

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 2:Global Variables In-Depth.....	25
Chapter 1:Quick Intro.....	3	Explanation (at the Top of the Script).....	25
Chapter 2:Structure of Project.....	4	Notes.....	25
Project Directory Structure.....	4	Extends.....	26
Godot Project Editor Structure.....	6	Settings.....	26
Input Map.....	6	Signals.....	31
Extra Information.....	7	Nodes.....	31
Chapter 3:Design of Main.tscn.....	8	States.....	32
Scripts Inside Main.tscn.....	9	Global Variables.....	33
Debug Scripts.....	10	Input.....	33
Chapter 4:Design of Player.tscn Scene.....	12	Movement.....	34
Player Scene Overview.....	12	Rotation.....	35
Design of Player.tscn.....	13	Falling.....	36
Scripts Inside Player.tscn.....	13	Jumping.....	37
Chapter 5:Design of Base.tscn Scene.....	17	Slope Speed.....	38
Overview of Base.tscn Geometry.....	17	Steps.....	38
Scripted Child Objects of Base.tscn (inside		Camera Step Smoothing.....	40
Main.tscn).....	19	Ray Casting.....	41
Chapter 6:Debug Scenes.....	21	Interaction.....	42
Coll_Sphere.tscn.....	21	Chapter 3:Functions Explained In-Depth.....	43
Pos_Visual_Axis.tscn.....	21	Quick Reference of the Functions.....	43
Section 2:Player.gd Script.....	22	Step_Player_Up().....	44
Chapter 1:Summary of Contents of Player.gd		InterpolateCamera().....	59
Script.....	22	Section 3:Normals Explained.....	60
Global Variables and Constants.....	22	Chapter 1:Normalization.....	60
Functions.....	23	Chapter 2:Dot Product.....	64
Ready.....	23	3D Dot Product Formula.....	67
Unhandled Input.....	23	Understanding the Dot Product.....	68
Physics Process.....	24		

Click a section to go to it.

Section 1: Project Overview

Chapter 1: Quick Intro

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

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

The main purpose of this project is to help people (including myself) learn how to program a player character in a video game. 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 assests that are used in this file.
- **export_presets.cfg**
 - Godot-generated presets for exporting the project.
- **icon.png**
 - The custom icon I made for the project.
- **LICENSE**
 - The license of this project, which is the MIT license.
- **project.godot**
 - The godot project file thing.
- **README.md**
 - The GitHub readme.

Godot Project 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 “Use of Debug Scripts and 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.
 - **Platofmr_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 nessecary.

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 initialy 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)**

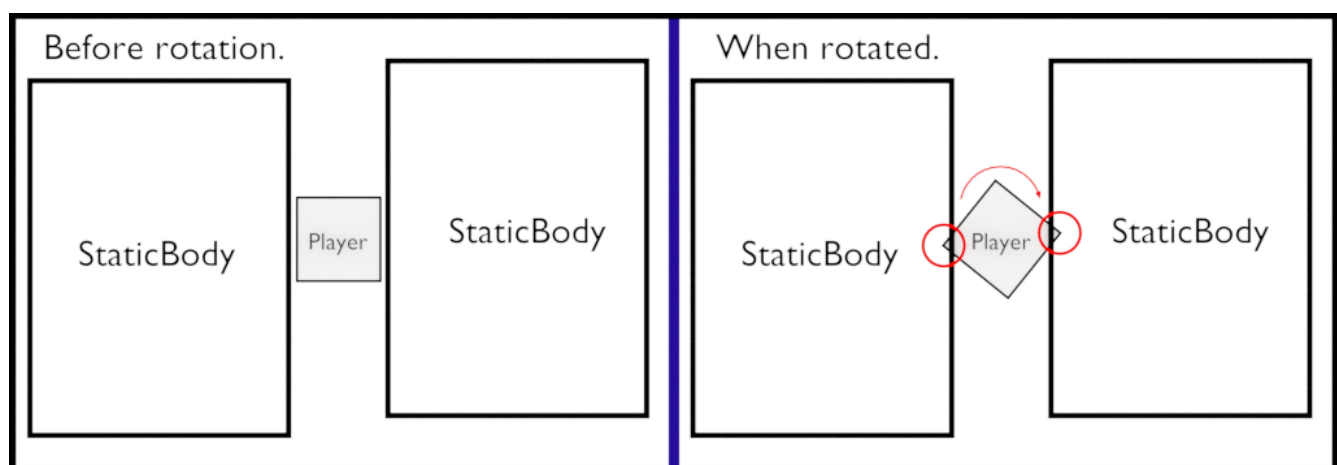
- The cross hair texture node.
- **Crosshair_Useable (TextureRect)**
 - The red circle that shows up when the player is looking at something he can use.
- **DEBUG (Control)**
 - The parent node for all debug information.
 - **Debug_Label (Label)**
 - The text that can be used for debug information, like showing the player's velocity or direction.
 - Can be edited in the player script using "**Debug_Label_String = str(Whatever)**".
 - Can be set in the player script using "**Debug_Label.set_text(Whatever)**".
 - This already exist at the end of the file, so you really don't need to set it twice.

Design of Player.tscn

The design is simple.

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

The "**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:

(continued on next page)

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


Chapter 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.

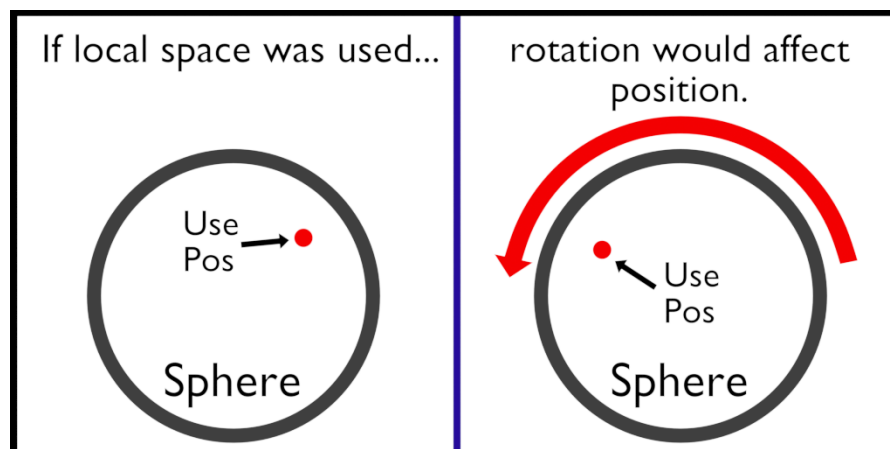
It first set’s up two Vector3 variables, “**Pos**” and “**Impulse**”. Then it set’s up a floating point variable called “**Impulse_Mul**” and set’s the impulse amount according to the object’s mass as set in the node’s settings. “**Pos**” is the position on the sphere that the impulse will be applied 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.

