and “else” statements in the “**FORWARD AND BACKWARDS CALCULATIONS**” section:

```
if(Pressed_FW and not Pressed_BW):
	# Move forward.
elif(not Pressed_FW and Pressed_BW):
	# Move backward.
elif(Pressed_FW and Pressed_BW):
	# Don’t move forward or backward.
elif(not Pressed_FW and not Pressed_BW):
	# Don’t move forward or backward.
else:
	# Don’t move forward or backward. This is a fail safe case.
```

Here we have the basic action taken in each “if” or “else” statement. Let’s go through each case.

First, we check to see if the "**Pressed_FW**" action is being pressed. At the same time we also check to see if the "**Pressed_BW**" action is *not* being pressed. We need to be specific about this. If the player if pressing the forward action, then the player will move forward.

Otherwise, if the player is not pressing "**Pressed_FW**" and is pressing "**Pressed_BW**", we make the player move backwards.

If that's not the case, we then check to see if *both* the forward and backward actions are being pressed. If they are, we don't do any movement at all. This isn't that important but I like it as I find it to be a cleaner way of doing things. And if the player is accidentally pressing both he won't fall off the edge of a platform or cliff eh may be on.

The last two statements are for when nothing is being pressed. The first "elif" statement checks if neither the forward or backwards keys are being pressed. If not then there is no velocity added in the forward or backward direction.

The very last "else" statement is there in case that, for whatever reason, none of the "if" statements before it don't work. I have no idea how that ma happen but you never know what could happen in the future. And it cost a microscopic amount of memory so I keep it in there.

Now that we've got the overview of how the "if" statements work let's look at all the code:

```
#       FORWARDS #
if(Pressed_FW and not Pressed_BW):
        TempMoveVel_FWAndBW.x = -DirectionInNormalVec3_FWAndBW.x
        TempMoveVel_FWAndBW.z = -DirectionInNormalVec3_FWAndBW.z
#       BACKWARDS       #
elif(not Pressed_FW and Pressed_BW):
        TempMoveVel_FWAndBW.x = DirectionInNormalVec3_FWAndBW.x
        TempMoveVel_FWAndBW.z = DirectionInNormalVec3_FWAndBW.z
#       BOTH            #
elif(Pressed_FW and Pressed_BW):
        TempMoveVel_FWAndBW.x = 0
        TempMoveVel_FWAndBW.z = 0
#       NEITHER                 #
elif(not Pressed_FW and not Pressed_BW):
        TempMoveVel_FWAndBW.x = 0
        TempMoveVel_FWAndBW.z = 0
#       FAILSAFE    #
else:
        TempMoveVel_FWAndBW.x = 0
        TempMoveVel_FWAndBW.z = 0
```

When the player presses the forward or backwards action the direction the player is looking is taken from "**DirectionInNormalVec3_FWAndBW**" and put into "**TempMoveVel_FWAndBW**". Note that we only take the X and Z axis of that normal. The Y axis velocity will be taken care of in different sections, like the falling or jumping section.

Also see that when the player presses the backwards action it simply does the opposite of what pressing the forward action does. Which in this case means that it's positive. I've already explained this before in the "**_physics_process() > Get Info**" section. But a shortened version of the explanation

would be that when I made the character I pointed him in the wrong direction on the Z axis. This doesn't cause any physics problems, it only means that I have to make the forward/backward movement velocity the opposite of whatever it would be if I had faced the player character in the +Z direction in the "**Player.tscn**" file.

```
	#	LEFT		#
	if(Pressed_LEFT and not Pressed_RIGHT):
		TempMoveVel_LeftAndRight.x = -DirectionInNormalVec3_LeftAndRight.x
		TempMoveVel_LeftAndRight.z = -DirectionInNormalVec3_LeftAndRight.z
	#	RIGHT		#
	elif(not Pressed_LEFT and Pressed_RIGHT):
		TempMoveVel_LeftAndRight.x = DirectionInNormalVec3_LeftAndRight.x
		TempMoveVel_LeftAndRight.z = DirectionInNormalVec3_LeftAndRight.z
	#	BOTH		#
	elif(Pressed_LEFT and Pressed_RIGHT):
		TempMoveVel_LeftAndRight.x = 0
		TempMoveVel_LeftAndRight.z = 0
	#	NEITHER		#
	elif(not Pressed_LEFT and not Pressed_RIGHT):
		TempMoveVel_LeftAndRight.x = 0
		TempMoveVel_LeftAndRight.z = 0
	#	FAILSAFE	#
	#This is a failsafe case.
	else:
		TempMoveVel_LeftAndRight.x = 0
		TempMoveVel_LeftAndRight.z = 0
```

Here it's the same as before, but we are doing the left and right movement of the character.

So if the player presses left, the X axis velocity of "**TempMoveVel_LeftAndRight**" will equal the negated value of the X axis of "**DirectionInNormalVec3_LeftAndRight**". And the Z axis will equal the negated value of the Z axis of "**DirectionInNormalVec3_LeftAndRight**".

Let's look at how that would be visualized:

This is the same picture I used in the "_physics_process() > Get Info" section of this document.

Let's say that that the X normal in this represents "**DirectionInNormalVec3_LeftAndRight**" and that it equals "**(-0.914192 , 0.0 , 0.405281)**". So when the player presses left, we want to take the negative value of the X axis of that vector and negate it. So that means it would look like this:

So now that we've negated the X axis, we will do the same now for the Z axis, which will look like this (on next page):

And as you can see the X normal is now on the left side of the player. And that's how we get the global direction to the left of the player.

So that means in the next "elif" statement that looks like this:

```
elif(not Pressed_LEFT and Pressed_RIGHT):
    TempMoveVel_LeftAndRight.x = DirectionInNormalVec3_LeftAndRight.x
    TempMoveVel_LeftAndRight.z = DirectionInNormalVec3_LeftAndRight.z
```

Since "**DirectionInNormalVec3_LeftAndRight**" is already where we want it we just use it as-is.

