Ateneo Virtual Reality Escape (AVRE) Framework Documentation

This document contains the full documentation of the Ateneo Virtual Reality Escape (AVRE) Framework. Previous versions of some elements of the AVRE framework can be found here.

Object Interaction System (OIS)

This subsystem was created to support the development of controller-based interactions with different virtual objects in a VR environment. These interactions can range from simple ones consisting of aiming and/or selecting one of the controller buttons, to more complex ones which may require the user to orient their controller in certain configurations and perform different motions.

These interactions are facilitated by two different kinds of objects

- ➔ Actor Objects - Objects that the user can control and interact with other objects in the virtual world.
    - ◆ Can be the user's virtual hand (controller) or objects the player can hold on to and use.
    - ◆ One-handed and Two-handed objects
- ➔ Receiver Objects - Objects where the action is executed (or receives the action)
    - ◆ Can be a specific object (i.e. a specific lock that gets a specific key actor), a class of objects (i.e. smashing breakable objects using an actor that can hit those objects), or all objects.

Objects can be both a receiver and an actor.

OIS Base Nodes

The following are the base nodes for this version of the Object Interaction System. In most cases, except for the OISActorComponent, these nodes are not to be used, unless you intend to create your own extensions to the Object Interaction System.

OISActorComponent

The OISActorComponent is placed as a child node of whatever Actor object you want to make. This can be the controller node itself, or any other object. The most common use would be as a child node to an XRToolsPickable scene or the pickable.tscn scene that comes with Godot XRTools.

Properties

Export variables:

TYPE	NAME	DESCRIPTION
String	receiver_group	A string that represents the group the particular actor can interact with
float	actor_rate	A float that represents the rate in which the actor affects the receiver. Set to 1.0 by default.
bool	trigger_action	A boolean for whether or not pressing the trigger is required for doing the action

Other accessible variables:

TYPE	NAME	DESCRIPTION
OISRActorStateMachine	actor_state_machine	The actor component's state machine
OISCollider	actor_collider	The actor component's collider
OISReceiverComponent	ois_receiver	The receiver the actor is currently colliding with. Usually null when the actor is not in use

Methods and Functions

RETURN TYPE	NAME	DESCRIPTION
Variant	get_actor()	Returns the actor that the Actor Component is attached to
void	set_receiver(receiver: OISReceiverComponent)	Sets the receiver of the actor (usually when the actor collides with a receiver)
OISReceiverComponent	get_receiver()	Returns the current receiver of the Actor
float	get_actor_rate()	Returns the rate of the actor
void	actor_component_enabled (b: bool)	Enables or disables the actor given the boolean

OISActorStateMachine

The OISActorStateMachine is placed as a child of the OISActorComponent. This node handles the initializing, processing, and transitioning between states. This node on its own should not be used, unless you are defining your own state machine. Instead, use the premade State Machine nodes that extend this.

Properties

Export variables:

TYPE	NAME	DESCRIPTION
NodePath	initial_state	The starting state of the State Machine

Other accessible variables:

TYPE	NAME	DESCRIPTION
OISRActorComponent	actor	The actor component the State Machine is attached to
OISActorState	state	The current state of the State Machine
OISReceiver	receiver	The receiver of the action. Will most likely be null until collision with a receiver.T
XRController3D	controller	The controller that is doing the action or the controller holding the object doing the action.

Methods and Functions

RETURN TYPE	NAME	DESCRIPTION
void	transition_to(target_state: String, msg: Dictionary = {})	Transitions to the OISState node given its name. The msg dictionary allows for additional parameters to be sent to the next state

void	initialize()	Initializes the State Machine. Is called during the _ready() function.
OISActorComponent	get_actor_component()	Returns the actor the state machine is attached to

OISActorState

The states that are managed by the OISActorStateMachine. These state nodes manage specific behavior depending on the state the actor is in. Similar to the OISActorStateMachine, this node should not be used on its own unless you are defining your own State. Instead premade state nodes should be used, and these nodes are automatically made by the premade OISActorStateMachine nodes.

Properties:

Accessible variables

TYPE	NAME	DESCRIPTION
OISRActorStateMachine	_ois_actor_state_machine	The State Machine managing the State
OISReceiverComponent	ois_receiver	The receiver of the action of the Actor Component
bool	trigger_on	A boolean for whether or not the trigger of the controller is being held. (Note that currently, it does not automatically know whether or not the trigger is being held. To use, you have to define a function that changes trigger_on.)

Methods and Functions

Note that all methods and functions of the OIS State are essentially virtual in that they have to be overridden and defined by any state class that extends OISState

RETURN TYPE	NAME	DESCRIPTION
void	update(_delta : float)	This serves as the OISState's process function. Instead of having a process function called every frame, instead, the update function is called by the State's State Machine every frame. This ensures that only the current state is running every frame.
void	physics_update(_delta : float)	This serves as the OISState's physics_process function. Instead of having a physics_process function called every frame, instead, the physics_update function is called by the State's State Machine every physics frame. This ensures that only the current state is running every frame.
void	_on_enter_collision(receiver : Variant)	Receives a signal from the actor's collider when the actor enters a collision
void	_on_exit_collision(receiver : Variant)	Receives a signal from the actor's collider when the actor exits a collision
void	_on_trigger_press()	Called when the trigger button of the controller is pressed
void	_on_trigger_release()	Called when the trigger button of the controller is released

void	enter_state(_msg: Dictionary = {})	Is automatically called by the State's State Machine upon entering the state. The _msg: Dictionary is used to pass additional parameters to the State when necessary.
void	exit_state()	Is automatically called by the State's State Machine upon exiting the state.

OISCollider

Used as the collider for OISActors. Doesn't do anything on its own except connecting its own body_entered and body_exited signals to the OISActorComponent's _on_ois_receiver_collision_entered function and the _on_ois_receiver_collision_exited function respectively. Two different types of OISColliders have been implemented so far, and they will be described later.

OISReceiverComponent

The OISReceiverComponent is placed as a child of whatever receiver the developer wants to make. The OISReceiverComponent should not be used on its own, unless you are defining your own receiver components. Instead, premade receiver components should be used.

Signals

SIGNAL	DESCRIPTION
action_started(requirement,	Signal emitted the moment an OIS action is

total_progress)	performed on a Receiver
action_in_progress(requirement, total_progress)	Signal emitted every frame during an OIS action
action_ended(requirement, total_progress)	Signal emitted the moment an OIS action ends. Doesn't necessarily mean when an OIS action is completed
action_completed(requirement, total_progress)	Signal emitted the moment the receiver's action requirement is met.

Properties

Export variables:

TYPE	NAME	DESCRIPTION
String	group	A string representing the group that the receiver will accept action from
float	requirement	The requirement for completing an action. What this parameter means changes depending on the action
bool	snap_actor	A boolean for whether or not the actor will snap to the receiver

Other accessible variables:

TYPE	NAME	DESCRIPTION
bool	completed	Is a boolean for whether or not the action has been completed on the receiver
Variant	interacting_object	The actor interacting with the receiver
float	total_progress	The total progress of the action

float	rate	The rate in which the action is being done to the receiver
Area3D	area_3d	The area of the receiver
CollisionShape3D	collision_shape_3d	The collision shape of the area of the receiver
Marker3D	marker_3d	A marker used for snapping an actor to the receiver

Methods and Functions

RETURN TYPE	NAME	DESCRIPTION
void	initialize_action_vars()	Initializes the initial variables for when the action starts
void	start_action_check(actor : OISActorComponent)	Called when the action starts on the receiver and assigns the actor to the interacting_object variable
void	end_action()	Called when the action ends
void	action_ongoing()	Called every frame the action is occurring
void	check_if_completed()	Checks if the action is completed.

Implemented OISActorStateMachines

OISSingleControllerASM

The OISSingleControllerASM extends OISActorStateMachine and is used as the state machine for an ActorComponent that is attached to the controller. This essentially functions as the State Machine for actions being performed by the player's "hand".

This State Machine by default has the premade OISActorStates: ControllerIdleState, ControllerActiveState, and the ActiveCollidingState. The default state of this state machine is Active as the player will not be holding any item by default.

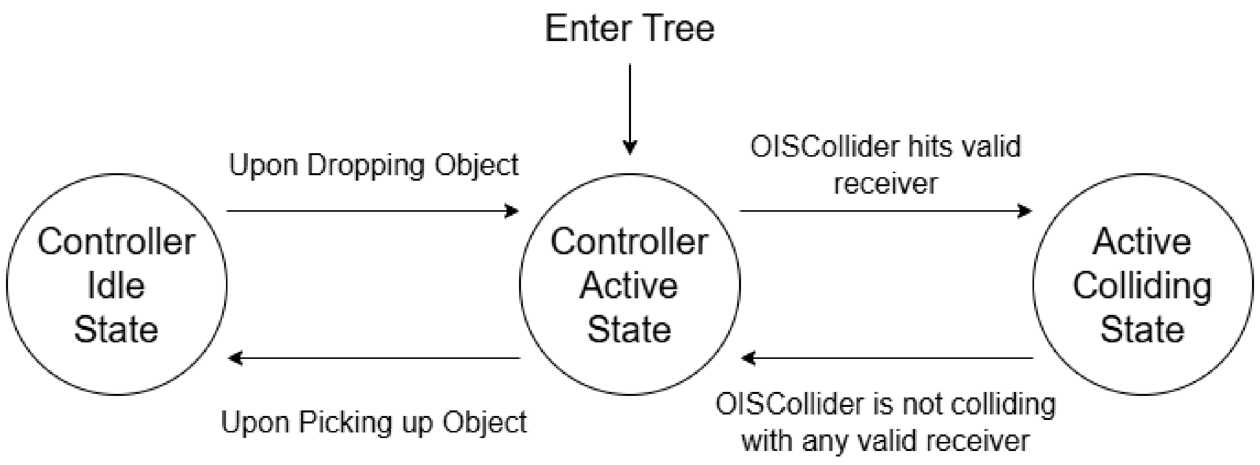


Fig. OISSingleControllerASM State Transition Diagram

The key difference between this new implementation of the OISSingleControllerASM and the previous version's single controller state machine is the removal of the Trigger State entirely. Instead, whether or not a receiver requires a trigger input is handled by an export boolean of the OISActorComponent. The reasoning for removing the Trigger State is that it essentially does the exact same thing as ActiveCollidingState most of the time, so consolidating the two states makes it simpler.

OISOneHandToolASM

The OISOneHandToolASM extends OISActorStateMachine and is used as the state machine for an ActorComponent that is attached to a pickable object. This functions as the State Machine for actions being performed by any one-handed tool.

This State Machine by default has the premade OISActorStates: ToolIdleState, ToolActiveState, and the ActiveCollidingState. The default state of this state machine is Idle since it will only be activated when the player picks up the actor.