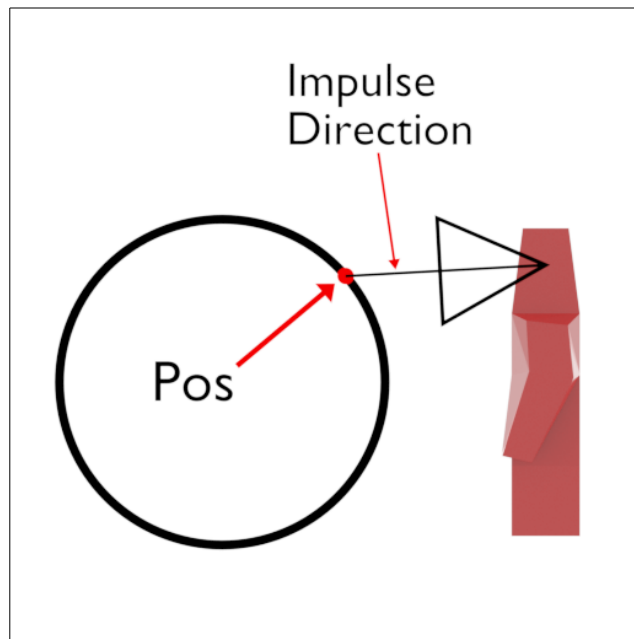
Here’s the final part where we set everything up to do our “**apply_impulse()**” on the sphere. We set the “**pos**” variable according to the player’s “**Use**” ray cast intersection position. Now here’s the confusing part: we do this in global space but relative to the sphere. So we take the ray cast 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 its 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 function “**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 = 0.2**
 - The multiplier that sets how much a slope affects the character's applied gravity when climbing up one.
 - Used in "**Slope_AffectSpeed()**" to say how much slopes affect walking speed when moving up one.
- **Slope_EffectMultiplier_ClimbingDown = 1.5**
 - The multiplier that sets how much a slope affects the character's applied gravity when climbing up one.
 - This is more than 1.0 because I want the player to "stick" to the slope when walking down it, so as to keep the character from raising off the platform and falling on it, over and over again.
 - It also gives a sense of gravity as a person tends to walk down slopes faster than up them.
 - Used in "**Slope_AffectSpeed()**" to say how much slopes affect walking speed when moving down one.

Use Action

- **Ray_UseDist = 2.0**

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

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 say where to interpolate to.

- **CamInterpo_StartingPos_Local_Y = 0.0**
 - The camera position, in local space, where it would be if it had not moved up with the player when encountering a step.
 - Used in:
 - “**Step_Player_Up()**” to set itself according to where it would be if it had not moved up with the player when encountering a step.
 - The “**_physics_process() > FINAL MOVEMENT APPLICATION > CAMERA INTERPOLATION**” as the first argument in the “**InterpolateCamera()**” function.
- **CamInterpo_CurrentTime_Secs = 0.0**
 - The current time of the camera interpolation that is in progress, in seconds.
 - Used in:
 - “**Step_Player_Up()**” to set the time of the interpolation to “**0**” after the player has gone up a step.
 - The “**_physics_process() > FINAL MOVEMENT APPLICATION > CAMERA INTERPOLATION**” section as the keeper of the return value of “**InterpolateCamera()**” and also as its second argument.

Ray Casting

Variables concerning the casting of rays.

- **Ray_SpaceState = null**
 - The state of the physical space in the game.
 - This holds all the information about collisions, collision boxes, where they are located, how big they are, their shape, and anything else pertaining to collision detection.
 - Used in several places.
- **Ray_From = Vector3(0.0, 0.0 ,0.0)**
 - The position that the ray cast will come from.
 - Used in several places.
- **Ray_To = Vector3(0.0, 0.0 ,0.0)**
 - The position that the ray cast will go to.
 - Used in several places.
- **Ray_Result = null**
 - The resulting dictionary of information received from “**Ray_SpaceState.intersect_ray()**”, which holds information like the collider, the ray cast collision position, or if the ray cast even hit anything in the first place.
 - Used in several places.

Interaction

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

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

Chapter 3: Functions Explained In-Depth

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

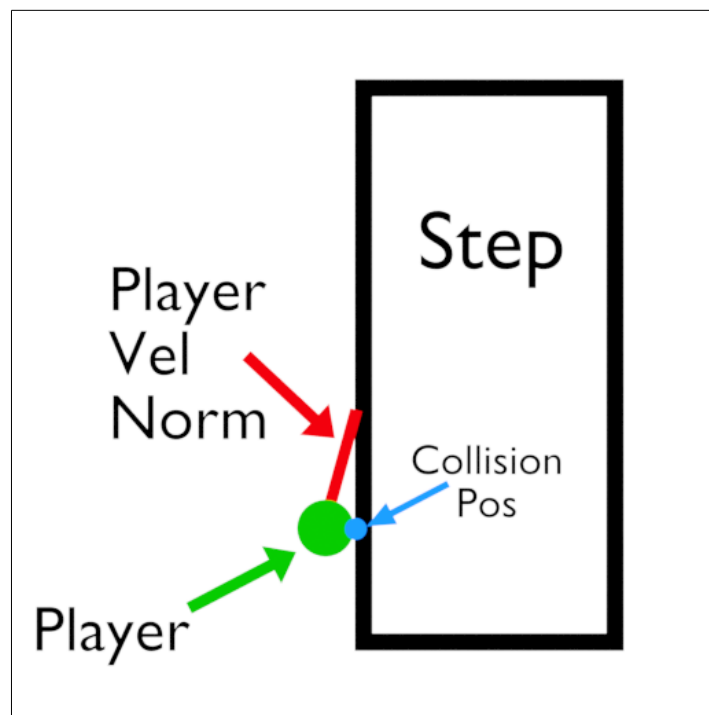
The Y axis, or the second element, of this vector will always equal “0”. This is because the second element of the first vector in this code is the player’s y position. That’s so we can just compare the horizontal axis positions without having to worry about the vertical position, which isn’t important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

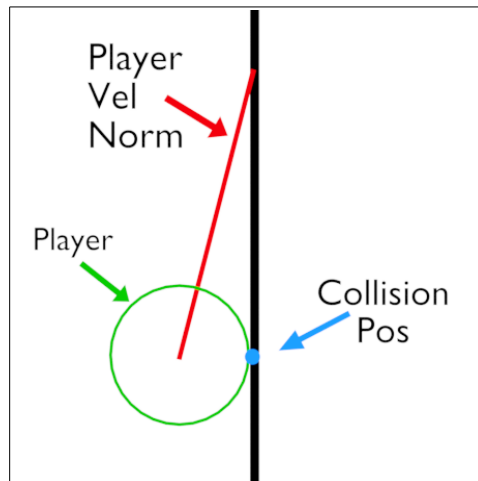
In this line of code, we have a lot going on.