So now that we have the velocity directions of the player, we now need to add them together and multiply it by the final walk velocity that we want. The code looks like this:

```
FinalMoveVel =
    (TempMoveVel_FWAndBW + TempMoveVel_LeftAndRight).normalized() *
    FinalWalkVelocity
```

So what we are calculating is the final movement velocity. So first thing we do is add the forward/backward velocity vector with the left/right velocity vector.

Let's say that the player presses both the forward and left movement keys. And let's say that there are the vectors:

```
TempMoveVel_FWAndBW = ( 0.914192 , -0.405281 )
TempMoveVel_LeftAndRight = ( -0.405281 , -0.914192 )
```

The vectors might look like this:

So if we add them together:

We get this:

(0.914192 , -0.405281) + (-0.405281 , -0.914192) = (0.508911, -1.319473)

Which looks like this:

Now, the only problem is that this resultant (the word for the sum of two vectors) is too long. Its current length is about "**1.41**". For information this number, called the "crest factor", read the "Normals Explained > Normals" chapter.

So now let's go to the next step in the line of code: the normalization.

(TempMoveVel_FWAndBW + TempMoveVel_LeftAndRight).normalized()

(0.508911 , -1.319473).normalized() = (0.359854 , -0.933008)

Then after that we multiply the final movement vector by the movement velocity. Let's say that the final movement velocity is "**10**":

(0.359854 , -0.933008) * FinalWalkVelocity

(0.359854 , -0.933008) * 10 = (3.598545, -9.330085)

This final result means that if the player were to keep going in this direction for 1 second the position of the player after that 1 second would be "**(3.598545, -9.330085)**". Remember that in "**move_and_slide()**" the movement vector (the first argument) doesn't need to be multiplied by "**delta**" as that is taken care of in the function.

VERTICAL MOVEMENT

Now we are going to be calculating the vertical velocity. There is much more going on here than in the horizontal velocity section as we have many things to check and there are three states of vertical movement: falling, jumping, and on the floor.

ON FLOOR

The first part we are going to look at is the "**On Floor**" section. The first lines are:

```
if(State_OnFloor and State_Falling and not State_Jumping):
    Falling_IsFalling = false
```

So this first "if" statement looks to see if the player is:

1) On the floor.
2) Falling.
3) Not jumping.

If these things are the case then we simply state that the player is no longer falling. This "if" statement is basically for when the player hits the floor after falling. When that happens we say that the player has stopped falling.

```
elif(not State_OnFloor and not State_Jumping):
    State_Falling = true
```

If the first "if" statement doesn't go through then we go on to this one which checks to see if the player is:

1) Not on the floor.
2) Not jumping.

If these statements are correct then the player must be falling. So we set the state as such.

```
elif(State_OnFloor and State_Jumping and not State_Falling):
    State_Jumping = false
    Jump_CurrentVel = 0
    State_Falling = true
    Falling_Speed = 0
    Falling_CurrentTime = 0
```

If the last couple of statements aren't true, then we check to see if:

1) The player is on the floor.
2) The player is jumping.

3) The player is not falling.

If these statements are correct then the player may have landed on a platform while jumping onto it, but without falling onto it. Confusing? This is just in case the player doesn't actually fall onto a platform or floor, but in stead jumps onto it from the side and just happens to touch the floor while going up. This case is for the off chance that might happen.

So that case tree took care of what happens when the player hits the floor. Now we move on to when the jump action is pressed.

JUMP PRESSED

Now we're in the "**JUMP PRESSED**" section.

```
if(Pressed_Jump and Jump_Released):
```

First thing we check is not only if the jump action was pressed, but we also check to see if the jump action was previously released. This is so the player can't keep holding down the jump action and keep jumping.

```
if(State_OnFloor):
	State_Jumping = true
	State_Falling = false
	Jump_CurrentTime = 0
```

Now we check to see if the player is on the floor. If he is then we:

- Set the "jumping" state to true.
- Set the "falling" state to false.
- And reset the jump timer to 0, which is the beginning of the jump.

JUMPING

In this section we take care of the actual jumping of the player, instead of just checking and setting states.

```
if(State_Jumping and not State_Falling):
	if(State_OnFloor):
		if(Jump_Vel <= Falling_Gravity):
			State_Jumping = false
			Jump_CurrentVel = 0
			State_Falling = true
			Falling_Speed = 0
```

```
Falling_CurrentTime = 0
```

First we check if the player is in the “jumping” state and not falling. If these are correct, then we check if the player is on the floor. If so, then we check to see if the jump velocity is can even pull the player up against the pull of gravity.

If the jump velocity isn’t enough to pull the player up, then what we basically do is cancel the jump by setting all the appropriate states and variables. This is another “just in case” statement.

To cancel the jump we:

- Set the “jumping” state to false.
- Set the jump velocity to 0.
- Set the “falling” state to true.
- Set the falling speed to 0.
  - The falling speed will be modified later in the script, in the “**FALLING**” section, allowing the player to still be affected by gravity when on the floor.
- Reset the falling timer.

```
elif(Jump_Released):
        Jump_CurrentVel = Jump_Vel
        FinalMoveVel.y = Jump_CurrentVel + -Falling_Gravity
```

This “elif” statement occurs if the jump velocity is enough to pull the player up against gravity and the jump action has been previously released.

If it has it sets the initial applied jump velocity to the “**Jump_CurrentVel**” variable, which is the velocity that will actually be applied to the player. It is attenuated as the player reaches the peak of his jump.

After that we set the final movement velocity to be negated gravity plus the current jump velocity. “**Falling_Gravity**” is negated because it’s a positive number. This is because it represents the power of gravity, not necessarily the direction. So, in theory, if you wanted the player to be pulled up instead of down, we would just invert the applied gravity.

```
elif(not State_OnFloor):
        if(Jump_CurrentTime < Jump_Length):
```

So if none of the previous statements are true, then we check to see if the player is not on the floor. If not, in the first “if” statement we check to see if the jump timer has reached the end of the jump length. If the jump isn’t over yet we move on to:

```
if(State_OnCeiling):
        Jump_CurrentVel = 0
        Falling_Speed = 0
        Falling_CurrentTime = 0
        State_Jumping=false
        Falling_IsFalling = true
        State_Falling=true
else:
        Jump_CurrentVel =
                Jump_Vel - (Jump_Vel * ( pow( Jump_CurrentTime / Jump_Length , 2.0 )))
        Jump_CurrentTime += delta
        FinalMoveVel.y = Jump_CurrentVel + -Falling_Gravity
```

If the player hits the ceiling with his head while in the middle of a jump, we cancel out the jump like we would in the previous section if the jump velocity isn't enough to get the player off the ground.

