

Godot

- [Getting Started with Godot](#)
- [Simple XR Rig](#)
- [VR Controller Input in Godot](#)
- [Basic Grab](#)
- [Common 3D Features](#)

Getting Started with Godot

Before jumping into VR, you should get yourself acquainted with the engine and how things work. Godot utilizes a node-based system, similar to Unity except everything is its own node.

Godot also has its own scripting language, GDScript, which is very similar to python. Most of the tutorials, guides, and resources you find online will make use of GDScript.

There is also support for C# which I recommend for both speed and consistency with the similarities with Unity. The C# community within Godot is growing fast as well.

Documentation

<https://docs.godotengine.org/en/stable/index.html>

Overview of Godot's key concepts

https://docs.godotengine.org/en/stable/getting_started/introduction/key_concepts_overview.html#scenes

Your first 3D game

https://docs.godotengine.org/en/stable/getting_started/first_3d_game/index.html

Additional Resources

[How to make a Video Game - Brackeys](#)

Simple XR Rig

A lot has changed from the original version of this page. Below, I have linked the first article but you must follow all the chapters. These include:

- Introduction to the XR system in Godot
- Prerequisites for XR in Godot 4
- OpenXR
- Setting up the XR scene

Setting up OpenXR

https://docs.godotengine.org/en/stable/tutorials/xr/setting_up_xr.html

You may find the following two articles: **Introduction to XR Tools** and **Basic XR Locomotion**. These articles are valid but I would caution against using them and consider designing your own implementation and use these articles for reference only. If you have completed work in Unreal for things like teleportation, you will find there is a cross-over on making this work in Godot/Unity/etc. by translating the nodes into the APIs in the newer engine. I (Wes) am happy to assist.

VR Controller Input in Godot

Preface:

I use C# for coding in Godot due to the speed and resources (we need all the resources we can get when building with VR). Below is an example using C# but can be converted to GDScript as well.

For more information about C# in Godot:

https://docs.godotengine.org/en/stable/tutorials/scripting/c_sharp/c_sharp_basics.html

Getting Started

Assuming you have followed the previous tutorial and confirmed VR is working with controller visuals (even cubes) and OpenXR, you can get started.

By default, after installing OpenXR within Godot enables a default **OpenXR Action Map**. Typically, I leave these defaults alone. There is a good article in the documentation that reviews how to create your own custom inputs, but these work great.

The event structure on how you process this input is up to you! Below is an EXAMPLE but should be structured!

1. Create subscription events for ButtonPressed and ButtonReleased.

Create a script on your XRController3D node (the VR controller) and the script should inherit from XRController3D.

Inside the script, create two override functions: `_EnterTree()` and `_ExitTree()`. Add `ButtonPressed` and `ButtonReleased` and subscribe to a method. Here is the layout:

```
public override void _EnterTree()
{
    ButtonPressed += OnButtonPressed;
    ButtonReleased += OnButtonReleased;
}

public override void _ExitTree()
{
    ButtonPressed -= OnButtonPressed;
    ButtonReleased -= OnButtonReleased;
}
```

Now, create two functions, `OnButtonPressed` and `OnButtonReleased`:

```
void OnButtonPressed(string name)
{

}

void OnButtonReleased(string name)
{

}
```

2. Process events

You will notice on the methods created above that there is a parameter called `string name`. This string is what is received from your **OpenXR Action Map** (or anything else with a button). For me, I have added a switch statement to process these inputs and invoke methods:

```
void OnButtonPressed(string name)
{
    switch (name)
    {
        case "grip_click":
            OnGripPressed();
            break;
        case "trigger_click":
            OnTriggerPressed();
            break;
        case "ax_button":
            OnAXPressed();
            break;
        case "by_button":
            OnBYPressed();
            break;
    }
}

void OnButtonReleased(string name)
{
    switch (name)
    {
```

```

        case "grip_click":
            OnGripReleased();
            break;
        case "trigger_click":
            OnTriggerReleased();
            break;
        case "ax_button":
            OnAXReleased();
            break;
        case "by_button":
            OnBYReleased();
            break;
    }
}

```

I'm not including every single method, but you can see that I now have a system to process input!