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:

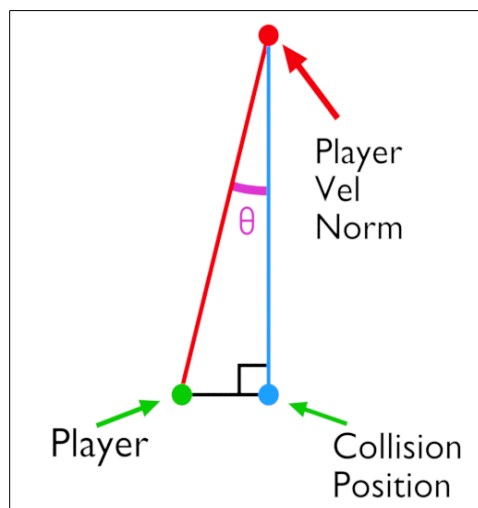
```
Step_CollPos_Global_RelToPlayer.dot(Step_PlayerVel_Global_Norm)
```

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.

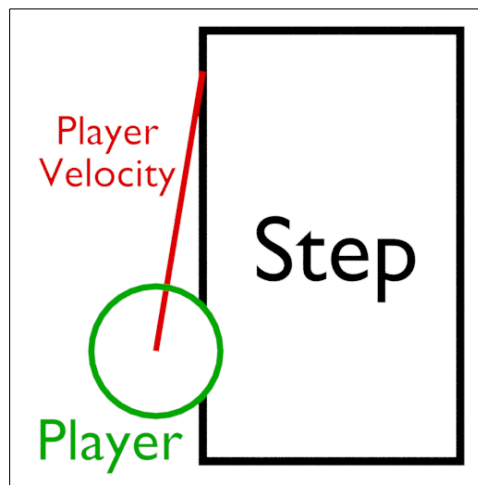
Okay, that all great, but what's the point? Let's look at the next line:

```
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.

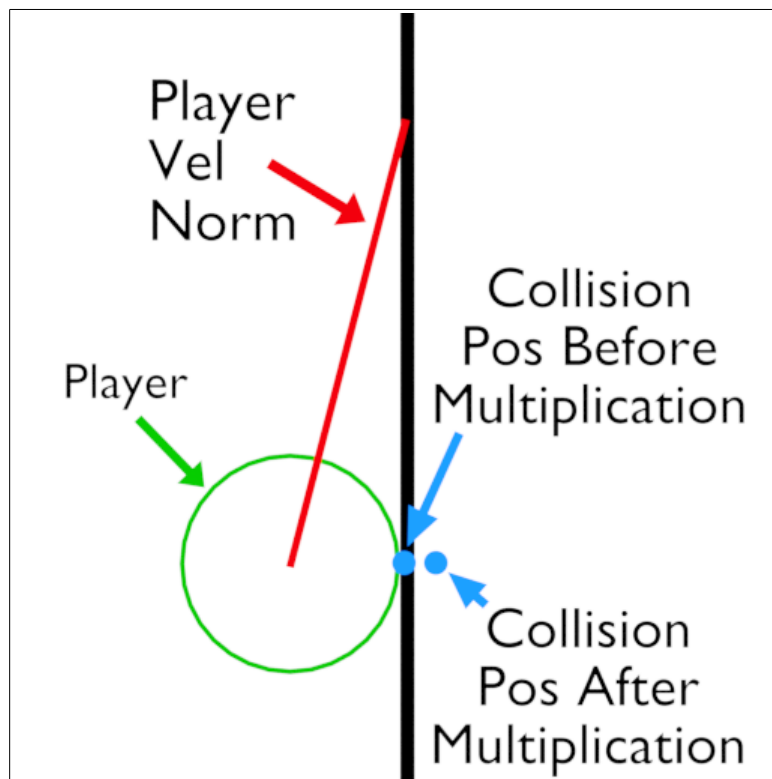
If the player is moving at *more* than a 11.5 degree angle, there is enough velocity of the player towards the step that he is able to move onto it.

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**”. This mea

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])
```

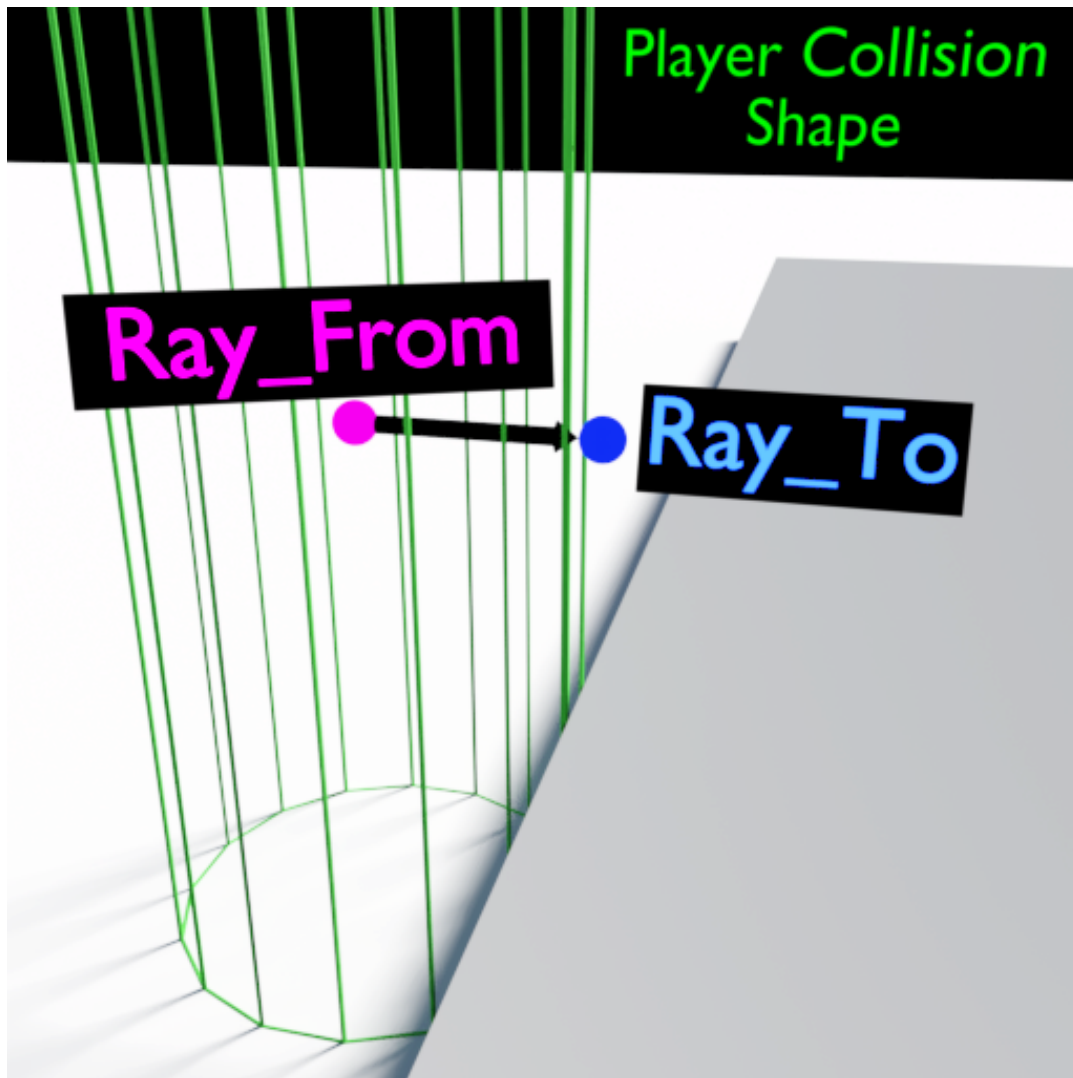
“**Ray_SpaceState**” get the state of the physics “world”, that is it get the state of everything in the scene that has anything to do with physics, like collision shapes, locations, etc. We call the ray tracing function from this space state.