Otherwise, if the player isn't hitting the ceiling, we calculate the current jump velocity, advance the jump timer, and apply the final vertical velocity.

Let's look at an example of how we calculate the jump velocity. We calculate the jump velocity as such:

```
# Hypothetical values.
Jump_Vel = 15.19
Jump_CurrentTime = 0.5
Jump_Length = 1.0

# The line of code. #
Jump_CurrentVel =
        Jump_Vel - (Jump_Vel * ( pow( Jump_CurrentTime / Jump_Length , 2.0 )))

# In real numbers. #
Jump_CurrentVel =

        # First we put in the real numbers.
        15.19 – (15.19 * ( pow( 0.5 / 1.0 , 2.0 ) ) )

        # Then we calculate the first argument of the "pow()" function.
        15.19 – (15.19 * ( pow( 0.5 , 2.0 ) ) )

        # Then we calculate the "pow()" function.
        15.19 – (15.19 * 0.25 )

        # Lastly we calculate what's in the parentheses and subtract that from "15.19".
        15.19 - ( 3.7975) = 11.3925
```

Let's walk through this, even though we already have in the above example.

```
15.19 – (15.19 * ( pow( 0.5 / 1.0 , 2.0 ) ) )
                    ^^^
                ( 0.5 / 1.0 ) = 0.5
                    ^^^
15.19 – (15.19 * ( pow( 0.5 , 2.0 ) ) )
```

First we calculate how long the jump has been going on in comparison to how long the jump should be. That's what the "**0.5 / 1.0**" is. It represents "**Jump_CurrentTime / Jump_Length**", which gets us the percentage of how far we are into the jump.

```
15.19 – (15.19 * ( pow( 0.5 , 2.0 ) ) )
                ^^^^
            pow( 0.5 , 2.0 ) = 0.25
15.19 – (15.19 * 0.25 )
```

Then we calculate the "**pow()**" function which gives us "**0.25**". And of course we use a "**pow()**" function here so that the jump velocity isn't too linear. Try making the jump linear to see how it looks, if you're curious.

```
15.19 – (15.19 * 0.25 )
15.19 – 3.7975 = 11.3925
```

And then we just do the rest of the line and we have our current jump velocity.

And again, after we get the jump velocity, we just add the current delta to the jump timer and put the current jump velocity into the final movement velocity vector that is used in "**move_and_slide()**".

Now let's move on to the next case that executes if the jump is over:

```
# Otherwise, if the jump is over...
else:
    Jump_CurrentVel = 0
    Falling_Speed = 0
    Falling_CurrentTime = 0
    State_Jumping=false
    Falling_IsFalling = true
    State_Falling=true
```

If the jump is over we cancel out the jump as we have previously when the jump velocity isn't enough to pull the player off the ground and the player hits a ceiling.

FALLING

Here we take care of when the player is falling. Note that if the player is on the ground and isn't literally falling, he is still considered to be "falling", as he is not jumping. When the player is on the ground the vertical velocity of the player is "-(**Falling_Gravity * Falling_Speed_Multiplier_Default)**", which by default would be "-(**9.8 * 0.25) = -2.45**". So that means whenever the player is standing still or walking on a perfectly level floor he is being pulled down at 2.45 m/s. If the player was to be pulled down anymore than that he would slide down on slopes when standing still.

On to the code:

```
if(State_Falling and not State_Jumping):
```

First we check if the player is falling and not jumping.

```
if(not State_OnFloor):
	if(Falling_Speed < Falling_TerminalVel):
```

Then we check to see if the player is not on the floor. If he isn't then we go on to check if the current falling speed of the player is less then terminal velocity. If it is we go on to this code, here:

```
if(not Falling_IsFalling):
	Falling_IsFalling = true

	Falling_CurrentTime = 0

elif(Falling_IsFalling and Falling_CurrentTime < Falling_TimeToHitTerminalVelSec):
	Falling_CurrentTime += delta

	Falling_Speed = Falling_TerminalVel *
			pow(Falling_CurrentTime / Falling_TimeToHitTerminalVelSec, 0.4)

	if(Falling_CurrentTime >= Falling_TimeToHitTerminalVelSec):
		Falling_CurrentTime = Falling_TimeToHitTerminalVelSec

else:
	Falling_CurrentTime = Falling_TimeToHitTerminalVelSec

Falling_Speed = Falling_TerminalVel *
		pow(Falling_CurrentTime / Falling_TimeToHitTerminalVelSec, 0.4)
```

The first "if" statement checks to see if the player hasn't started falling yet. This happens when the "**State_Falling**" bool is first switched to true.

If the player just started falling then we set the "**Falling_IsFalling**" bool to true to say that the player has started falling, and then we reset the falling counter to 0 to signify the start of the falling process.

Otherwise, in the second statement, which is an "elif" statement, we check two things:

1) Has the player started falling?
2) Is the current falling time less then the time it takes to hit terminal velocity?

If these are both true then we advance the falling timer. In that same "elif" statement we check to see if the falling timer is past its limit. If it is we set it back to its limit so as to not go over by a tiny amount, which can cause a little "jump" when the player hits terminal velocity as the velocity goes above terminal velocity and then back down quickly in the next loop.

The last statement, which is an "else" statement, executes when the falling timer has passed its limit. If that happens then we set it back down to its limit.

After these statements, we finally calculate the falling speed, using a "**pow()**" function to have a non-linear speed increase.

```
# Otherwise, if the player is on the floor...
else:
	# If the player is moving...
	if(FinalMoveVel.x != 0 or FinalMoveVel.z != 0):
		Slope_AffectSpeed()
	else:
		Falling_Speed_Multiplier = Falling_Speed_Multiplier_Default

	Falling_Speed = Falling_Gravity * Falling_Speed_Multiplier
```

So, what happens if the player *is* on the floor?