3. Expand (with some Object Oriented Programming)!!

Obviously, this script would be a pain to copy for both hands and alter the names of the methods slightly. You can do this if you wish to keep it simple.

Object Oriented Programming (OOP):

For me, I created a parent script called something like XRHand.cs. Then, I created two scripts called LeftHand.cs and RightHand.cs that are children of XRHand. Using the methods above, I made them virtual so in my LeftHand.cs script for example, I can configure special implementation.

Here is the final script for XRHand.cs

```

using Godot;
using System;

public partial class XRHand : XRController3D
{
    public override void _EnterTree()
    {
        ButtonPressed += OnButtonPressed;
        ButtonReleased += OnButtonReleased;
    }
}

```

```

        InputFloatChanged += OnInputFloatChanged;

        InputVector2Changed += OnInputVector2Changed;
    }

public override void _ExitTree()
{
    ButtonPressed -= OnButtonPressed;
    ButtonReleased -= OnButtonReleased;

    InputFloatChanged -= OnInputFloatChanged;

    InputVector2Changed -= OnInputVector2Changed;
}

void OnButtonPressed(string name)
{
    switch (name)
    {
        case "grip_click":
            OnGripPressed();
            break;
        case "trigger_click":
            OnTriggerPressed();
            break;
        case "ax_button":
            OnAXPressed();
            break;
        case "by_button":
            OnBYPressed();
            break;
    }
}

void OnButtonReleased(string name)
{
    switch (name)
    {
        case "grip_click":
            OnGripReleased();

```

```

        break;
    case "trigger_click":
        OnTriggerReleased();
        break;
    case "ax_button":
        OnAXReleased();
        break;
    case "by_button":
        OnBYReleased();
        break;
    }
}

void OnInputFloatChanged(string name, double value)
{

}

void OnInputVector2Changed(string name, Vector2 value)
{
    switch (name)
    {
        case "primary":
            OnThumbstickMoved(value);
            break;
    }
}

// Digital Input
public virtual void OnGripPressed() { }
public virtual void OnGripReleased() { }
public virtual void OnTriggerPressed() { }
public virtual void OnTriggerReleased() { }
public virtual void OnAXPressed() { }
public virtual void OnAXReleased() { }
public virtual void OnBYPressed() { }
public virtual void OnBYReleased() { }

// Axial-1D Input

```



```
// Axial-2D Input
public virtual void OnThumbstickMoved(Vector2 value) { }

// Signals

}
```

And as for LeftHand.cs or RightHand.cs, I would just override a button that I would need:

```
using Godot;
using System;

public partial class LeftHand : XRHand
{
    public override void OnGripPressed()
    {
        GD.Print("Right grip released!");
    }

    public override void OnGripReleased()
    {
        GD.Print("Left grip released!");
    }
}
```

The LeftHand.cs and RightHand.cs would now go on their respective XRController3D nodes, in the scene.