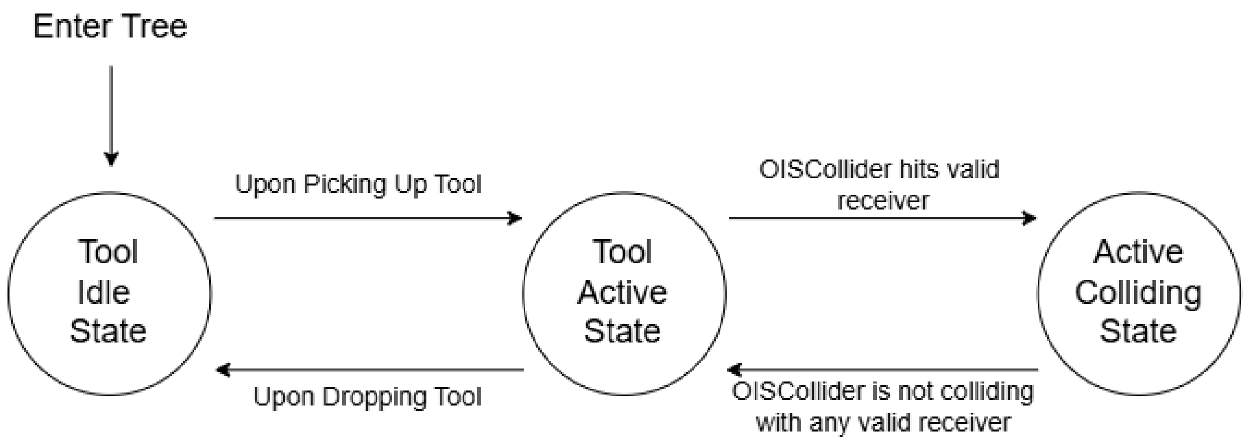


Fig. OISOneHandToolASM State Transition Diagram

Similar to the OISSingleControllerASM, this omits the Trigger State from the previous version for the same reason.

OISTwoHandToolASM

The OISTwoHandToolASM extends OISActorStateMachine and is used as a state machine for a pickable Actor that uses both hands. In addition to the default export variables of the OISActorStateMachine, it has an additional export variable.

Export variables:

TYPE	NAME	DESCRIPTION
bool	require_two_handed	A boolean that determines whether or not the actor will work with one hand. If false, it will work with one hand with half the output rate. If true, it will only work if being held by both hands.

This State Machine by default has the premade OISActorStates: ToolIdleState, ToolOneHandActiveState, ToolTwoHandActiveState, OneHandActiveCollidingState, TwoHandActiveCollidingState. The default state of this state machine is Idle since it will only be activated when the player picks up the actor.

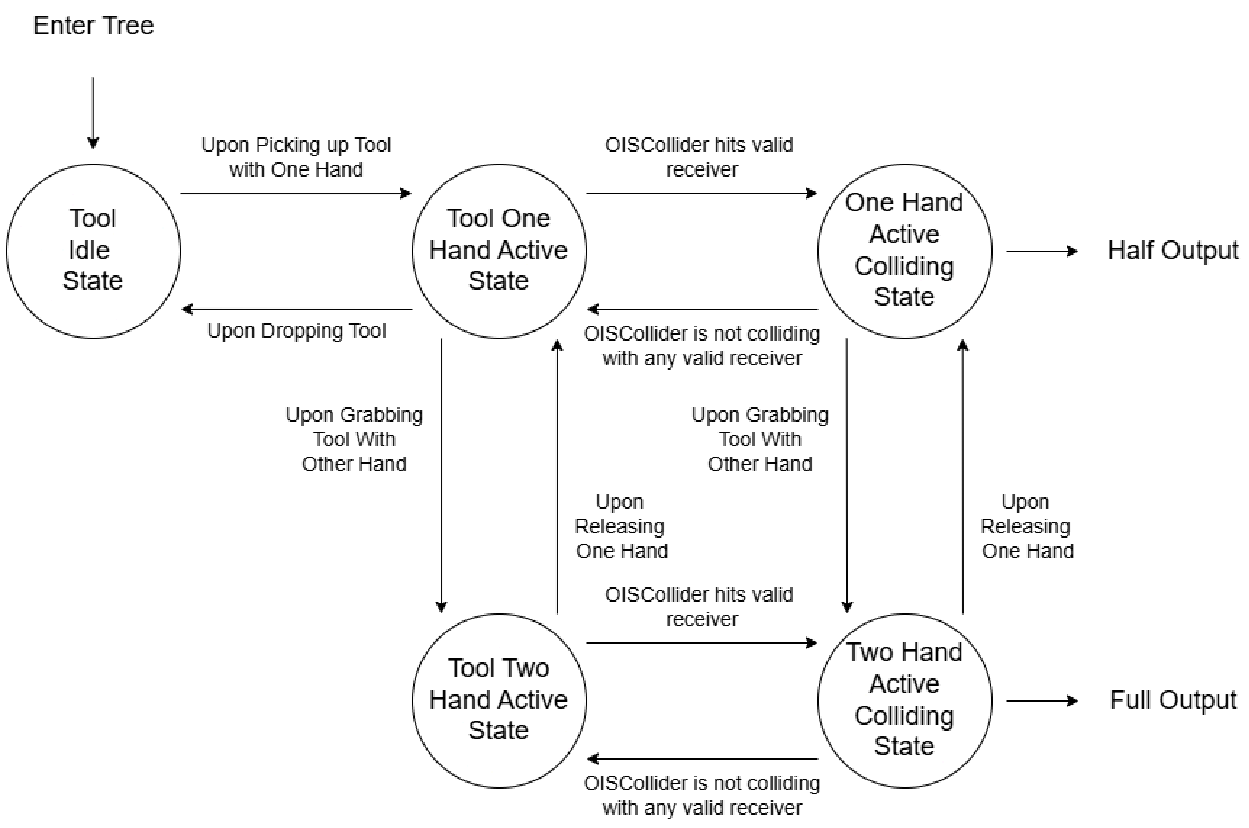

Enter Tree
Tool Idle State
Upon Picking up Tool with One Hand
Upon Dropping Tool
Tool One Hand Active State
OISCollider hits valid receiver
OISCollider is not colliding with any valid receiver
One Hand Active Colliding State
Half Output
Upon Grabbing Tool With Other Hand
Upon Releasing One Hand
Upon Grabbing Tool With Other Hand
Upon Releasing One Hand
Tool Two Hand Active State
OISCollider hits valid receiver
OISCollider is not colliding with any valid receiver
Two Hand Active Colliding State
Full Output

Fig. OISTwoHandToolASM State Transition Diagram

Implemented OISActorStates

These will serve as explanations of how each of these states function. However, if you are only using the default State Machines of the framework, you won't ever have to use any of these nodes, as these nodes are created by default by the default state machines. You will only ever have to place these nodes if you want to create your own state machines.

ControllerIdleState

The idle state used for the OISSingleControllerASM. The State Machine will go to this state when the player grabs an object. It will exit this state and go into ControllerActiveState as soon as the player is no longer holding anything with their hand.

ControllerActiveState

The default state of the OISSingleControllerASM. As long as the player is not holding anything, this state will continue waiting for the player to collide with a receiver. If the player's hand collides with a receiver, the state will transition into ActiveCollidingState.

ToolIdleState

The default state of the OISOneHandToolASM. As long as it is not being held, it will remain in this state. This state will transition into ToolActiveState when the player holds the object.

ToolActiveState

The active state used for the OISOneHandToolASM. When the player is holding the object, it will remain in this state, and will wait for the object to collide with a receiver. If the object collides with a receiver, the state will transition into ActiveCollidingState.

ActiveCollidingState

When the actor is colliding with a receiver, it will be in this state. If an actor ever leaves collision, the State Machine will return to its respective active states. This State also handles the processing of the actor and receiver’s action.

ToolOneHandActiveState

A special Active State used for the OISTwoHandToolASM. It functions similarly to the regular ToolActiveState, but serves as the active state if the two hand tool is being held by only one hand.

ToolTwoHandActiveState

A special Active State used for the OISTwoHandToolASM. It functions similarly to the regular ToolActiveState, but serves as the active state if the two hand tool is being held by both hands.

OneHandActiveCollidingState

A special Active Colliding State used for the OISTwoHandToolASM. It functions similarly to the active colliding state, however, since it is being held by one hand, its output to the receiver will be halved.

TwoHandActiveCollidingState

A special Active Colliding State used for the OISTwoHandToolASM. It functions similarly to the active colliding state. Since the actor is being held by two hands, the output of the receiver will be full.

Implemented OISReceivers

As mentioned before, the default OISReceiverComponent node should not be used on its own. Instead any of the following OISReceiverComponents can be used. Each of these receivers have their own unique parameters, but all share the common functions and export variables of the default OISReceiverComponent.

OISWipeReceiver

This receiver takes any movement from the actor so long as the movement is sufficiently fast enough.

Additional Export variables:

TYPE	NAME	DESCRIPTION
float	buffer	Serves as the minimum value of the movement per frame that will add progress to the action. A buffer of 0

		means that any movement will contribute to the action.

OISTwistReceiver

This receiver takes a rotating/twisting action from the receiver.

Additional Export variables:

TYPE	NAME	DESCRIPTION
enum	twist_direction	Determines the direction of the twisting action that contributes to action progress. (Clockwise or Counter Clockwise)
bool	single_direction	A boolean that determines whether progress will only be affected by the chosen twist direction. If false, you can have negative progress.

OISStrikeReceiver

This receiver takes a striking action from the actor. Essentially a motion swinging towards the receiver.

Additional Export variables:

TYPE	NAME	DESCRIPTION
float	strike_range	The range in which the actor will hit the receiver’s center.

OISDirectionalSwipeReceiver

Similar to the wipe action, but a specific direction can be specified so that the receiver will only receive wipe actions along that direction.

Additional Export variables:

TYPE	NAME	DESCRIPTION
float	buffer	Serves as the minimum value of the movement per frame that will add progress to the action. A buffer of 0 means that any movement will contribute to the action.
Vector3	swipe_direction	A Vector3 that represents a 3D vector of the direction of the swipe action. The vector can be any magnitude as it is normalized when checking the direction.

OISCrankReceiver

This receiver takes a revolution around a center point along a specific axis from the actor.

Additional Export variables:

TYPE	NAME	DESCRIPTION
enum	twist_direction	Determines the direction of the cranking action that contributes to action progress. (Clockwise or Counter Clockwise)
bool	single_direction	A boolean that determines whether progress will only be affected by the chosen twist direction. If false, you can have negative progress.

Vector3	axis_of_rotation	A Vector3 that represents the axis the action will be revolving around.

OISAttachReceiver

This receiver allows for two objects to be attached to each other by pressing the trigger button on both controllers. This deletes the two objects being attached to each other and instantiates a new one.

Additional Export variables:

TYPE	NAME	DESCRIPTION
float	buffer	Determines the distance between the objects before they are allowed to be attached.
bool	is_primary_attacher	Determines which object is the primary attacher. A primary attacher can only be attached to a non-primary attacher.
String	replacement_object_path	The file-system path of the object that will replace the two objects that are going to be attached.

OISDetachReceiver

Detaches an object into two separate objects by doing a motion where you move your hands away from each other while both hands are holding the object. It deletes the original object and instantiates two new objects.

Additional Export variables:

TYPE	NAME	DESCRIPTION

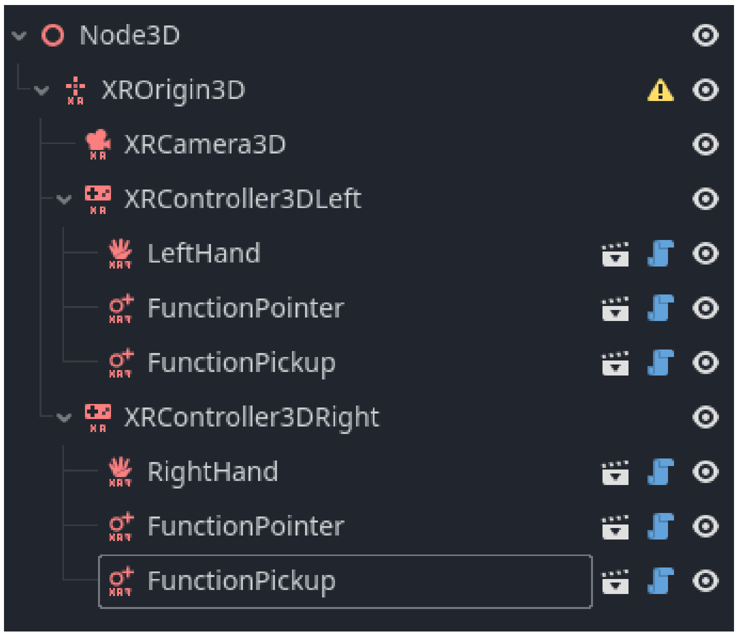
String	primary_replacement_object_path	The file-system path of one of the objects that will replace the object that is going to be detached. If the replacement object has an attach receiver, it will be set as the primary attacher.
String	secondary_replacement_object_path	The file-system path of one of the objects that will replace the object that is going to be detached.