First we check to see if the player is moving. If he is then we execute "**Slope_AffectSpeed()**" (click link to go to chapter explaining how it works).

Otherwise, if the player is not moving we set "**Falling_Speed_Multiplier**" to "**Falling_Speed_Multiplier_Default**", which is "**0.25**" by default.

After that we calculate the final falling speed using that multiplier.

```
FinalMoveVel.y = -Falling_Speed
```

After all these "if" and "elif" statements we apply the falling speed to the Y axis of the final movement velocity, making sure to negate it so that the player falls down and not up.

FINAL MOVEMENT APPLICATION

Now we're getting to the end of the script. Here we actually start applying all the calculation we have done up to this point.

MOUSE ROTATION

Here we rotate the player and his camera. Check the "**_unhandled_input()**" section for more information.

```
if(Mouse_Moved):
        self.rotate_y(deg2rad(-Mouse_Rel_Movement.x) * Cam_RotateSens)
        Node_Camera3D.rotate_x( deg2rad(-Final_Cam_Rot_Local_X) *
                                        Cam_RotateSens)

        Mouse_Moved = false
```

First we check if the mouse was even moved this frame. If it was we rotate the Y axis of the whole player according to the relative mouse movement on the X axis. Let's look at this line more closely:

```
self.rotate_y(deg2rad(-Mouse_Rel_Movement.x) * Cam_RotateSens)
```

So first we have "**self.rotate_y()**". The "self" referred to here is the node that the script is attached to. And "**rotate_y()**" is a function of "**Spatial**", one the ancestors of "**KinematicBody**". There's no need to worry about physics when rotating the character on the Y axis because the collision shape of the player is a capsule. So if the player rotates he won't hit anything accidentally like he might if he had a cube collision shape:

At any rate we rotate the player on the Y (vertical) axis. We convert the X axis of "**Mouse_Rel_Movement**" to radians so that the mouse movement has more accuracy.

Let me explain. "**rotate_y()**" takes an angle in radians as its only argument. If the mouse moved only 1 pixel on the screen, then after being multiplied by "**Cam_RotateSens**" it would equal "**0.25**" radians. That means that the player would rotate 0.25 radians (14.3 degrees) when the player only moved the mouse one pixel. That's way too much.

So instead we make the relative motion of the mouse represent degrees. So if the player moved the mouse only 1 pixel, after being multiplied by "**Cam_RotateSens**" it would equal "**0.25**" degrees, instead of radians, therefore having greater accuracy.

We could change all of this by just multiplying the relative mouse movement by a smaller number, but I've already written the script. And besides that I don't think it would have any advantage to performance and I like to try to keep the "**Cam_RotateSens**" below 1.0 but not too low.

After we rotate the whole player, now we need to rotate the camera up and down. We do this the same way we rotate the whole player, but when we specify the amount to rotate the camera we have to negate it as moving the cursor down provides a positive number, and vice versa.

Then we set "**Mouse_Moved**" to false to say that the mouse movement has been taken care of and to not rotate the character anymore until a new mouse movement event occurs.

LINEAR MOVEMENT

Here we finally apply the linear velocities that we've calculated in the previous sections.

```
if(State_OnFloor):
        FinalMoveVel += get_floor_velocity() * delta

move_and_slide(FinalMoveVel, FloorNormal, SlopeStopMinVel, MaxSlides,
                        MaxFloorAngleRad)
```

The first thing we do is find out if the player is standing on something. If he is we add the floor's velocity to the player. Otherwise, if the player is not on the floor this calculation never takes place.

After that we apply the final movement velocity vector with all the previous calculations. And that's it.

GET STATES

After we move the player, we still have some things left to do. First thing is we need to get the new states of the player, as they're updated whenever "**move_and_slide()**" is executed.

```
State_OnFloor = is_on_floor()
State_OnWalls = is_on_wall()
State_OnCeiling = is_on_ceiling()
SlideCount = get_slide_count()
Player_Position = translation
```

STEPS

All we do here is execute "**Step_PlayerUp()**". Click the link to find out more about this function.

```
Step_Player_Up()
```

CAMERA INTERPOLATION

Here we check the progress of any camera interpolation that is taking place and execute "**InterpolateCamera()**". Click the link more for information.

```
if(CamInterpo_DoInterpolation == true):
        CamInterpo_CurrentTime_Secs =  InterpolateCamera(
                                                CamInterpo_StartingPos_Local_Y ,
                                                CamInterpo_CurrentTime_Secs ,
                                                delta
                                                        )
```

So first we check if the "**CamInterpo_DoInterpolation**" bool is true to find out if we need to do it. If it we execute "**InterpolateCamera()**", making sure to have it give the return value to "**CamInterpo_CurrentTime_Secs**" so that we know how long the interpolation has been going on.

TOUCH FUNCTION

All we do here is execute "**Touch_CheckAndExecute()**". Click the link for more information.

```
Touch_CheckAndExecute()
```

DEBUG

The last section.

```
Debug_Label_String = "FPS: " + str(Engine.get_frames_per_second())
Debug_Label.set_text(Debug_Label_String)
```

We set the debug label string as the FPS. Then we set the label according to the debug label string variable. This section isn't necessary but can be useful.

Section 3: Math Terms and Functions Explained

This section is for math terms and functions that might need explaining.

Chapter 1: lerp()

"**lerp()**" is the linear interpolation function. The first argument is the "**from**" variable and the second is the "**to**" variable. The third argument is the "**weight**", using a normalized value, that you want to interpolate between the two.

As an example, let's say we have a "**lerp()**" function that looks like this:

Result = lerp(0.5 , 1.5 , 0.5)

"**Result**" will equal "**1.0**". This is because the third argument, called the "weight", is "**0.5**". And since this is a normalized value, what it really means is "**50%**".

Since "**50%**" is halfway between "**0%**" and "**100%**", that means that we want to find what's halfway through "**0.5**" and "**1.5**". As you probably already figured out it's "**1.0**".