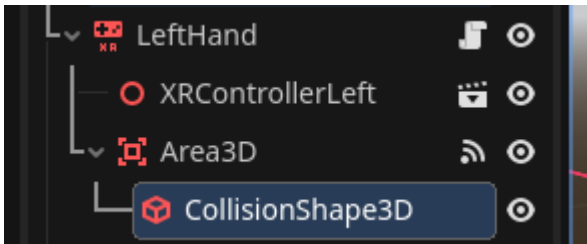
Basic Grab

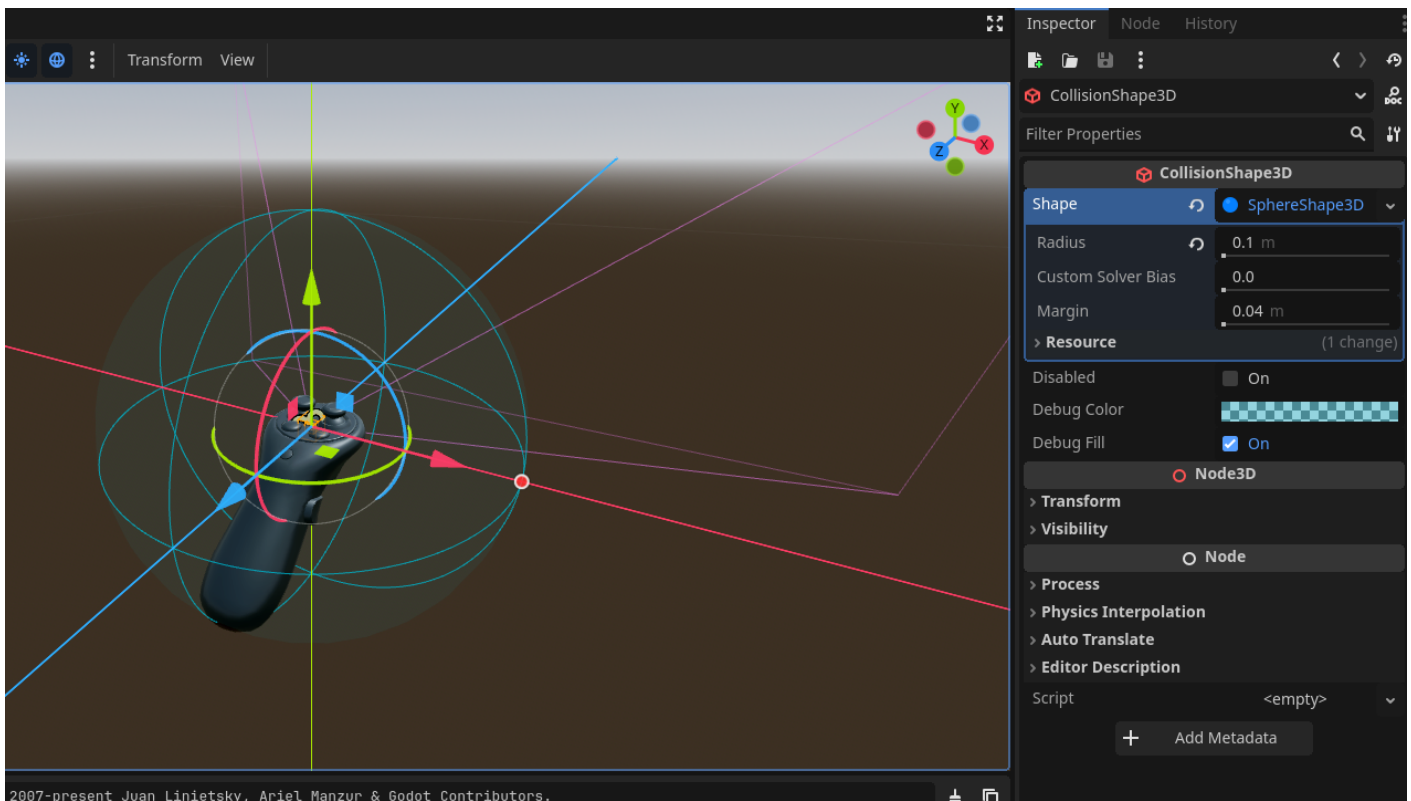
Prerequisites:

- Basic [familiarity](#) with nodes in Godot
- [XRRig](#)
- [Controller Input](#)
- Ensure you have a ready LeftHand and RightHand script for input.

Setup for XR Rig

For each *XRController3D* node (VR hands), add an **Area3D** node as a child. You should get a warning about the Area node needing a collision shape, so add a new **CollisionShape3D** node as a child of the **Area3D** node. For my **Shape** type, I chose a new **SphereShape3D** and gave it a radius of **0.1m**. Do this for all XRController3D nodes if you are using pickup interactions.