OIS Tutorials

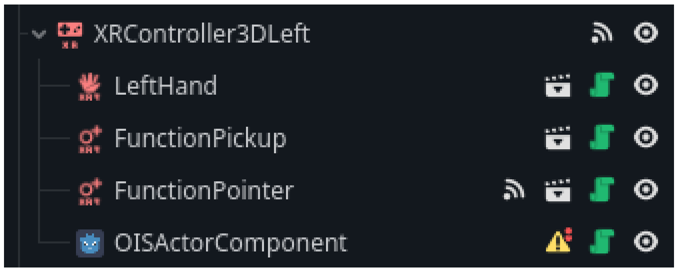
Turning the Player's Hand into an Actor

This section will guide you on how to turn the player's controller/hand into an Actor.

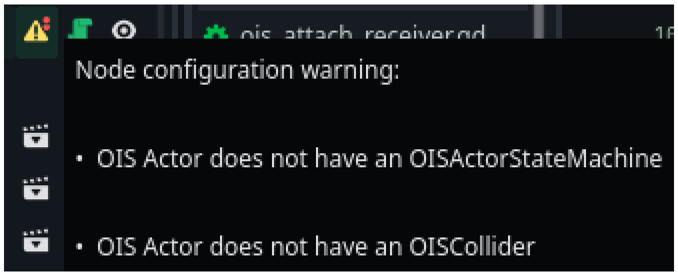
Recall the node hierarchy when creating your XRPlayer.



To turn your Controller into an Actor, the first step is to add an OISActorComponent node as the child of the XRController3D nodes.

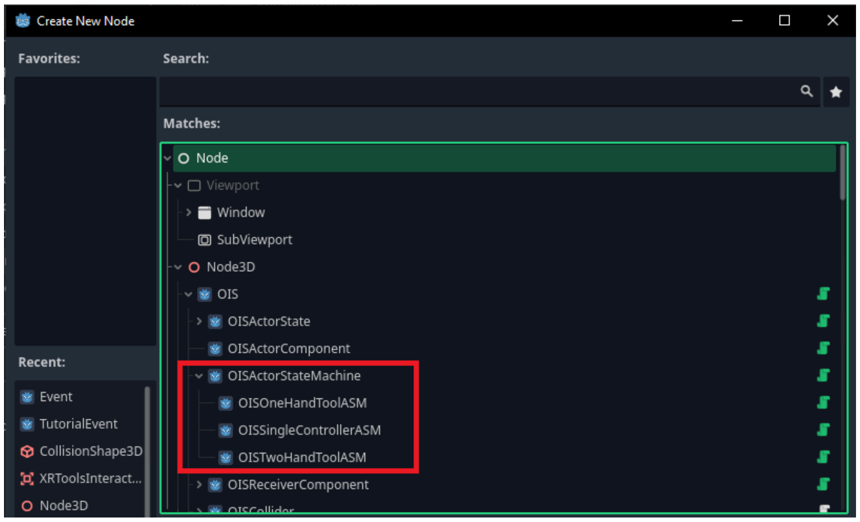

XRController3DLeft
LeftHand
FunctionPickup
FunctionPointer
OISActorComponent

This node will start out with 2 warnings that say that it does not have an OISActorStateMachine and an OISCollider.

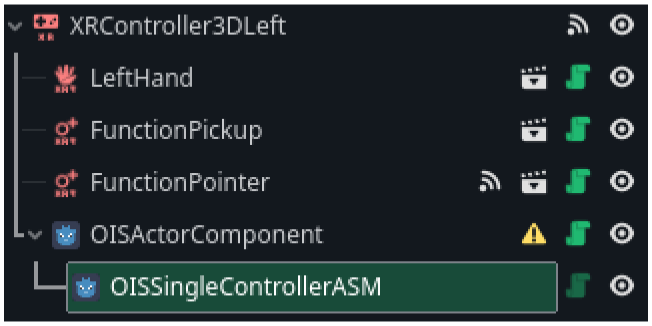

Node configuration warning:

- OIS Actor does not have an OISActorStateMachine
- OIS Actor does not have an OISCollider

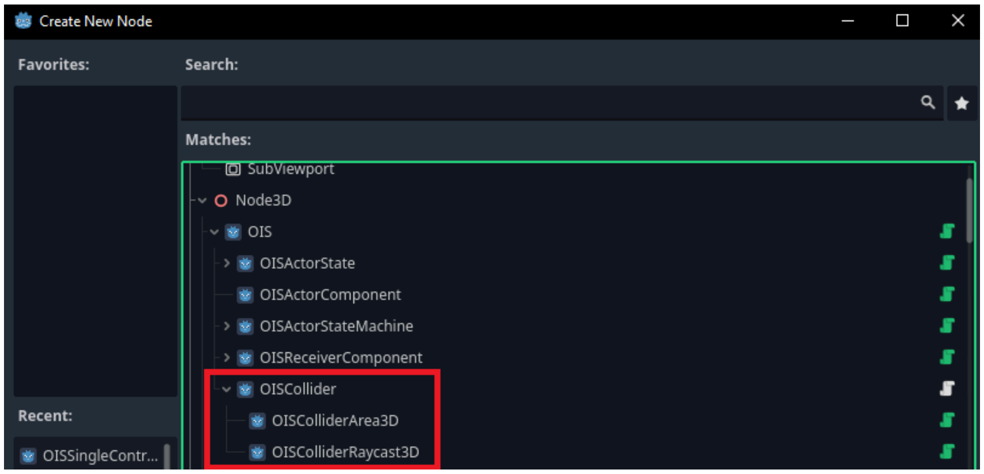
To fix these warnings, we first add an OISActorStateMachine. When you go to the Create Node window, you will notice that OISActorStateMachine will have several nodes that extend it.

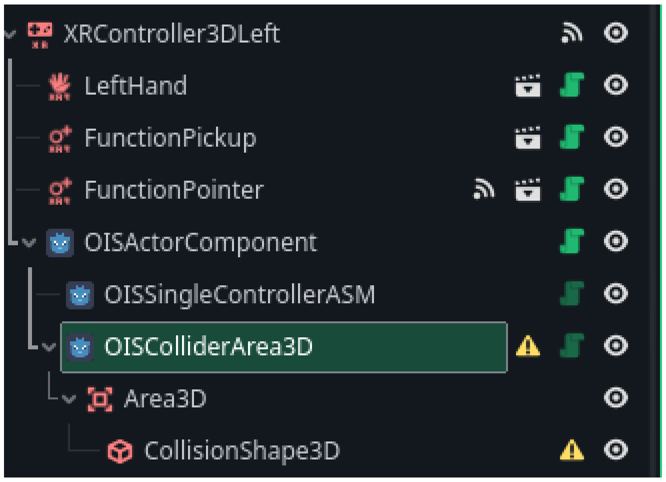


DO NOT use the OISActorStateMachine node. Instead, select OISSingleControllerASM, and the first warning should be removed.

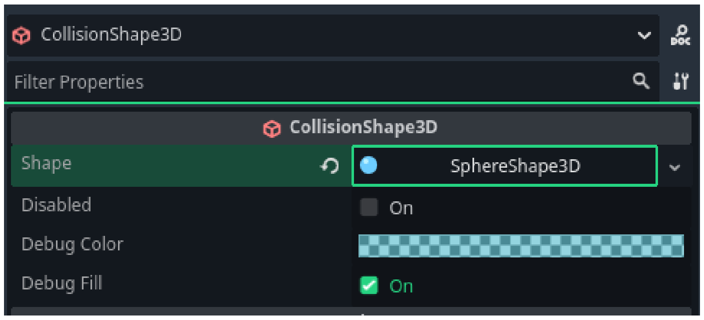


Then, to remove the second warning, an OISCollider should be added. Similar to the OISActorStateMachine, there will be several nodes that extend OISCollider. Again, DO NOT use OISCollider on its own. For now, select OISColliderArea3D.





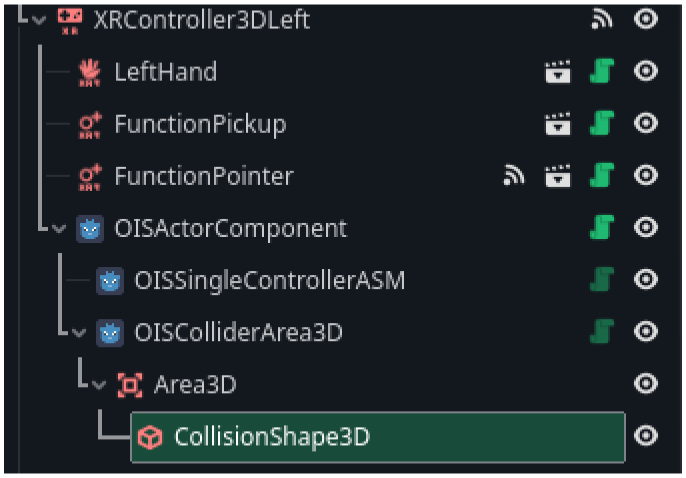
The OISColliderArea3D would have a warning, but this could be resolved by adding a shape to its CollisionShape3D grandchild.


CollisionShape3D

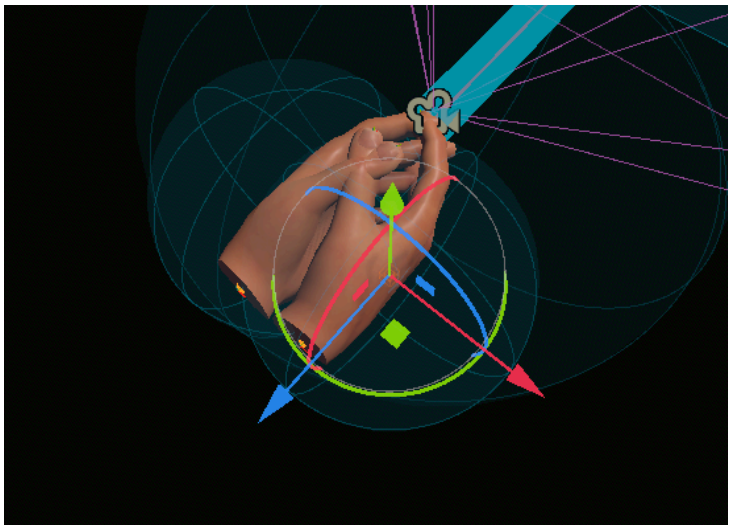
Filter Properties

CollisionShape3D

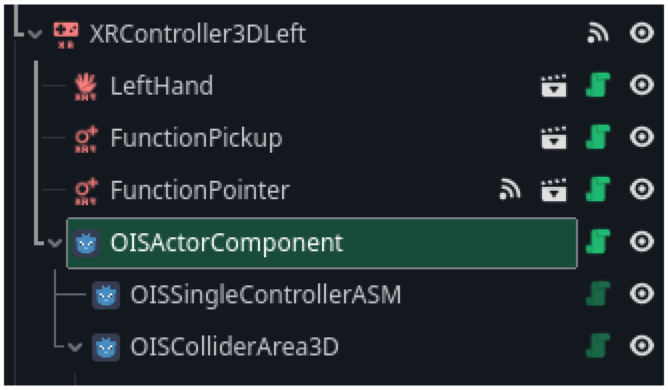
Shape	SphereShape3D
Disabled	On
Debug Color	▨
Debug Fill	✓ On