Another way of looking at it is by this formula:

Result = (from + (weight * (to – from)))

And in real numbers, according to our example:

Result = (0.5 + (0.5 * (1.5 – 0.5))) = 1.0

And if we were to have a weight of "**0.25**":

Result = (0.5 + (0.25 * (1.5 – 0.5))) = 0.75

And if this was visualized (on next page):

It would look like this. Of course, you can just use "**lerp()**" without all the manual math, but it's helpful to have an idea of what's going on under the hood.

You can also use a "**pow()**" function, explained in the next chapter, as the weight argument to get a non-linear result.

Chapter 2: pow()

"**pow()**" is simply the exponential function. It works like this:

pow(4 , 2) = 16

So in this example we take "**4**" and multiply it by itself twice, so that it equals "**16**", as "**4*4=16**".

You can also do this:

pow(4 , 2.5) = 32

So what we are doing here is multiplying "**4**" by itself twice, and then we multiply that result by take half of "**4**", which is, of course, "**2**". It looks like this:

pow(4 , 2.5) = 4*4*2 = 32

The second argument, the exponent, can be anything you want. Let's say you do this:

pow(4, 0.5) = 2

Did you see what happened? We were able to be the square root of "**4**" by using an exponent, which in this case is "**0.5**". Another example:

pow(4 , 0.25) = 1.414213562

Okay, this is useful. But what can it be used for? It can be used for many things, but in this script we use it to make a linear falloff into a smooth falloff.

Let's take a look at some examples. Let's say we have a car. And whenever the player presses on the accelerate button, it takes some time to get to max speed. Let's look at some variables:

CarSpeed = 0.0
MaxSpeed = 120
TimeToMax = 25
CurrentTime = 0.0

They should be self-explanatory. The last variable, "**CurrentTime**", is how long the player has been pressing the accelerator button. Here would be the code that we are using right now:

CarSpeed = (CurrentTime / TimeToMax) * MaxSpeed

So let's look at a real-number example:

CarSpeed = (12.5 / 25) * 120 = 60

This code can be visualized as a graph which looks like this:

As you can see, the acceleration of the car's speed is a straight shot from 0 to 120. That's okay, but let's make it better like this:

CarSpeed = pow((CurrentTime / TimeToMax) , 0.5) * MaxSpeed

And in a real number example:

CarSpeed = pow((12.5 / 25) , 0.5) * 120 = 84.85

Which would look like this visualized:

As you can see, the acceleration is quite different now, with the car accelerating quickly in the first few seconds, but then taking much longer to get to max speed after that. This is more how cars accelerate, and you can simply adjust the exponent to whatever you want to edit the acceleration of the car.

Let's look at another example, this time using a exponent larger then 1:

```
result = pow( x , 2 )
```

Which would look like this, visualized:

As you can see, the curve is now in the opposite direction. The script uses this kind of falloff for when the player jump. When the player presses the jump button, the velocity starts out at full velocity, according to whatever "**Jump_Vel**" is. Then over time the curve above is subtracted from the max jump velocity. It looks like this in the script:

```
Jump_CurrentVel = Jump_Vel - (Jump_Vel * ( pow(Jump_CurrentTime / Jump_Length, 2) ))
```

So a real-number example would look like this:

```
Jump_CurrentVel = 15.19 - (15.19 * ( pow(0.5 / 1.0 , 2) ))

#Simplified.
Jump_CurrentVel = 15.19 - ( 15.19 * ( 0.25 ) )

#Simplified.
Jump_CurrentVel = 15.19 – 3.7975

#Simplified.
Jump_CurrentVel = 11.3925
```

So, in the above example, the current jump time is "**0.5**", or half way through the jump. So, we take that and multiply it to the second power. That gives us "**0.25**". Then we multiply that by the max jump velocity, which gives us "**3.7975**". Then, lastly, we take the max jump velocity and subtract "**3.7975**" from it to get the current jump velocity.

Over time, the jump velocity tapers off, so that as the player reaches the peak of the jump, he comes to a gradual stop in the air instead of going up and then sharply start falling down.

So, as with our example, it would look like this:

And that's about it.

Section 4: Normals Explained

I really recommend looking up about normals, vectors, and anything else explained here in other places so as to help you understand these things better, as I'm still learning how to teach.

You can check out the Godot documentation concerning vectors, transforms, and matrices here:

http://docs.godotengine.org/en/stable/tutorials/math/index.html

If that link doesn't work go to the Godot documentation and look up "Math" or "Vector Math" and go from there.

You'll need to know about coordinate systems, like "x" and "y" positions, to understand this section.

This section talks about 2D and 3D vectors as used in GDScript and in 2D and 3D graphics. Vectors as used in programming languages are similar but different. There's no need to worry about them now. Just remember that this is all for use in GDScript and Godot.

The first chapter is on the next page (because I can't fit the first picture on this page).

Chapter 1: Coordinate System

You probably already know what a coordinate system is, but I'll go over it just in case. First off, let's look at this picture:

A coordinate system is basically just a grid that represents positions.

Let's look at the upper right portion of the grid:

In this picture we see a bunch of numbers above and to the right of the grid. These represent the positions of the lines.

In the above picture there is a dot. The dots position is represented as "**(0.9 , 0.2)**". Let's look at another picture that will help us see what the position of the dot is.

We see the red lines intersect at the blue dot. If you follow the lines to the numbers, you will see that they end at certain numbers. The numbers are "**0.2**" and "**0.9**".

A coordinate, or position, is represented like this: **(0.9 , 0.2)**

The first number, "**0.9**", is on what's called the "X" axis. It tells us the horizontal position of the dot. The second number, "**0.2**", is on what's called the "**Y**" axis. It represents the vertical position of the dot. So "**(0.9 , 0.2)**" means that the dot is 0.9 units to the right, and 0.2 units up.

The position numbers don't mean anything in particular. That is, they don't represent meters, feet, inches, or anything else unless specified. They are arbitrary.

So the dot in the above picture is located 0.9 units to the right and 0.2 units up.

But what happens when one of the numbers is negative?

The first number of the blue dot's position is "**-0.3**". This means that it's 0.3 units to the left. In coordinate systems a negative "**-**" symbol just tells us to go in the opposite direction. More on this later.

So if the second number was negative what direction would the dot go?

That's right. It would go down.