“**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.

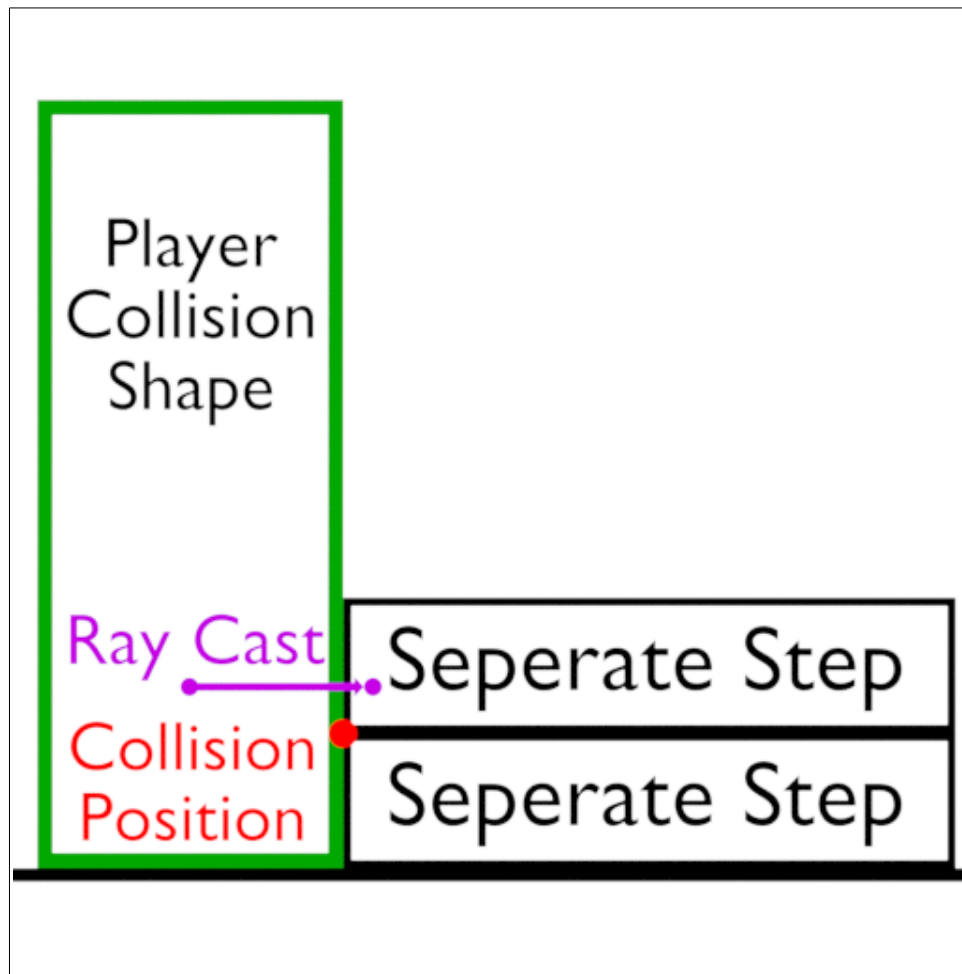
“**Ray_Result**” cast the ray by using “**Ray_SpaceState.intersect_ray()**” and keeps the result of that ray cast, which we can look at in the next line of code.

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.