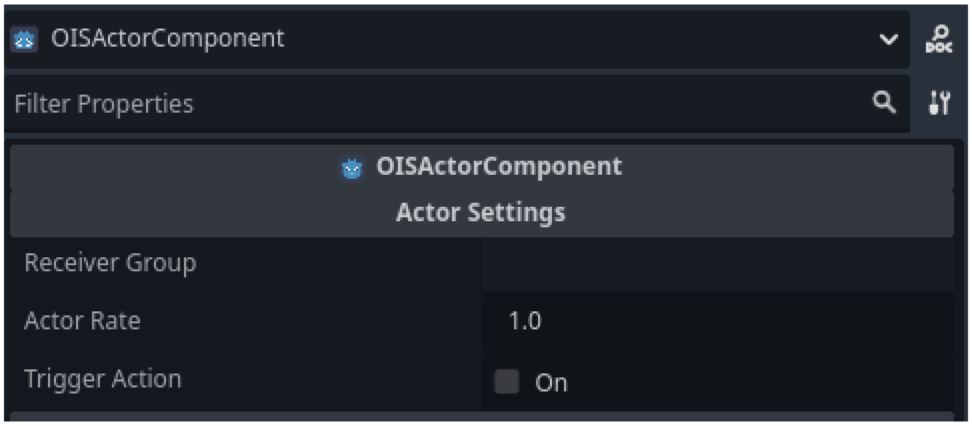

```
XRController3DLeft
    LeftHand
    FunctionPickup
    FunctionPointer
    OISActorComponent
        OISSingleControllerASM
        OISColliderArea3D
            Area3D
                CollisionShape3D
```

Now you should be seeing no warnings. You may move the OISCollider around to fit it around where the player's hand is.



There is one last thing you need to do to make your controller/hand into an Actor. Select the OISActorComponent node in the Scene Tree, and the Inspector should show the OISActorComponent settings.




OISActorComponent

Filter Properties

OISActorComponent

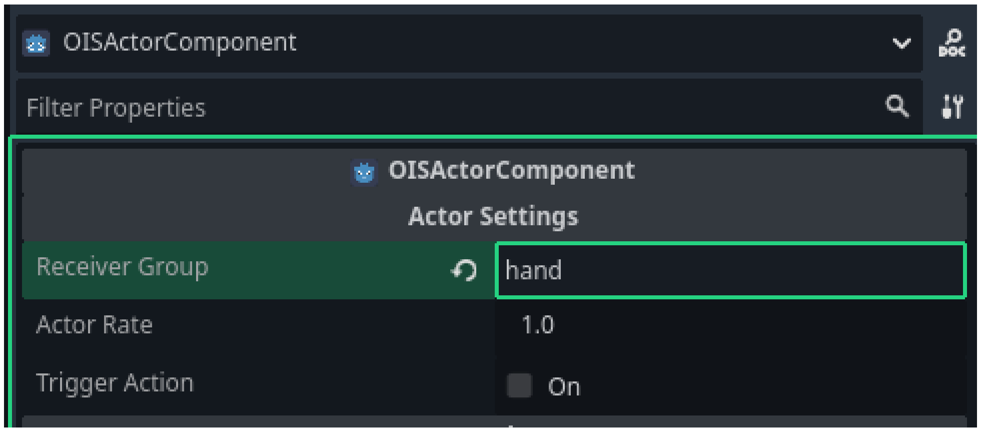
Actor Settings

Receiver Group

Actor Rate 1.0

Trigger Action On

Just add a group in the Receiver Group text field, and it should be done. In this case, “hand” is set as the Receiver Group.


OISActorComponent

Filter Properties

OISActorComponent

Actor Settings

Receiver Group hand

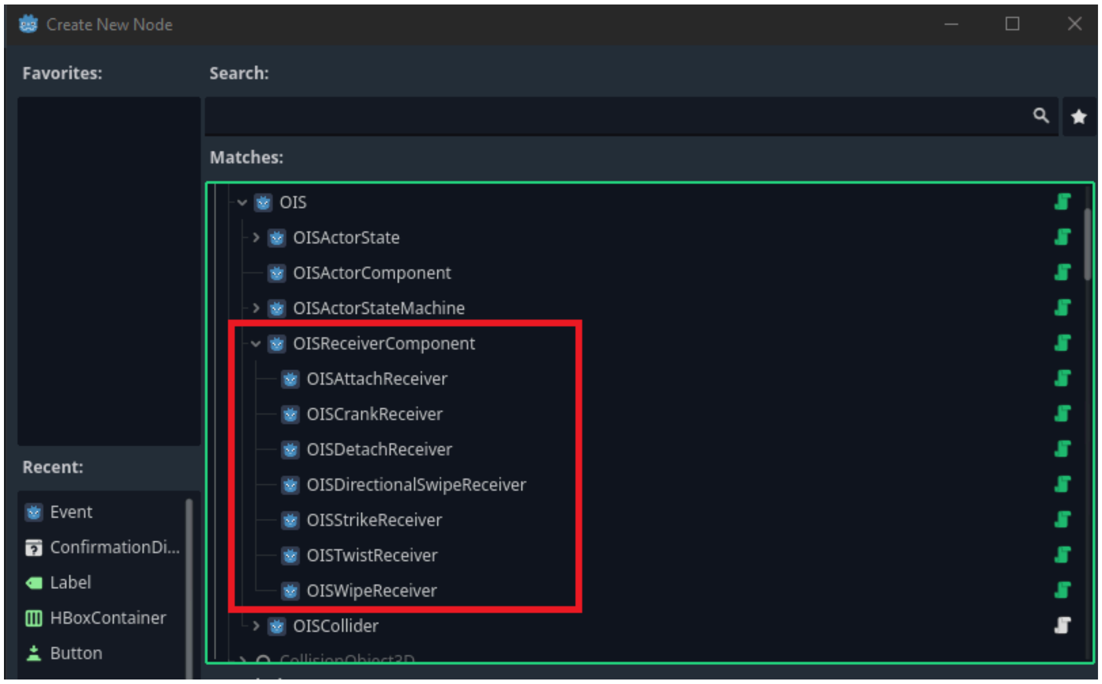
Actor Rate 1.0

Trigger Action On

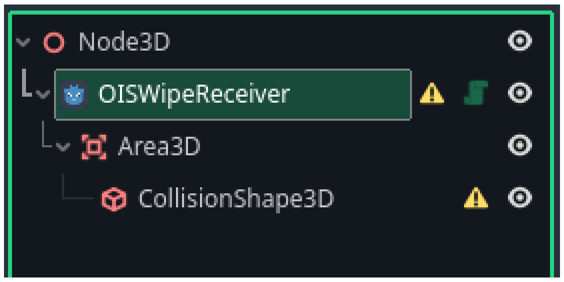
Now, any receiver with “hand” as its group will receive actions from the player’s controller/hand.

Creating a Receiver

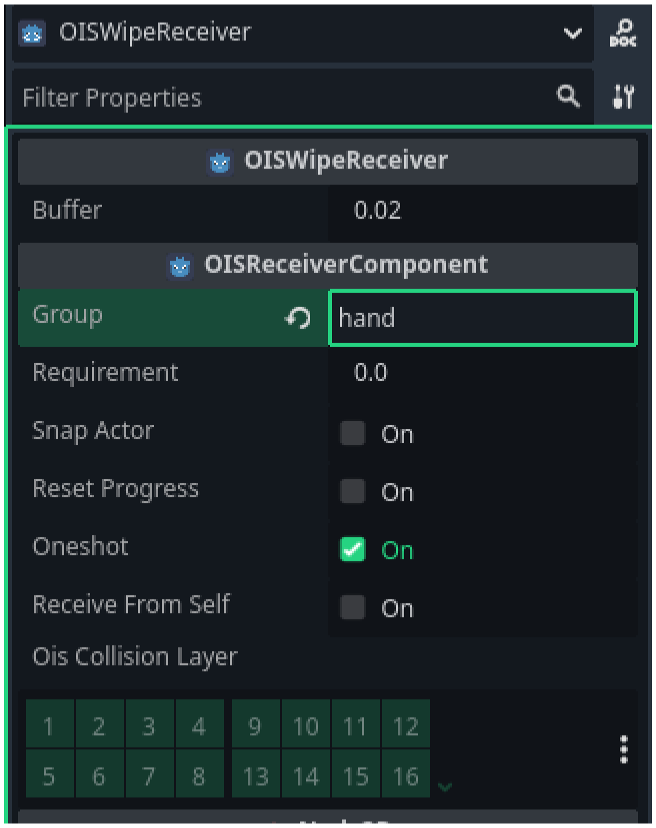
A receiver can be added to any node, even nodes that are actors. To make a receiver, add a node in the scene tree, and look for OISReceiverComponent. DO NOT add the OISReceiverComponent on its own, unless you plan on creating your own receiver. Instead, choose any of the pre-built nodes that extend OISReceiverComponent.



Once you add the node, you will be met with a warning that tells you to add a shape to the CollisionShape3D. Adding a shape will remove the warning.



To make the receiver work, select the receiver node, and edit the parameters on the inspector. In the Group parameter, just add any existing group you have set for any of your actors. In this case, the receiver can receive actions from the hand actor that was made previously.



OISWipeReceiver

Filter Properties

OISWipeReceiver

Buffer	0.02

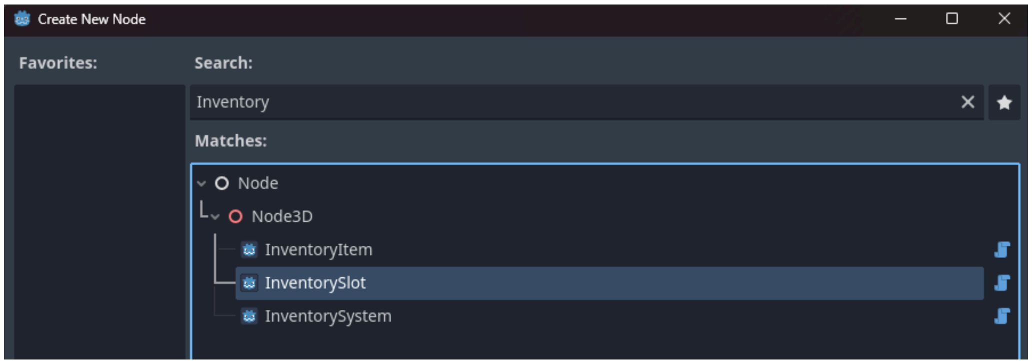
OISReceiverComponent

Group	hand
Requirement	0.0
Snap Actor	☐ On
Reset Progress	☐ On
Oneshot	✓ On
Receive From Self	☐ On

Ois Collision Layer

Inventory System

The AVRE Framework's Inventory System makes use of three nodes, the **InventorySystem** node, the **InventorySlot** node, and the item-applied **InventoryItem** node.



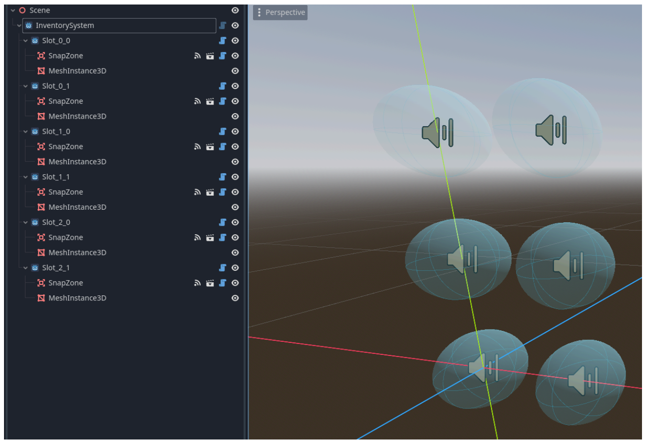
Search: Inventory

Matches:

```
Node
└ Node3D
   InventoryItem
   InventorySlot
   InventorySystem
```

InventorySystem

The **InventorySystem** node is a standalone node that handles a number of InventorySlot nodes, and can be configured according to the needs of the developer.