So what does a coordinate system look like in 3D? Like this:

All we do is just add one extra number, and it represents the new "**Z**" axis. In Godot the X axis goes east to west, the Z axis goes north and south, and the Y axis goes up and down. But in other

programs, like Blender, the Z axis is the one that goes up and down, and the Y axis goes north and south. You'll get used to it, just keep trukin'.

In most 3D programs the X axis is represented by red, the Y axis is represented by green, and the Z axis is represented by blue. So it would look like this in Godot:

So to reiterate, the Y axis is vertical in Godot. And as a side note the white grid lines represent 1 unit of length. I just think of these units as meters, because it's easier to work with when I'm trying to make real world measurements of characters, buildings, etc.

Next let's look at this picture:

In this picture we have a red box. The center of the box, where the dot is, represents the origin of the box. The origin of the box is the position that the box holds.

The position of the box is relative to the origin of the scene. It's called an **offset**. It represents the position of the cube in relation to the center of the scene. It's almost as if you were moving the red box from "**(0, 0)**" to "**(0.4 , 0.7)**". It might look like this (on next page):

In this picture the box position is also called its offset because the box is “offset” from the center of the scene by the amount specified in the parentheses.

This is called **global space**. It means that the vector that is given is in relation to the center of the whole scene.

The center of the scene in Godot looks like this:

So anything in “global space” is in relation to this origin.

The other kind of “space” is **local space**. This is a vector that represents a position relative to another object.

Let’s take a look at another example picture:

In this picture, notice how the coordinates of the boxes are still in relation to the origin of the whole scene.

Now let’s look at that picture again, but instead of the origin of the scene being the center of the graph, let’s make the red box the center of the graph:

This is what is considered local space. Whenever you get a vector coordinate in local space, you get a vector that is relative to the object you are asking for it from. So if you ask for the position of the

blue box in local space from a script that is attached to the red box, you will get "**(0.3 , -0.4)**" because the blue box is 0.3 units to the right of the red box and -0.4 units to the south of the red box.

If you want to get the local coordinates of the blue box from the red box's script, you could do this in GDScript:

```
var blue_box_pos = get_node("BlueBox").to_local( blue_box_pos.get_transform().origin )
```

And that will give you the local coordinates of the blue box from the red box.

But let's say you want to get the local coordinates of the red box from the blue box's script. What would they look like? It would be the same but in the opposite direction:

```
red_box_pos = (-0.5 , 1.0)
```

It's the exact opposite because you are calling it from the other object now.

Chapter 2: Vectors

There are basically two kinds of vectors: a “programming vector” and a “Euclidean vector”.

In programming languages like C++, Python, or GDScript a “vector” is a variable that can hold 2 or more numbers and can be resized at any time. And by “resize” I mean that the computer can change how many numbers the vector can hold while the program is running.

But in trigonometry (and therefore 2D and 3D math) a “Euclidean vector” is basically an arrow that represents a direction and length in 2D or 3D.

Programming Vector

As far as we’re concerned right now, a programming vector is simply a variable that holds a coordinate in two or three numbers. In GDScript it looks like this:

```
var 2D_Vector = Vector2( 0.1 , 0.5 )
var 3D_Vector = Vector3( 0.5 , 0.2 , 0.7 )
```

If you were to print the “**2D_Vector**” variable onto the console it would look like this:

```
( 0.1 , 0.5 )
```

This is simply a coordinate. It just tells us a position, like the ones we used in the “Coordinate System” section.

Euclidean Vector

A Euclidean vector is a little different. It’s a component of trigonometry. In trigonometry a vector is basically an arrow that represents a direction and a length. Let’s look at the picture on the next page:

The blue arrow is what is called a “Euclidean vector”. Let’s call it “**Vector_01**”. In GDScript this vector would look like this:

```
vec Vector_01 = Vector2( 0.91 , 0.39 )
```

Do you see why this is confusing? Because this is the same as what I called the “programming vector”. But a “programming vector” and a “Euclidean vector” are still technically different. Just remember that when we create a vector in GDScript using “**Vector2()**” or “**Vector3()**” we are simply creating a variable that holds two or three numbers that can represent a Euclidean vector or a position.

In the previous picture the blue arrow is a vector. It starts at “**(0.0 , 0.0)**” and goes to “**(0.91 , 0.39)**”. But in the code the vector only holds one position:

```
vec Vector_01 = Vector2( 0.91 , 0.39 )
```

How can this be an arrow with only one position? The vector always starts at the origin, so there’s no need to specify the starting point.

Also note that in the picture it says that the arrow has a length of “**1**”. If we were to draw a line from the origin to the position specified in the vector above we would get a Euclidean vector. Then if you got an imaginary ruler and measured the line, it would be 1 unit long.

Normals

A normal is simply a vector with a length of 1. Let’s go back to the previous picture again:

On the bottom of the picture it mentions that the vector has a length of 1. When a vector has a length of 1 it's considered to be a "normal".

Let's say you have a vector that has a length of 2.5. But you want to make it have a length of 1. How can you do so? By a method called "normalization". Check the "Normalization" chapter for more information on how to normalize your vectors.

The reason you might want a vector to have a length of 1 is so you can find out the direction something is pointing, and if it has a vector length of 1 you don't have to worry about getting incorrect results. And if you ever need to make it longer, shorter, or add another vector to it you can. For instance in order to get a dot product between two vectors they both need to have a magnitude of 1. You could then use the dot product to get the angle between two vectors.

Another reason you might want a normalized vector is for use with player velocity. Let's say you have a player that presses the forward button. In order to find out which direction to move the player you need to get the direction the player is facing. In this script you do that by getting the Z axis normal of the transform:

```
get_global_transform().basis.z
```

(Continued on next page)

Which might look like this:

In this example picture, the Z axis normal of the player is "**(-0.425209 , 0.904983)**". When getting the transform normals they always have a length of 1 unless they have been manually manipulated. If you were to find out the length of this vector with the Pythagorean theorem (discussed at the beginning of the Normalization chapter) we would find the length to be 1.