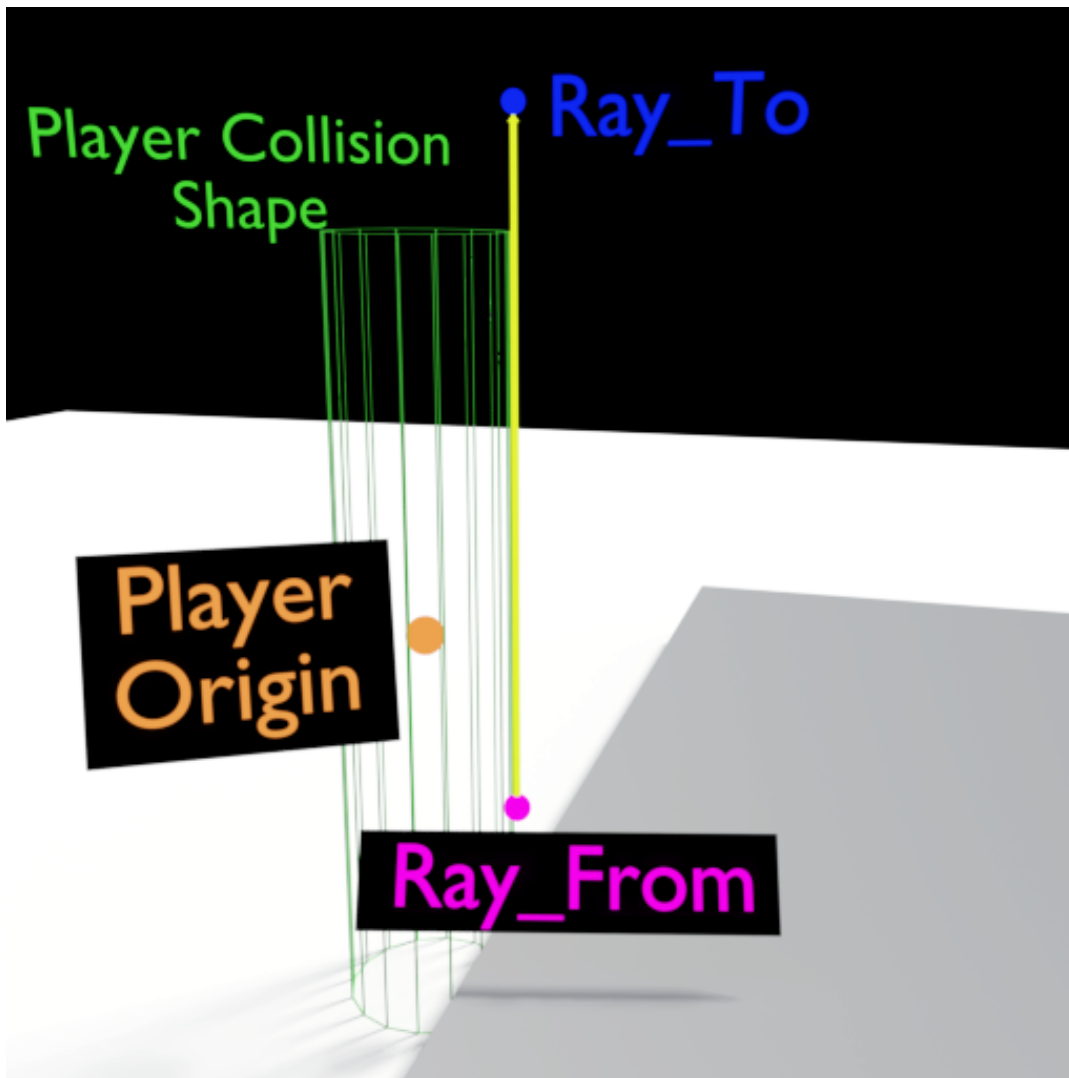
```
if(Ray_Result.empty()):
```

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.

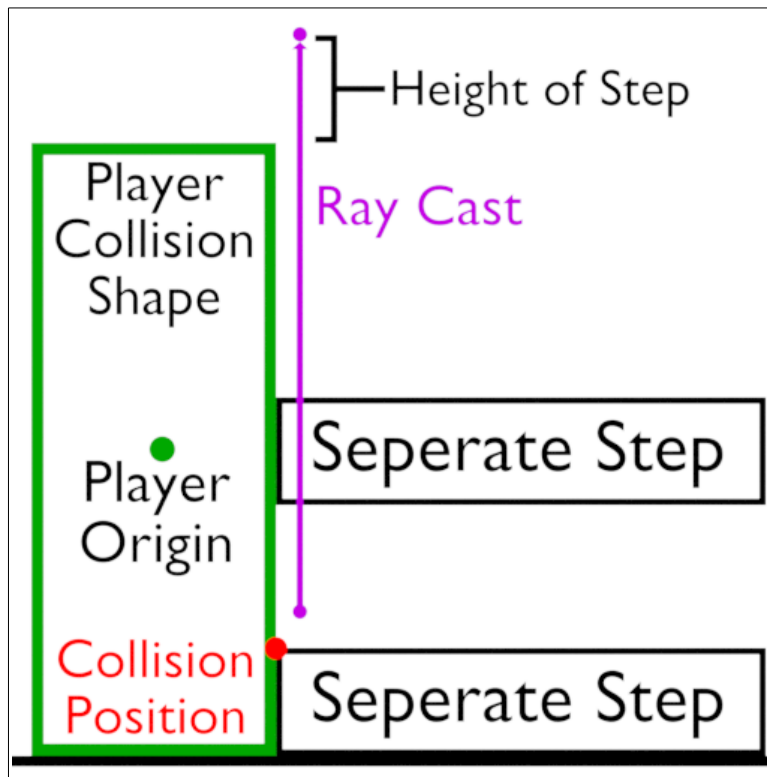
But, if it is empty, we move on with the code.

```
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)
```

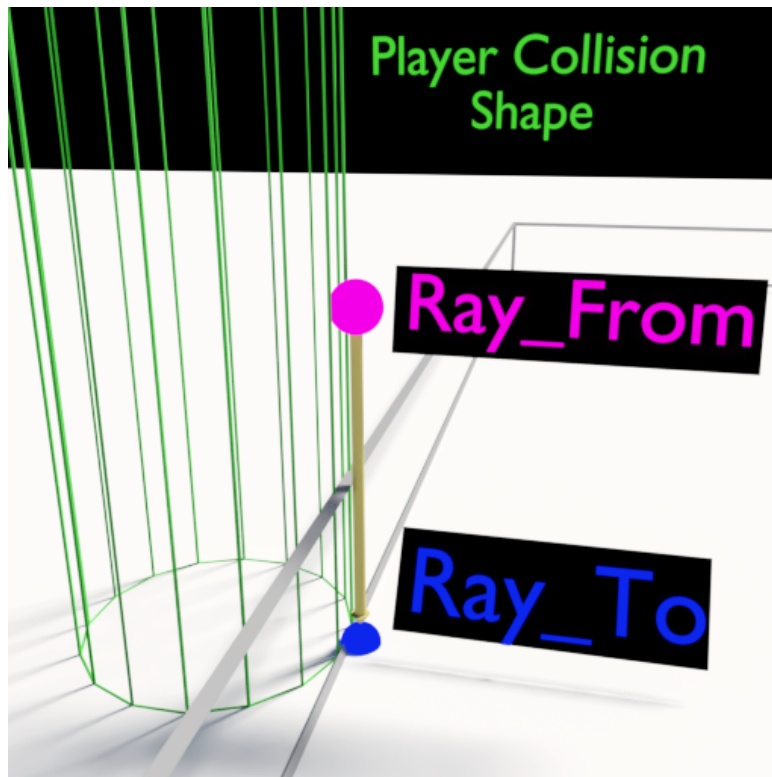
```
Ray_Result = Ray_SpaceState.intersect_ray(Ray_From,Ray_To,[self])
```