An InventorySystem node right after instantiation.

On instantiation, the InventorySystem node will come with 6 InventorySlot nodes, in a 3-row, 2-column layout.

The InventorySystem node requires some setting up after a XRPlayer node has been set up, which can be configured by the developer in the Inspector. The different settings for InventorySystem will be explained below**.

Properties

Export variables

****Bolded items must be configured by the developer before runtime.**

TYPE	NAME	DESCRIPTION
bool	reinitialize_inventory	Allows the developer to reinitialize the inventory system when the number of rows or columns are changed. Once enabled, it automatically disables.
int int	row_count column_count	Changes the number of rows and columns of the inventory system, updates in realtime.
int int	row_spacing column_spacing	Adjusts the spacing between slots in between rows and/or columns, updates in realtime.
float	update_slot_radius	Allows the developer to change all the radii of the slots at once when it is changed. Once enabled, it automatically disables.
float	slot_size	Changes the radii of all slots in the InventorySystem, needs Update slot radius to be checked to take effect.
bool	follow_player_camera	Allows the developer to set if the inventory system should follow the player camera if enabled. Enabled by default.

XROrigin3D	**xr_anchor_to_object**	**REQUIRES A XROrigin3D node.** **NOT REQUIRED if Follow Player Camera is disabled.** In order to allow the inventory to follow the player, the position of the player node is taken into account – thus this variable. Assign an XROrigin3D node from the player node to this variable.
XRCamera3D	**xr_camera_anchor**	**REQUIRES A XRCamera3D node.** **NOT REQUIRED if Follow Player Camera is disabled.** In order to allow the inventory to follow the player, the position of the camera node is taken into account – thus this variable. Assign a XRCamera3D node from the player node to this variable.
XRController3D	**inventory_toggler_hand**	**REQUIRES A XRController3D node.** Allows the player to enable/disable the inventory system with an assigned button to a specific controller.
String	**inventory_toggler_button**	A string that determines which button on the controller needs to be pressed to enable/disable the inventory system. **Refer to the OpenXR action map for possible button assignments.**

AudioStreamPlayer3D	audio_node	Allows the developer to assign a sound effect when players enable/disable the inventory system. Makes use of an AudioStreamPlayer3D node. Check the documentation of AudioStreamPlayer3D for its usage.
Vector3	local_transform_adjustment	A Vector3 variable that allows the player to offset the inventory system from the player.
float	slots_distance_to_player	A float that sets the distance of the slots from the player.
float	height_adjustment	A float that sets the height of the inventory system from the ground.
float	camera_window_width	A float that sets the threshold for the camera before it moves again. Default value is 1.

Other accessible variables

TYPE	NAME	DESCRIPTION
Dictionary	inventory_dictionary	Allows the developer to access the current inventory system contents as a Dictionary.
var	inventory_toggled	A boolean for determining whether or not the inventory is visible or not.
var	space_count_row	Row count.
var	space_count_column	Column count.

Vector3	position_offset	Accessible Vector3 for the offset of the inventory system node.
float	prev_camera_rotation	Value of the previous camera rotation, intended for the non-continuous inventory following option.
float	current_camera_rotation	Current value of the camera rotation, intended for the non-continuous inventory following option.

Methods and Functions

RETURN TYPE	NAME	DESCRIPTION
void	initialize_inv()	Initializes the inventory system using default parameters.
void	_update_inventory_system()	Only intended for the Editor, allows for some changes to the InventorySystem to be reflected in realtime when editing.
void	clear_all_children()	Clears all children of the InventorySystem, intended for resetting the node.
void	update_slot_item(what, row, col)	Updates the InventorySystem's dictionary content with the object file path, and which slot row and column it is placed in.
void	center_inventory()	Centers the inventory to where the player is looking.
void	reposition_inventory(delta :float)	This method is called every frame in the physics_process method to ensure that the inventory system follows where the player looks.

Dictionary	export_save_data()	Returns the contents of the inventory system in a Dictionary.
void	import_save_data(data: Dictionary)	Allows import of a specified format of Dictionary into the InventorySystem

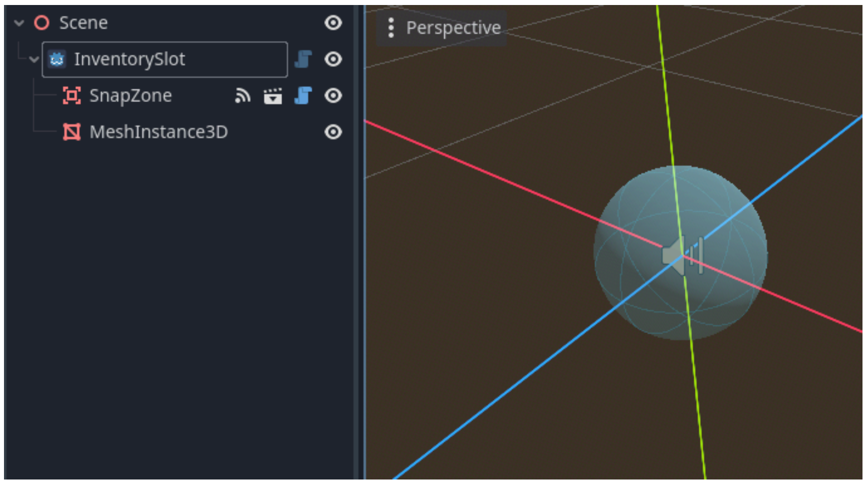
After set up, the InventorySystem node should be usable during runtime.

InventorySlot

The **InventorySlot** node is a standalone node that acts as a slot for pickable items. Pickable items can be placed in an InventorySlot node. An InventorySlot node can be placed anywhere needed.

Works hand-in-hand with the InventoryItem node to provide more effects for the pickable item but does not necessarily require pickables to have InventoryItem.

The InventorySlot node works immediately once placed into the scene.



An InventorySlot shown in the Editor

An InventorySlot node is also configurable by the developer. These settings are explained below:

Properties

Signals

SIGNAL	DESCRIPTION
current_object_in_slot(object, row, col)	Signal emitted with information regarding an object, and which slot, identified by which specific row and column in the inventory system is it placed in.
slot_picked_up	Signal emitted when the slot picks up an object.
slot_dropped	Signal emitted when the slot drops its object.

Export variables

TYPE	NAME	DESCRIPTION
bool	update_slot_settings	Allows the developer to reinitialize the inventory slot for changes (e.g. slot radius, slot material)
bool	slot_enabled	Set whether or not the developer wants the slot enabled or disabled.
float	snap_zone_radius	Radius of which the slot can detect pickables and pick it up. Must activate “Update slot settings” for changes to reflect.
XRToolsPickable	default_object	Default object of the slot at runtime. Must be an XRToolsPickable node.
String	group_required	The slot can only pick up objects in the group specified.

bool	funny_effect	Rotates the contents of the slot. Disabled by default.
bool	ignore_inventory_item_scale	Only works if InventoryItem is applied to a pickable item. Ignores the shrink/enlarge size set in InventoryItem and keeps the current item scale when put into the slot.
StandardMaterial3D	slot_material_override	Allows the developer to change the material of the sphere for the InventorySlot. Must activate “Update slot settings” for changes to reflect.

Other accessible variables

TYPE	NAME	DESCRIPTION
MeshInstance3D	snap_zone_mesh	Mesh intended for the snapzone.
SphereMesh	mesh_shape	Mesh intended for the visible shape for the slot.
Scene	snap_zone_scene	Loaded XRToolsSnapZone scene.
XRToolsSnapZone	snap_zone	An instantiated SnapZone from XRTools

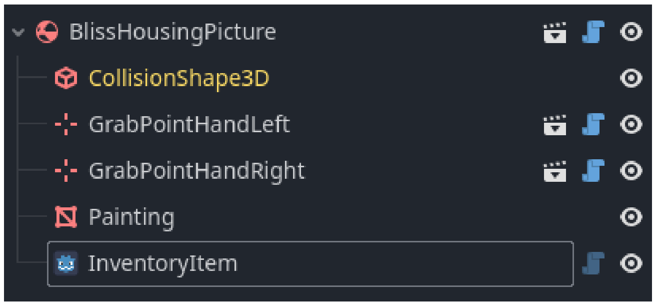
Node3D	current_object	Object currently in slot, if no object is slot, this is null.
bool	is_parented	Whether or not the slot is parented to an InventorySystem node.

Methods and Functions

RETURN TYPE	NAME	DESCRIPTION
void	_set_current_slot_object(what)	Method intended for setting the slot object and responsible for emitting various signals.
void	_drop_current_slot_object()	Drops currently held Pickable.
void	_body_entered_area(body: Node3D)	Determines if a body has entered the area and checks if it collides with the slot shape.
void	_body_exited_area(body: Node3D)	Determines if a body has exited the area and checks if it no longer collides with the slot shape.
void	_pick_up_object(body: Node3D)	Picks up an object and calls other relevant methods.
void	_pick_up_object_init(body: Node3D)	Picks up an object in a different method, usually intended for debug purposes.

InventoryItem

The **InventoryItem** is an additional node that can be placed within a pickable item scene. This node can provide additional functionalities such as shrinking/enlarging the item when it comes into contact with slots, or ensuring that the item is unique.



An InventoryItem node shown as a child of a pickable object scene.

Pickable items with the InventoryItem node can be configured, as explained below**:

Properties

Export variables

****Bolded items must be configured by the developer before runtime.**

TYPE	NAME	DESCRIPTION
bool	unique	Allows the developer to set if the item is unique, meaning only one item of its kind can exist within the scene with respect to the InventorySystem.
Node3D	**defined_mesh**	Mesh of the pickable. Must be set to ensure the slot works properly.
CollisionShape3D	**defined_collision_shape**	Collision mesh of the pickable. Must be set to ensure the slot works properly.

float	preferred_scale	Scale at which the pickable transforms when it collides with a slot. Default at 0.5 (half the size of the original object)
Vector3	object_transform_adjustment	Moves the pickable mesh and collider once inside the slot, intended for offsetting and proper object placement purposes.
Vector3	object_rotation_adjustment	Rotates the pickable mesh and collider once inside the slot, intended for offsetting and proper object placement purposes.
Node3D	additional_mesh	Set an additional separate mesh that should also be affected by the shrinkage/enlargement of the pickable item once it comes in contact with a slot.
bool	exclude_additional_mesh_transform	Exclude the additional mesh from shrinkage/enlargement when the pickable item comes in contact with a slot.
bool	has_custom_shrink_position	Must be set if the item is too wide or large and the position of the object is too far from the hand once it shrinks/enlarges.

Node3D Node3D	grab_point_right grab_point_left	These points will be used as the offsets for the custom shrink position if "Has Custom Shrink Position" is enabled.

Other accessible variables

TYPE	NAME	DESCRIPTION
Vector3	preserved_mesh_scale	Vector3 containing the original scale of the mesh.
Vector3	preserved_collider_scale	Vector3 containing the original scale of the mesh collider.
Vector3	preserved_mesh_transform	Vector3 containing the original transform of the mesh.
Vector3	preserved_collider_transform	Vector3 containing the original transform of the mesh collider.
Vector3	preserved_mesh_rotation	Vector3 containing the original rotation of the mesh.
Vector3	preserved_collider_rotation	Vector3 containing the original rotation of the mesh collider.
Vector3	addt_preserved_mesh_scale	Vector3 containing the original scale of the additional mesh.