And from these vectors we are able to find out the angle at which the player is facing in global *or* local space. So if we were to get the dot product of this normal against a vector that points straight north:

```
DotProduct = Vector2(-0.425209 , 0.904983).dot( Vector2(0.0 , 1.0) )
DotProduct =  0.904983
```

Which looks like this (on next page):

And if we were to get the angle of the player's Z axis normal in radians:

Angle_Radians = 0.439457

We would get "**0.439457**" radians, or ~25 degrees. In Godot when rotating an object a positive number is CCW and a negative number is CW. I don't know why this is but I'm sure there's a reason.

Anyway, that's vectors and normals.

Chapter 3: Transforms

Transforms are related to vectors/normals as in they are simply the X, Y, and Z (if 3D) coordinates of the normals of a given object.

So let's say we have a 2D object and this is its transform matrix:

	X	Y	Offset
X	1.00	0.00	0.00
Y	0.00	1.00	0.00

The X and Y rows that are colored red and green are the normals. And the X and Y columns that are gray are the X and Y position of those normals. The "Offset" column tells us how much to add to each axis after everything else is set.

Let’s look at it in visual form:

In this picture, the two normals are represented as a red and green arrow for the X and Y normal, respectively.

And the coordinates of the normals are shown as "**X Normal: (1.0 , 0.0)**" and "**Y Normal: (0.0 , 1.0)**". The rows represent the "arrow", or normals, and the columns represent the position of these normals.

The last column is the offset of all the normals together. So let's say the matrix looked like this:

	X	Y	Offset
X	1.00	0.00	3.00
Y	0.00	1.00	1.00

This is the same as the last image, but instead of the player being in the center of the scene, he is now 3 units to the right and 1 unit north, like this:

So when the offset number is in the X axis row, which in this example is "3.0", that means we add "3.0" to all the normals. In Godot an objects transform matrix is in relation to its parent node.

One last example. When an object is rotated, what does the matrix look like? Let's say the object is rotated 25 degrees clockwise. It would look like this:

	X	Y	Offset
X	0.906121	-0.422618	0.000000
Y	0.422618	0.906308	0.000000

And in visual form it would look like this:

In 3D we just add one extra row and column:

	X	Y	Z	Offset
X	0.906121	0.000000	-0.422618	0.000000
Y	0.000000	1.000000	0.000000	0.000000
Z	0.422618	0.000000	0.906308	0.000000

And it would look like this in 3D (on next page):

Y Normal:
(0.0 , 1.0 , 0.0)
Z Normal:
(0.422618 , 0.0 , 0.906308)
X Normal:
(0.906121 , -0.422618)

Chapter 4: Normalization

Normalization is simple: it changes the magnitude (length) of a vector to 1.0. This is so it can be used in any way you want by just multiplying, manipulating, or comparing the vector against other vectors or scalers.

It goes like this: let's say we have a 2D vector called "**2D_Vector_01**" that is "**(2.5, 7.5)**".

But we want to dot product this vector to another one called "**2D_Vector_02**" that is "**(0.5, 0.1)**". How do we do this? If we didn't normalize both of these vectors, the result of a dot product between them would be "**2.0**". This is not what we need.

So what we do is normalize both vectors. We do that in Godot by simply by calling "**normalized()**" from the vector we want to normalize. Example:

```
2D_Vector_01 = 2D_Vector_01.normalized()
```

Pythagorean Theorem

But what is going on behind the scenes? It's simple. Let's take our first vector of "**(2.5, 7.5)**". We need to get the magnitude (length) of the vector. So we use the **Pythagorean theorem** to get it. It goes like this:

$$\textbf{magnitude} = \sqrt{x^2 + y^2}$$

Which in this case would be:

magnitude = $\sqrt{2.5^2 + 7.5^2} = \sqrt{6.25 + 56.25} = \sqrt{62.5} = 7.90569415$

The variables "**x**" and "**y**" always need to be positive numbers.
The Pythagorean theorem actually just looks like this:

a2 + b2 = c2

If "a", "b", and "c" were the lengths of the three sides of a triangle. But in order to get the length of a vector, we do just as we have done in the previous code snippets. I just wanted to let you know what the Pythagorean theorem actually was.

We could of also used "**Vector01.length()**" to get the magnitude of the vector, but we wouldn't of learned as much.

Anyway, we now know how long the vector is. But what we want is for it to be "**1.0**". We get this by simply taking each element of the vector and divide them by the magnitude, like this:

normalized_vector = (2.5 / 7.90569415 , 7.5 / 7.90569415) = (0.316227766 , 0.948683298)

So now our normalized vector is "**(0.316227766 , 0.948683298)**". And if we were to get the length of that vector it would simply be "**1.0**". Awesome.

If we normalized "**2D_Vector_02**" (that is "**(0.5, 0.1)**"), it would be "**(0.980581, 0.196116)**". *Now* we can dot product these two vectors, which would equal "**0.496139**", which is what we want. If we wanted to we could get the angle between these two vectors by using the following code:

angle = rad2deg(acos(2D_Vector_01.dot(2D_Vector_02))) = ~60.255 degrees

Not let's look at a new example. Let's say that the length of a vector was something like this:

This is already normalized, right? Because the vector's coordinates are in the range of 0.0 to 1.0? No, it isn't normalized because the *length of the vector is not 1*. What is the length of the vector? Let's find out:

magnitude = $\sqrt{(0.84^2 + 0.86^2)}$ = 1.202164714

So the length of our vector is actually "**1.2**"… Well, then, let's make it "**1.0**":

normalized_vector = (0.84 / 1.202~ , 0.86 / 1.202~) = (0.698739524 , 0.715376179)

Chapter 5: Dot Product

Practically speaking, the dot product simply states how much alike one vector is to another. So let's imagine two vectors:

To make this easy, these two vectors have the same length of 1. When we do a dot product between these two vectors, we are asking how much alike these two vectors are in both length and direction.

Since these vectors are normalized, when we do a dot product on these vectors, we are pretty much just asking how much alike are their directions.

We do this in Godot by simply using "**dot()**" inside one of the vectors you are going to use in the dot product, like this:

```
DotProduct = Vector_A.dot(Vector_B)
```

But what is happening when we do this? There are actually two ways of calculating a dot product, but I will only show you one (the easier one). It goes like this:

```
DotProduct = (Vector_A.x * Vector_B.x) + (Vector_A.y * Vector_B.y)
```

We multiply the X's, multiply the Y's, then add them together.