Setup basic grabbing

In your "world" scene, add a **RigidBody3D** node to your scene and add a **CollisionShape3D** and **MeshInstance3D** node as children, as well as any setup needed for those nodes. In the **RigidBody3D** node, create a new script called **Grippable.cs** and ensure the script is attached. Now, let's add some code:

```
using Godot;
using System;

public partial class SimpleGrab : RigidBody3D, IGrabbable
{
    [Node3D] parentNode;

    public override void _Ready()
    {
        parentNode = (Node3D)this.GetParent();
    }

    public void Pickup(Node3D receivedController)
    {
    }
```

```

    _Freeze = true;

    Reparent(receivedController, true);
}

public void Drop(Vector3 receivedVelocity)
{
    _Freeze = false;

    Reparent(parentNode);

    //CallDeferred("set_axis_velocity", receivedVelocity);
    SetAxisVelocity(receivedVelocity);
}
}

```

Let's review what's happening here:

- The **parentNode** is a variable used to cache where we need to re-parent this object when it is dropped, which is usually back to the world. In our **_Ready** method, we just get the current parent.
- The **Pickup** method and its reference to the controller doing the interaction is done here. We **freeze** physics while we are holding this object and parent it to our controller. The **true** parameter in **Reparent** is passed to keep the global transform before parenting so the pickup looks like we are grabbing an intended point.
- The **Drop** method does the reverse of **Pickup**, except an added line that gives the object velocity from the controller when the object is let go. *There is an optional line if you wish to use CallDeferred instead of SetAxisVelocity (for advanced users).*

Add functionality to our hand (each)

Assuming you have completed all necessary prerequisites, you should have a **LeftHand.cs** and a **RightHand.cs** with some possible functionality. Here is additional code to add to each hand:

```

Vector3 velocity;
Vector3 previousPosition;

Area3D area;
Grippable grippable;

```

```

public override void _Ready()
{
    area = GetNode<Area3D>("Area3D");
}

public override void _Process(double delta)
{
    velocity = (Position - previousPosition) / (float)delta;
    previousPosition = Position;
}

public override void OnGripPressed()
{
    var bodies = area.GetOverlappingBodies();
    foreach (var body in bodies)
    {
        if (body is Grippable _)
        {
            grippable = body as Grippable;

            grippable.PickUp(this);

            return;
        }
    }
}

public override void OnGripReleased()
{
    if (grippable != null)
    {
        grippable.Drop(velocity);
    }
    grippable = null;
}

```

Let's review what this code achieves:

- A variable, **velocity**, is added for when we need to let to drop an object and give it velocity. *The benefit is this calculation does not involve a physics node or other physics*

calculations that are not needed. You may optionally add a boolean flag in the `_Process` method if you do not wish to calculate velocity every frame except for when you are actually holding something.

- In **OnGrippedPressed**, we use our **Area3D** node to check all overlapping bodies. If one of these bodies is a Grippable type, then we attempt to pick it up and return out of the method.
- For **OnGrippedReleased**, we do a check to see if we are actually holding a grippable object and, if we are, we drop the object and pass along our hand's velocity. Finally, we indicate we are no longer holding anything in this hand.

Testing

If you have reached this stage, go ahead and test functionality with both hands.

Where to go from here?

If you are looking to add advanced interactions like sliders and dials, I may recommend keeping these methods for grabbing types, but add new classes that track the controller's movement instead of parenting when picked up or dropped.

Common 3D Features

This page consists of the culmination of the Brackeys video: How to make 3D Games in Godot. The video is not a tutorial, but a guide to common engine features like lighting, physics, materials, animations, and more. For convenience, video link and its times stamps are below:

[How to make 3D Games in Godot \(video\)](#)

[3:35](#) 3D Space

[6:13](#) Greyboxing

[9:31](#) Terrain

[10:01](#) Playing the Game

[11:11](#) Character Controller

[14:15](#) 3D Physics

[16:10](#) Graphics

[17:44](#) 3D Assets in 1 min

[18:45](#) Assets in Godot

[22:38](#) StandardMaterial3D

[26:53](#) Scene Workflows

[30:53](#) Collision

[32:57](#) Replace Greybox

[35:43](#) Animated Characters

[38:33](#) Speed up Workflow

[39:24](#) Environment

[42:00](#) Lighting

[45:38](#) Tonemap

[47:18](#) Camera

[48:45](#) Render Quality