Vector3	addt_preserved_mesh_transform	Vector3 containing the original transform of the additional mesh.
Vector3	addt_preserved_mesh_rotation	Vector3 containing the original rotation of the additional mesh.
bool	is_resized	Boolean determining if the object is already resized or not.
bool	body_collision_detected	Boolean determining if a collision with another body is detected.
bool	slot_interaction_detected	Boolean determining if a collision with an InventorySlot is detected.
bool	is_in_slot	Boolean to check if this object is in a slot.
bool	is_grabbed	Boolean to check if this object is being grabbed.
Array	is_colliding_with	Boolean to check if this object is colliding with what objects. All colliding objects are stored in this accessible Array.
Vector3	shrink_position	Vector3 determining the shrink position when custom shrink position is enabled.

Methods and Functions

RETURN TYPE	NAME	DESCRIPTION
void	_resize_mesh(scalex : float)	Resizes the mesh to a specified percentage.
void	_on_out_slot_transform()	Reverts object to original scale, rotation and transformation.
void	_on_in_slot_transform()	Transforms object to specified scale, rotation and translation.
void	_force_enlarge_item()	Forces the object to revert to its original scale.
void	_force_shrink_item()	Forces the object to shrink to the specified scale.
void	_grabbed(pickable: Variant, by: Variant)	Determines if the object is grabbed by which hand, and changes the shrink position with respect to which hand grabbed the object.
void	_released(pickable: Variant, by: Variant)	Reverts all transforms to the original, and is intended for when custom shrink position is enabled.

Other accessible variables

TYPE	NAME	DESCRIPTION
MeshInstance3D	snap_zone_mesh	Mesh intended for the snapzone.

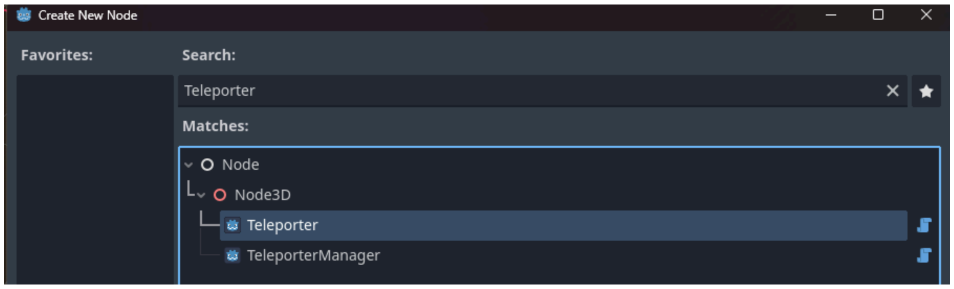
SphereMesh	mesh_shape	Mesh intended for the visible shape for the slot.
Scene	snap_zone_scene	Loaded XRToolsSnapZone scene.
XRToolsSnapZone	snap_zone	An instantiated SnapZone from XRTools
Node3D	current_object	Object currently in slot, if no object is slot, this is null.
bool	is_parented	Whether or not the slot is parented to an InventorySystem node.

Teleportation / Locomotion System

The AVRE Framework's Teleportation System consists of two types of nodes, the **TeleporterManager** node and the **Teleporter** node.

This subsystem is optional and can be replaced with a different traversal system should the developer wish to.

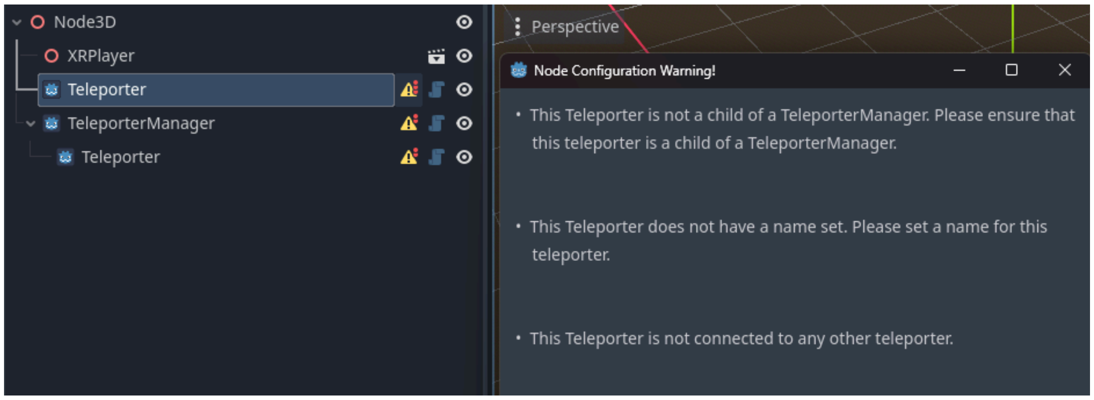
Please note that the XRPlayer node must be set up first before making use of this subsystem.



Teleporter

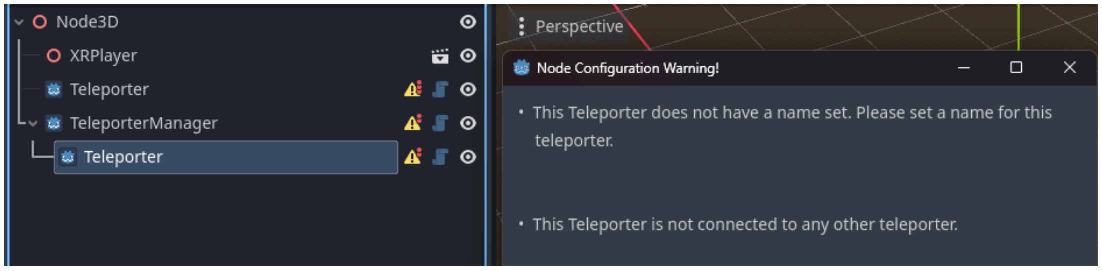
The **Teleporter** node contains the details needed by the TeleporterManager to teleport the player around the scene.

On instantiation, the Teleporter node will come with a warning.
This node **MUST BE A CHILD** of a **TeleporterManager** node, otherwise, an additional warning will be shown that must be addressed.



- This Teleporter is not a child of a TeleporterManager. Please ensure that this teleporter is a child of a TeleporterManager.
- This Teleporter does not have a name set. Please set a name for this teleporter.
- This Teleporter is not connected to any other teleporter.

A Teleporter node that is not a child of TeleporterManager showing an additional warning.



- This Teleporter does not have a name set. Please set a name for this teleporter.
- This Teleporter is not connected to any other teleporter.

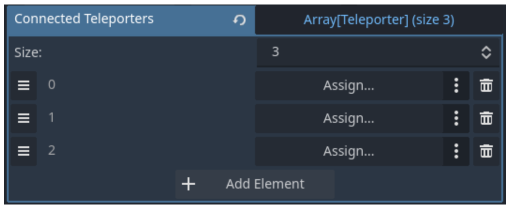
A Teleporter node that is a child showing less warnings.

The rest of the warnings for the Teleporter node will be addressed once the Teleporter node is properly configured in the Inspector.
The properties of the Teleporter node are explained below:

Properties

Export variables

TYPE	NAME	DESCRIPTION
String	teleporter_name	The name of the teleporter. It is recommended to name the teleporter to its corresponding location for easier access. Teleporter default name is "Teleporter", it is recommended to change this to remove a warning during instantiation.
bool	teleporter_enabled	Whether or not the teleporter is enabled.
bool	teleporter_active	A signal-dependent boolean intended for events.
Vector3	teleporter_position	A Vector3 variable that takes the current position of the node and uses it as a reference for the teleportation point. (Does not need to be changed)

Vector3	teleporter_rotation	A Vector3 variable that takes the current rotation of the node and uses it as a reference for the teleportation point. (Does not need to be changed)
Array[Teleporter]	connected_teleporters	An array of Teleporters that are connected to this specific Teleporter. Developers can define the array size to their requirements and set the teleporters to each array element. This must be set to remove one of the warnings during instantiation.  Connected Teleporters Array[Teleporter] (size 3) Size: 3 0 Assign... 1 Assign... 2 Assign... Add Element
Vector3	spectator_camera_position	**(Optional - only relevant if a spectator camera is assigned to the TeleportManager)** The position of the spectator camera when teleporting to this specific teleporter.
Vector3	spectator_camera_rotation	**(Optional - only relevant if a spectator camera is assigned to the TeleportManager)** The rotation of the spectator camera when teleporting to this specific teleporter.

Other accessible variables

TYPE	NAME	DESCRIPTION

MeshInstance3D	default_teleporter_mesh	Mesh instance for the teleporter body.
CylinderMesh	default_mesh_shape	Default mesh shape.
MeshInstance3D	default_teleporter_arrow	Mesh instance for the directional arrow.
PrismMesh	arrow_mesh	Mesh intended to serve as a directional arrow.
StaticBody3D	default_static_body	Default static body intended for objects with collision.
CollisionShape3D	default_collider	Default collider mesh instance.
CylinderShape3D	default_collider_shape	Default collider shape.
StandardMaterial3D	teleporter_material_override	Default material for the teleporter.
bool	current_teleporter	Boolean that checks if the teleporter is currently the one where the player is standing.
bool	aimed_at	Boolean that checks if the teleporter is being aimed at with FunctionPointer.

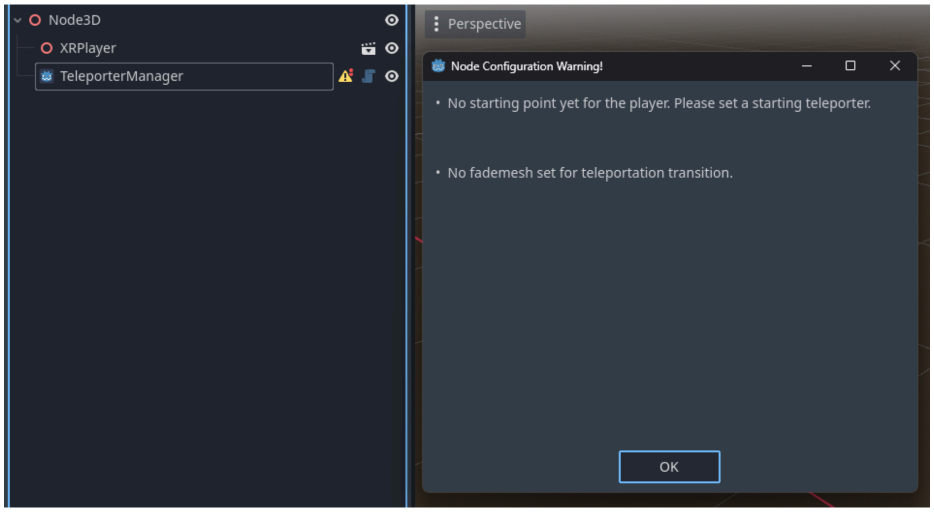
Methods and Functions

RETURN TYPE	NAME	DESCRIPTION
void	_set_up_teleporter_mesh()	Sets up the teleporter mesh during initialization.
void	_update_teleporter_name()	Updates the teleporter name based on the entered String in the inspector. Is not changeable within the method.
void	_update_teleporter_state()	Updates the color of the material of the teleporter depending on its state (aimed at, current, active, inactive)
void	_update_connections()	Updates the array of teleporters to include all other teleporters that are connected to it.

TeleporterManager

The **TeleporterManager** node manages all Teleporter nodes in the scene. It is also responsible for moving the player around when they decide to teleport around the scene.

On instantiation, the TeleporterManager node will show warnings.