In the example above, it would go like this:

```
DotProduct = (0.60 * 0.87) + (0.79 * 0.47) = 0.522 + 0.3713 = 0.8933
```

So our dot product is "**0.8933**". Okay, that's great, but so what? Well, with this we can find the angle between these two vectors.

To do so in Godot, we simply need to use "**acos()**".

Angle = acos(0.8933) = 0.466161766

Note that every program/calculator may use either radians or degrees when using trigonometry functions. In Godot they will always use radians. Depending on the program/calculator, if you use a trigonometry function it may use angles in radians or degrees. You'll have to find out. On my calculator program on my computer I can change the angle units used to use either degrees or radians. Always make sure of what you are using.

The result "**0.466161766**" is in radians, so if we convert that into degrees either using Godot's built-in function…

Angle_Degrees = rad2deg(0.466161766) = 26.709101783 degrees

Or you can just do it yourself (slower; just use the built-in function):

Angle_Degrees = 0.466161766 * (180/PI) = 26.709101783 degrees

So the angle between these vectors is about 26.7 degrees. Cool.

But what if we had two vectors like this:

How much alike are these two vectors? Let's find out. Our formula is this:

DotProduct = (Vector_A.x * Vector_B.x) + (Vector_A.y * Vector_B.y)

And the real numbers are:

DotProduct = (0.0 * 1.0) + (1.0 * 0.0) = 0.0 + 0.0 = 0.0

Do you see how it gets the the similarity between the two vectors? The X axis position on the first vector is "**0.0**" and we multiply that against the X axis of the second vector. But it doesn't matter what the second vector's X axis is because anything multiplied by 0 is always 0. The same with the Y axis, in this example. It's get multiplied by the second vector's Y axis, which is "**0.0**".

So, really, when we use the dot product what we are asking is "How similar are vector 1 and vector 2?" And in this case the answer is "They are not at all similar," or in mathematical words "0".

So what's the angle between these two vectors?

Angle = acos(0.0) = 1.570796327

And conversion to degrees...

Angle_Degrees = 1.570796 * (180/PI) = 90

We already knew this, but it's nice to see it done with math, so we can see how it

One last example:

So how similar are these two vectors? They're exactly the opposite of each, they seem to have nothing in common. Wouldn't they have a dot product of "**0.0**"? Well, think about it. They are exactly

the same, but they are going in exactly the opposite directions. So really, what would their dot product be?

DotProduct = (-0.801838 * 0.801838) + (-0.597542 * 0.597542) = −0.642944178 + −0.357056442 = -1.0

Note that I had to use more precise numbers. I did this because if I used the rounded numbers in the picture I would of gotten "**−0.9881**". So, of course, more precise number equal more precise numbers.

Also, if you were to calculate the dot product above you would actually get "**−1.00000062**". This is okay as this is the nature of programming with floating points in computers. It's not always possible to get *exactly* "**1.0**" when using a dot product because the numbers can only be so precise. But when programming for a video game, it's okay to have less precision, as it is almost never noticed as the numbers are actually quite precise in the first place. The only exception I can think of is maybe something to do with physics. Sometimes errors or glitches occur because of lack of precision, but that's to be expected. If it were any more precise video games would run slowly.

So the result of our dot product is "**-1.0**". Our assumption was kind of right. The vectors weren't anything alike in direction, but they still had the same number but they were negative. So we result in a "**-1.0**" because they are the same, but they are just going in exactly the opposite direction from each other.

So what would the angle of that be?

Angle = acos(-1.0) = 3.141592654

And converting it to degrees…

Angle_Degrees = 3.141592654 * (180/PI) = 180

This comes as no surprise, but the angle of these two vectors is "**180**". Sweet.

3D Dot Product Formula

To get a dot product from two 3D vectors, it goes like this:

DotProduct = (Vector_A.x * Vector_B.x) + (Vector_A.y * Vector_B.y) + (Vector_A.z * Vector_B.z)

Understanding the Dot Product

So what *exactly* is going on? It's confusing! Well, let's say we have these vectors:

Now, in order to visualize what's going on, we will tilt both the vectors at the same time so as to lay Vector A down on the black line, so that it will be "**(1.0 , 0.0)**":

So Vector A will be our "base" vector, the one that will be the first argument of the dot product, or in Godot it will be the vector we will be calling the "**dot()**" function from, like this:

```
DotProduct = VecA.dot(VecB)
```

Imagine there is a line from the end of Vector B going straight down, like this:

Since we've laid the vectors down, Vector A only has a non-zero number on the X axis, which is "**0.972219**". This is the dot product: it basically means how much do you need to multiply Vector A by to get a right triangle. If you were to multiply **"(1.0 , 0.0)**" by "**0.972219**", what would you get? "**(0.972219 , 0.0)**", of course. And if we were to get the length of Vector A after that, it would also be "**0.972219**".

Why is this important? Well, just look at what we now have:

It's a right triangle! Now we can see what angle we've got by using arcsine, arc cosine, or arctangent!

So let's say we want the angle between these two vectors. We have the dot product, which is "**0.972219**".

acos(0.972219) = 0.236265072 radians = 13.536991487 degrees

We get the angle, in radians, between the two vectors. Why is this? Remember that “cosine” simply means the length of the adjacent edge compared to the length of the hypotenuse edge:

That means that the adjacent side is about “**0.97**” the length of the hypotenuse. And remember that all of these sides are in relation to the “**θ**” angle. So if the cosine of “**θ**” is the adjacent’s length divided by the hypotenuse’s length, then that means if we do it backwards, we will get “**θ**”, or the angle of the two vectors.

We use “**arc cosine**” in this case, which simply means “do cosine but backwards”. There is also “**arcsine**” and “**arctangent**”. And, as we have already seen, we just use “**acos()**” to do an arc cosine calculation.

I suggest looking for more tutorials on this subject. It takes a while to understand what is going on, but once you do you have it. Just give it time and come back to this subject again and again until you get it.