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

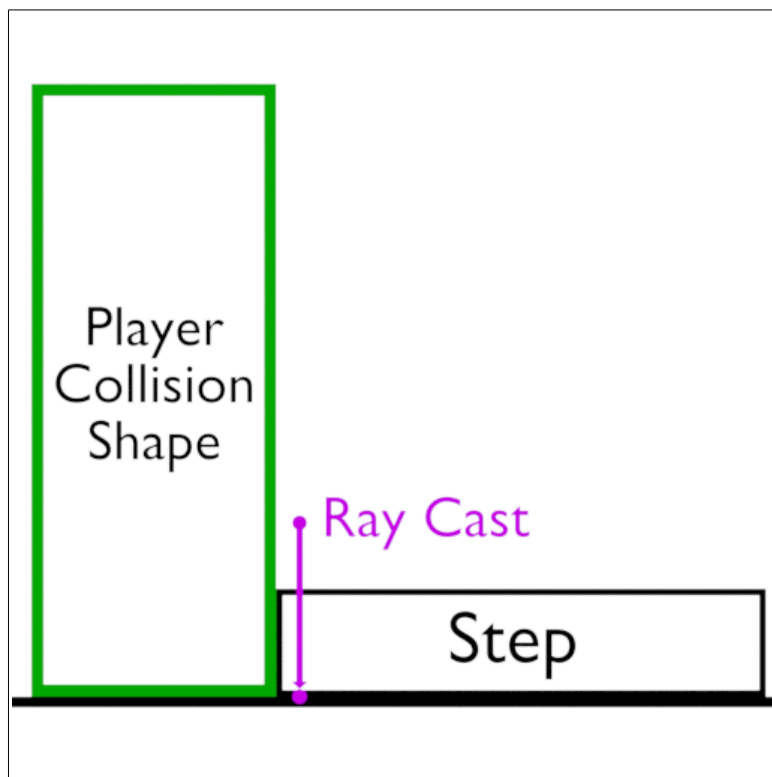
“**Ray_From**” is the same as it was last time, being the modified step collision position.

“**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 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.

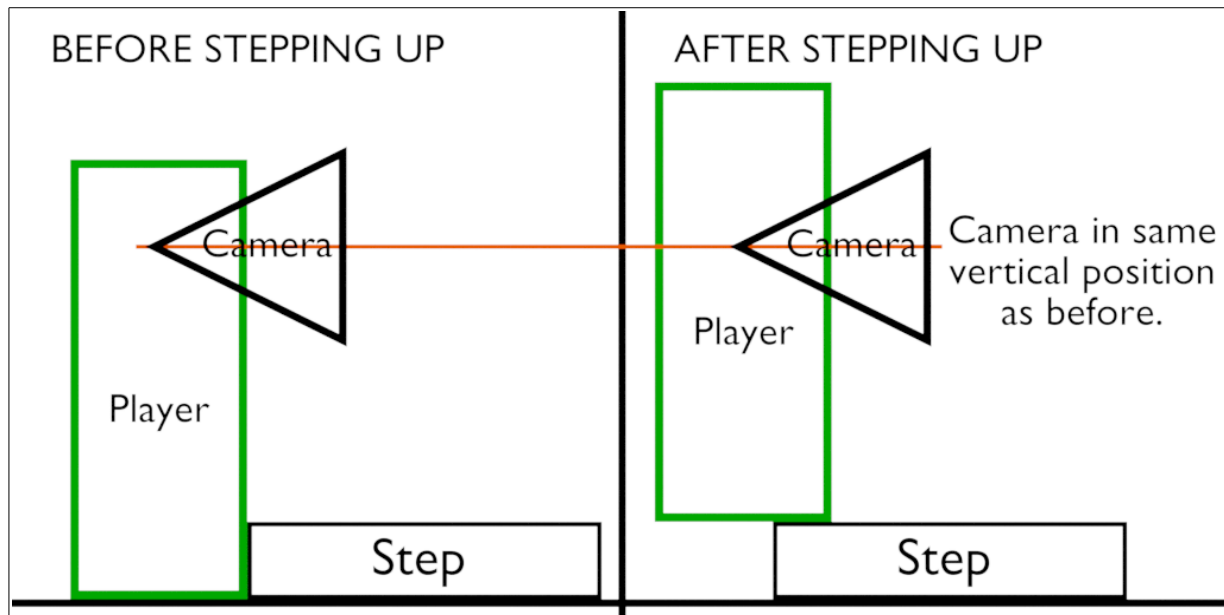
Then we get the Y axis position of the step. Let's say it's **"2.0"**. So let's take that and subtract **"1.6"** from it. We get **"0.4"**, which is the height of the step.

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()

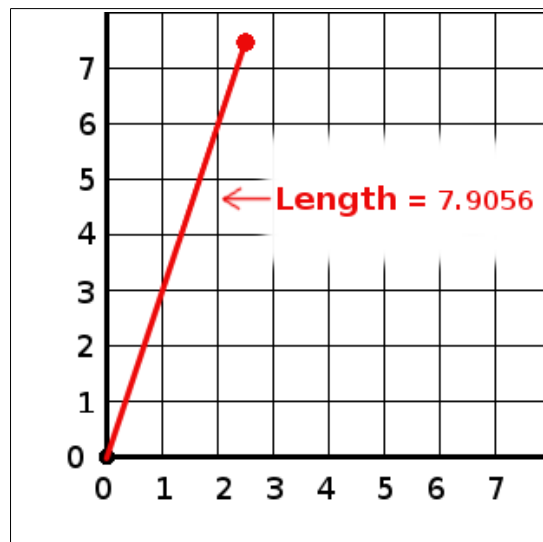
Section 3: Normals Explained

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.

Chapter 1: 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 scalars.

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:

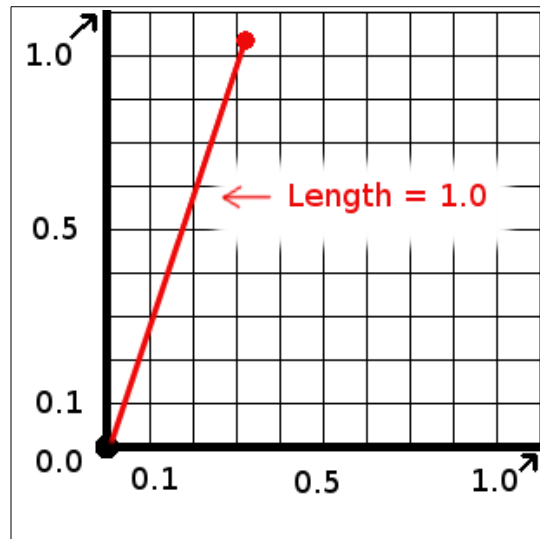
```
2D_Vector_01 = 2D_Vector_01.normalized()
```

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:

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

Which in this case would be:

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



We could also use “**Vector01.length()**” to get the magnitude of the vector, but we wouldn’t have 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 dividing them by the magnitude, like this:

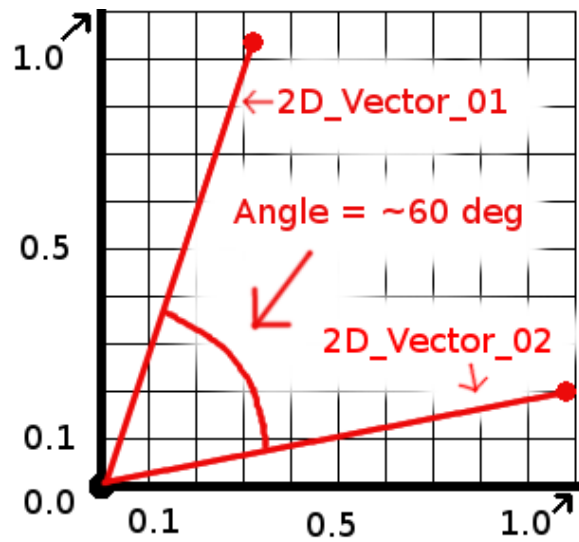
$$\text{normalized_vector} = (2.5 / 7.90569415, 7.5 / 7.90569415) = (0.316227766, 0.948683298)$$

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.