- No starting point yet for the player. Please set a starting teleporter.
- No fademesh set for teleportation transition.

An unconfigured TeleporterManager node.

These warnings can be fixed after configuring the TeleporterManager node, the settings of which will be explained below:

Properties

Signals

SIGNAL	DESCRIPTION
location_changed(location_name)	Signal emitted when the player teleports to another Teleporter.

Export variables

****Bolded items must be configured by the developer before runtime.**

TYPE	NAME	DESCRIPTION
Teleporter	**current_location**	The starting position
bool	enabled	If the teleporter manager is enabled or not.

String	**teleporter_trigger_button**	A string that determines which button on the controller needs to be pressed to enable/disable the inventory system. “Trigger_click” is assigned as the default value. **Refer to the OpenXR action map for possible button assignments.**
XROrigin3D **XRCamera3D** **XRToolsFunctionPointer** **XRToolsFunctionPointer**	**xr_origin** **xr_camera** **xr_left_function_pointer** **xr_right_function_pointer**	Auto-assigned if XRPlayer is present in the scene. If not, must be configured manually.
AudioStreamPlayer3D	audio_node	Allows the developer to assign a sound effect when players teleport. Makes use of an AudioStreamPlayer3D node. Check the documentation of AudioStreamPlayer3D for its usage.
Node3D	**fade_mesh**	A mesh that is being used that fades in or out when the teleporter is used. More details on how to set up a proper fade mesh in the section below.

Camera3D	spectator_camera	A third-person camera separate from the XR Camera being used by the player in their headset.
bool	update_connections	Update all teleporters and their connections.
Teleporter	pointing_at	Debug option, determines which teleporter is being pointed at by the player.
XRController3D	active_controller	Debug option, determines which of the two auto assigned controllers is active and will be prioritized.

Other accessible variables

TYPE	NAME	DESCRIPTION
XRController3D XRController3D	_controller_left_node _controller_right_node	Returns the XRController3D for the FunctionPointer associated from a hand.
SphereMesh	mesh_shape	Mesh intended for the visible shape for the slot.
bool	teleport_called	Boolean called if a teleporter is activated and teleported to.
bool	initial_teleport	Boolean called at runtime, when the game is started.

Methods and Functions

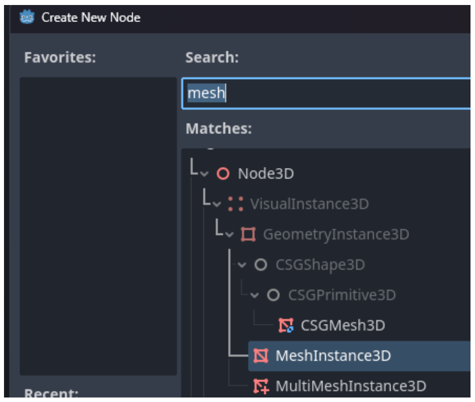
RETURN TYPE	NAME	DESCRIPTION
void	_set_teleporter_states()	Changes the states of teleporters depending on whether or not they should be active or inactive.
void	_teleport_player(teleporter: Teleporter)	Teleport the player to a specific Teleporter.
void	_teleport_spectator_camera(teleporter: Teleporter)	Teleport the assigned spectator camera to the defined spectator camera position when teleporting to a specific Teleporter
void	_runtime_pointer()	Method being called in the node's physics process method that checks if a Teleporter is being aimed at.
void	_initialize_xr_components()	Method to automatically set XR-related export variables. Called when in Editor.
void	_initialize_xr_origin_nodes(xr_origin_nodes : XROrigin3D)	Method to automatically detect both FunctionPointers from the hands from an XROrigin3D node. Called when in Editor.
void	_connect_controller_buttons()	Connects controller pressed signals to teleporter specific methods.
void	_fade_in()	Fades the screen in.

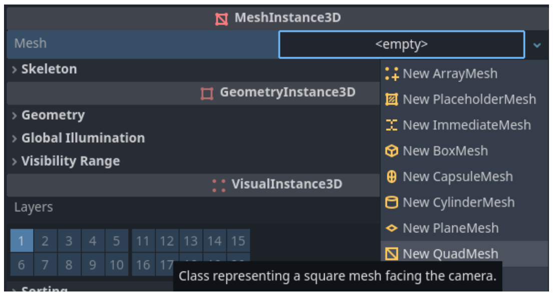
void	_fade_out()	Fades the screen to black using the defined fade mesh.

Setting up a proper fade mesh

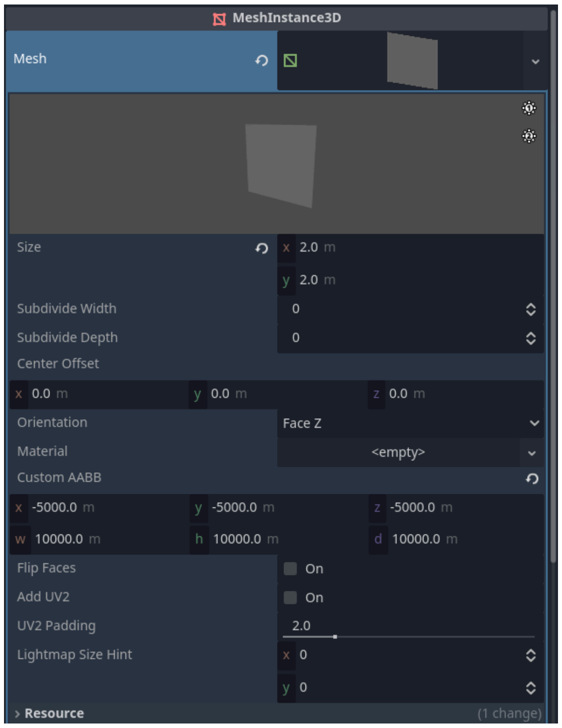
Setting up a proper fade mesh is important so that the TeleporterManager can properly fade in and fade out the screen when a teleportation is taking place.

In order to make a fade mesh, the developer must first make a **MeshInstance3D** and apply a new **QuadMesh** as the mesh in the inspector.





Then, the **QuadMesh** must be set to this configuration:



MeshInstance3D

Mesh

Size x 2.0 m y 2.0 m

Subdivide Width 0

Subdivide Depth 0

Center Offset x 0.0 m y 0.0 m z 0.0 m

Orientation Face Z

Material <empty>

Custom AABB x -5000.0 m y -5000.0 m z -5000.0 m w 10000.0 m h 10000.0 m d 10000.0 m

Flip Faces On

Add UV2 On

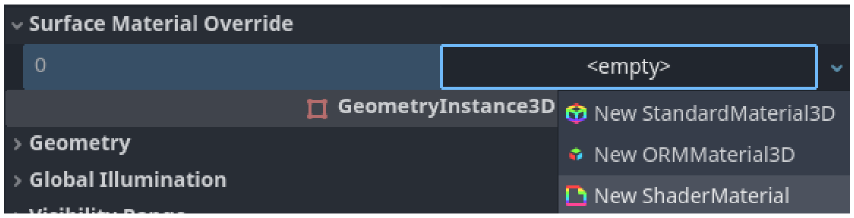
UV2 Padding 2.0

Lightmap Size Hint x 0 y 0

Resource (1 change)

- Size: x: 2.0m, y: 2.0m.
- Orientation: Face Z
- Custom AABB: x,y,z: -5000m, w,h,d: 10000m

Then, a material override for the surface must be applied on the mesh. Select a new **ShaderMaterial**.



Surface Material Override

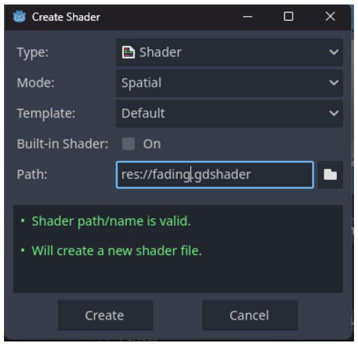
0 <empty>

GeometryInstance3D

Geometry

Global Illumination

Expand the material and click on Shader, and make a new Shader. Save the shader to a proper folder in your project directory.


Create Shader

Type: Shader

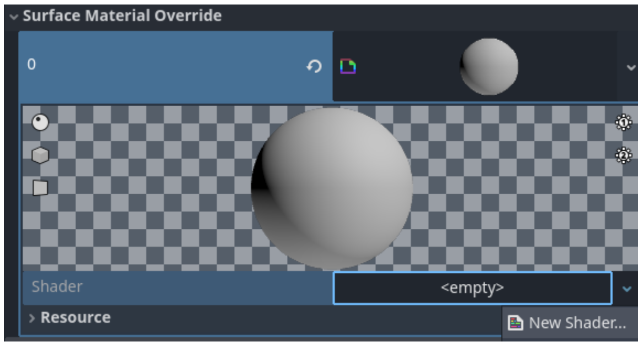
Mode: Spatial

Template: Default

Built-in Shader: On

Path: res://fading.gdshader

- Shader path/name is valid.
- Will create a new shader file.



Edit this shader file and use the following code and save.

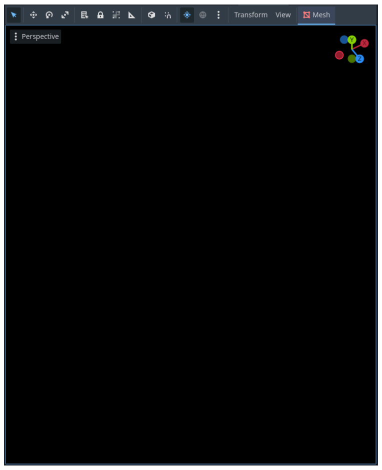
```
shader_type spatial;
render_mode      depth_test_disabled,      skip_vertex_transform,      unshaded,
cull_disabled;

uniform vec4 albedo : source_color;


void vertex() {
	POSITION = vec4(VERTEX.x, -VERTEX.y, 0.0, 1.0);
}

void fragment() {
	ALBEDO = albedo.rgb;
	ALPHA = albedo.a;
}
```

If correctly done, your editor viewport should look like this:

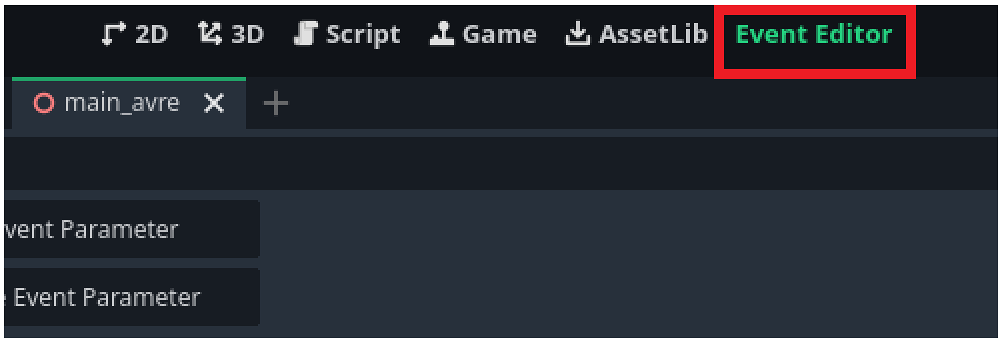


Now, this fade mesh can be used in the TeleporterManager in the Fade Mesh setting.