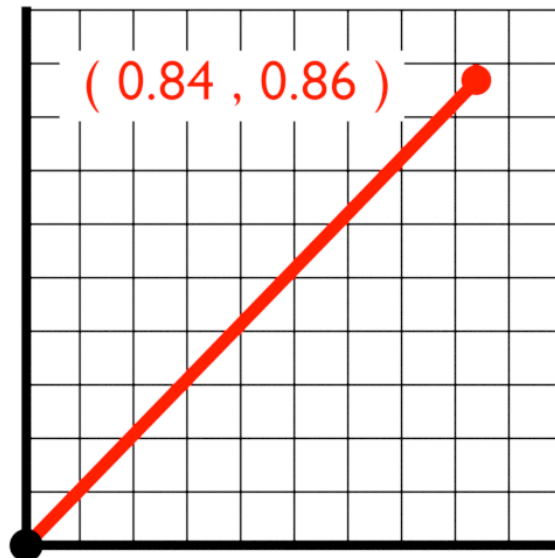
If we normalized “**2D_Vector_02**” (that is “**(0.5, 0.1)**”), it would be “**(0.980581, 0.196116)**”. Now we can dot product these two vectors, which would equal “**0.496139**”, which is what we want. If we wanted to we could get the angle between these two vectors by using the following code:

$$\text{angle} = \text{rad2deg}(\text{acos}(\text{2D_Vector_01.dot(2D_Vector_02)})) = \sim 60.255 \text{ degrees}$$

(Check next page for illustration.)



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:

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

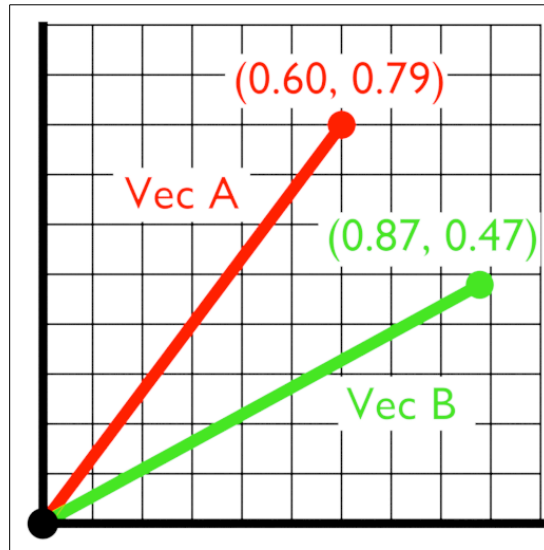
So the length of our vector is actually "1.2"... Well, then, let's make it "1.0".

$$\text{normalized_vector} = (0.84 / 1.202\sim, 0.86 / 1.202\sim) = (0.698739524, 0.715376179)$$

And, finally, there's our normalized vector, “(0.698739524, 0.715376179)”.

Chapter 2: 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()**”.

$$\text{Angle} = \text{acos}(0.8933) = 0.466161766$$

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...

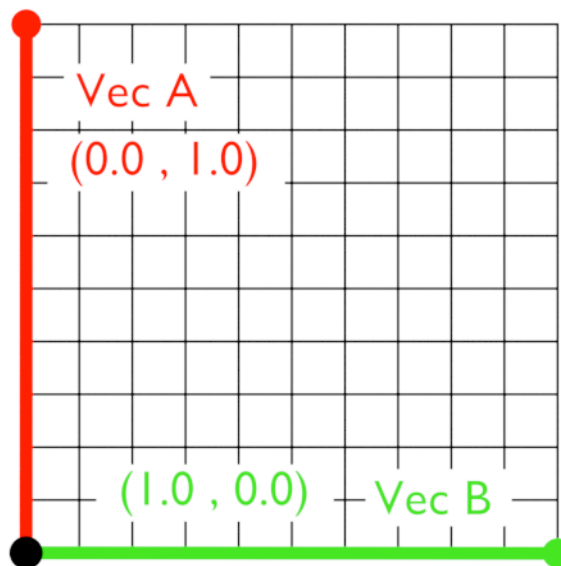
$$\text{Angle_Degrees} = \text{rad2deg}(0.466161766) = 26.709101783 \text{ degrees}$$

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

$$\text{Angle_Degrees} = 0.466161766 * (180/\text{PI}) = 26.709101783 \text{ degrees}$$

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:

$$\text{DotProduct} = (\text{Vector_A.x} * \text{Vector_B.x}) + (\text{Vector_A.y} * \text{Vector_B.y})$$

And the real numbers are:

$$\text{DotProduct} = (0.0 * 1.0) + (1.0 * 0.0) = 0.0 + 0.0 = 0.0$$

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?

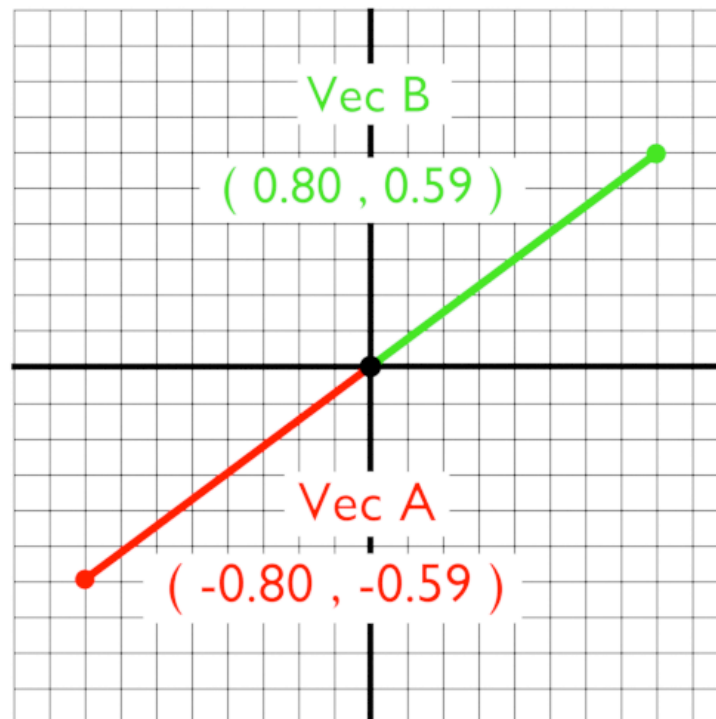
$$\text{Angle} = \text{acos}(0.0) = 1.570796327$$

And conversion to degrees...

$$\text{Angle_Degrees} = 1.570796 * (180/\text{PI}) = 90$$

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?

$$\text{DotProduct} = (-0.801838 * 0.801838) + (-0.597542 * 0.597542) = -0.642944178 + -0.357056442 = -1.0$$

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 like dogs.

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?

$$\text{Angle} = \text{acos}(-1.0) = 3.141592654$$

And converting it to degrees...

$$\text{Angle_Degrees} = 3.141592654 * (180/\text{PI}) = 180$$

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

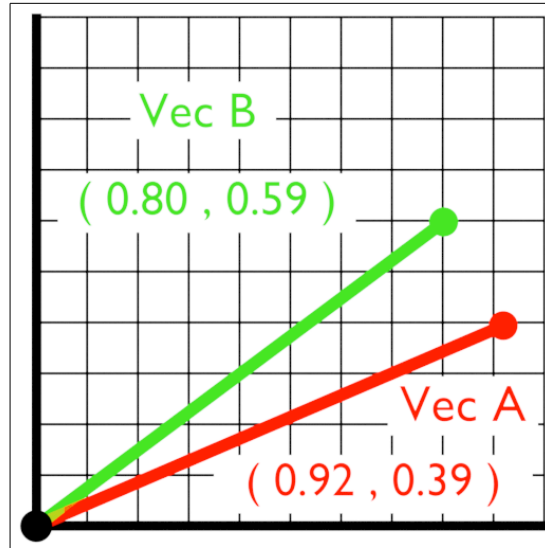
3D Dot Product Formula

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

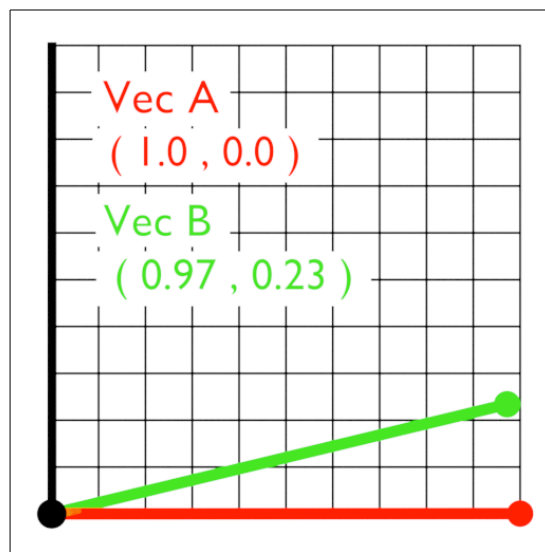
$$\text{DotProduct} = (\text{Vector_A.x} * \text{Vector_B.x}) + (\text{Vector_A.y} * \text{Vector_B.y}) + (\text{Vector_A.z} * \text{Vector_B.z})$$

Understanding the Dot Product

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



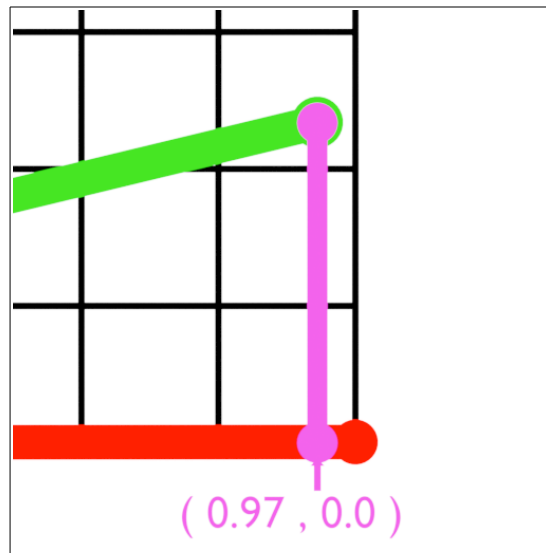
Now, in order to visualize what's going on, we will tilt both the vectors at the same time so as to lay Vector A down on the black line, so that it will be “(1.0 , 0.0)”:



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:

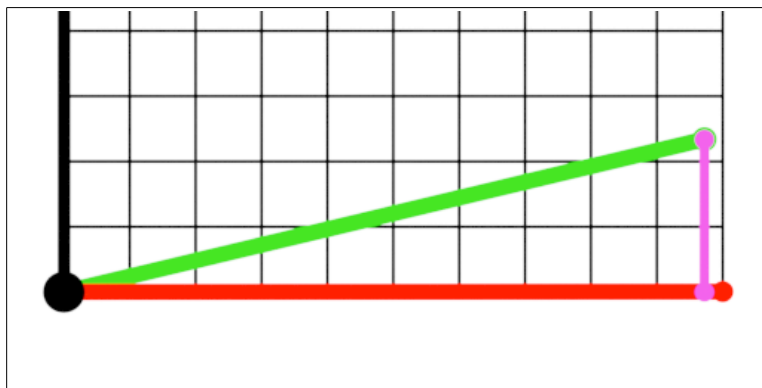
$$\text{DotProduct} = \text{VecA.dot}(\text{VecB})$$

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:

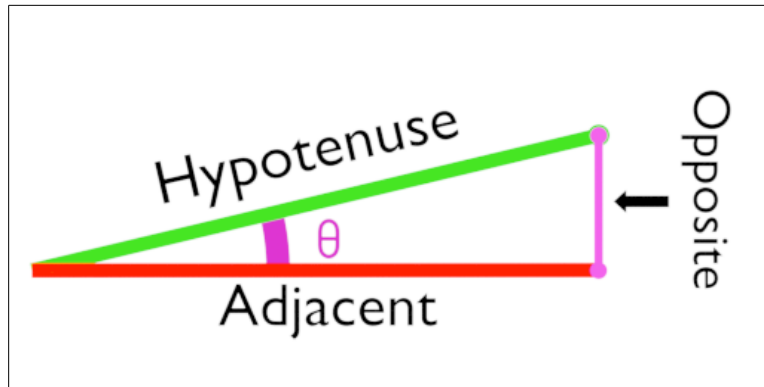


It's a right triangle! Now we can see what angle we've got by using arcsine, arc cosine, or arctangent!

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

$$\text{acos}(0.972219) = 0.236265072 \text{ radians} = 13.536991487 \text{ degrees}$$

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 “**arccosine**” 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 arccosine 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.