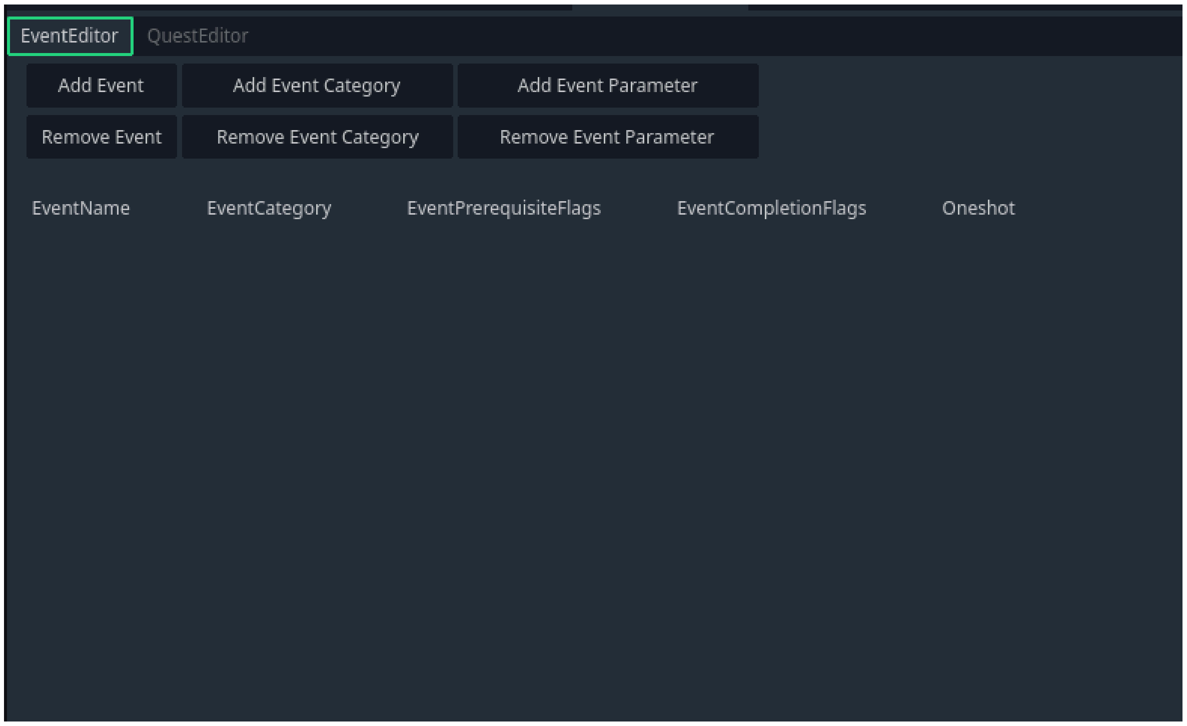
Event Management System

The Event Management System allows you to easily add events in your game that can all be connected to one another. These events can help you make certain things happen only at specific times, or when other events have already been completed, or even make a chain of events that the player has to follow.

The Event Management System can be found at the top-bar of the Godot Editor upon adding the AVRE addon.



This opens the Event Editor at the center of the Godot Editor.



This editor will allow you to start creating your events and quests.

Event Editor

There are 5 default parameters for events. **EventName, EventCategory, EventPrerequisiteFlags, EventCompletionFlags**, and **Oneshot.**

EventName is the event's name, which will be used to actually place the event in the game.

EventCategory can help with managing event behavior based on the category of event (Though this would require some coding on the developer's end).

EventPrerequisiteFlags are the flags required for the event to start. If those flags are not met, the event will not start. This is useful for ensuring that certain events can only occur when other events have already been completed.

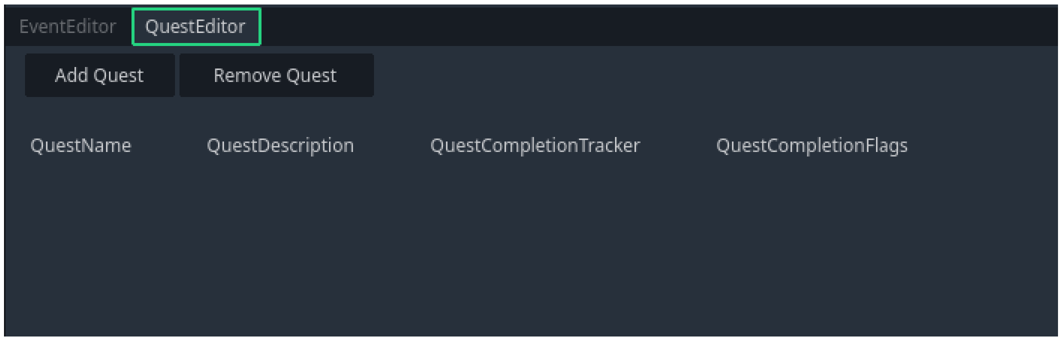
EventCompletionFlags are the flags that are triggered upon the completion of an event. All events will have a default completion flag of [EventName]_Done. Aside from that, you may also add additional flags.

Oneshot is a boolean that determines whether an event is repeatable or if it will only ever happen once.

Aside from the 5 default parameters, you have the option to add additional parameters based on the needs of your game, though using these additional parameters will require additional coding on the developer's part.

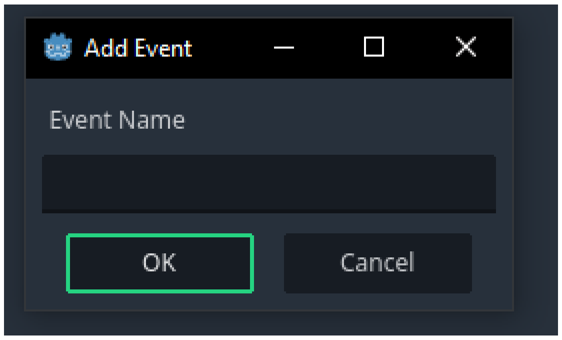
Quest Editor

The QuestEditor allows you to make quests which are a set of events. It allows you to assign events you have created and make a single complex chain of events. Similar to events, Quests also have completion flags which can be used as prerequisites to events.

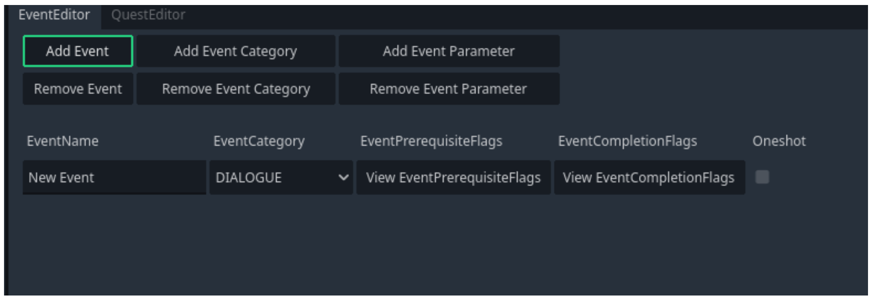


Creating Events

To create an event, just click on the Add Event button, and a popup will appear asking you to give the event name.



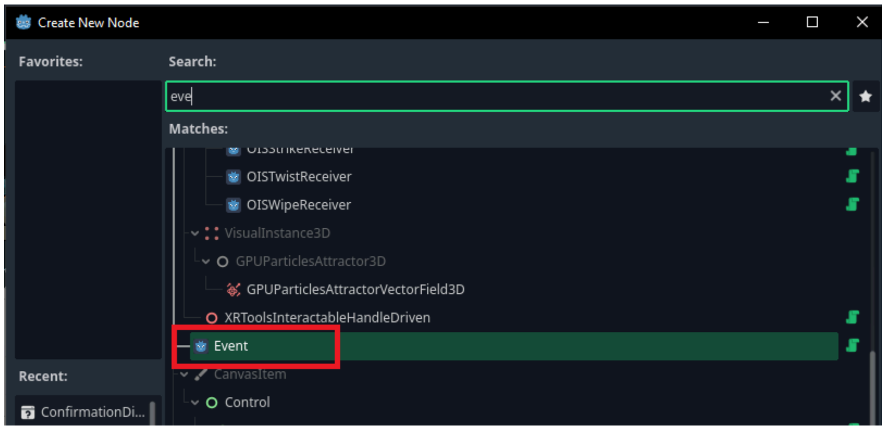
After inputting an event name and pressing OK, the event will appear on the editor.



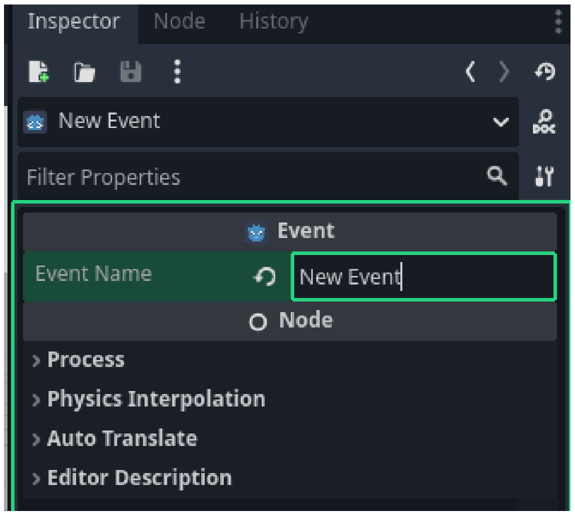
EventName	EventCategory	EventPrerequisiteFlags	EventCompletionFlags	Oneshot
New Event	DIALOGUE	View EventPrerequisiteFlags	View EventCompletionFlags	☐

You can edit the parameters here.

To actually add an event in the game, you will need to add the event node in the scene.

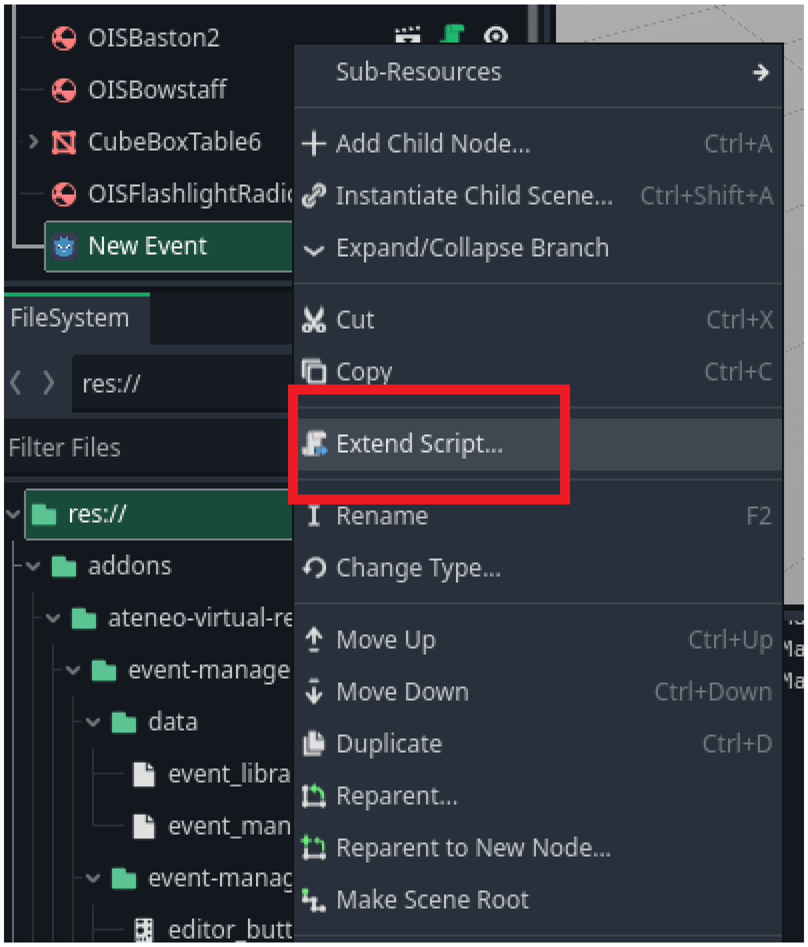


Then, select the event node and edit the event name in the inspector. Make sure that the name you use here matches the event name you want to add to the game.

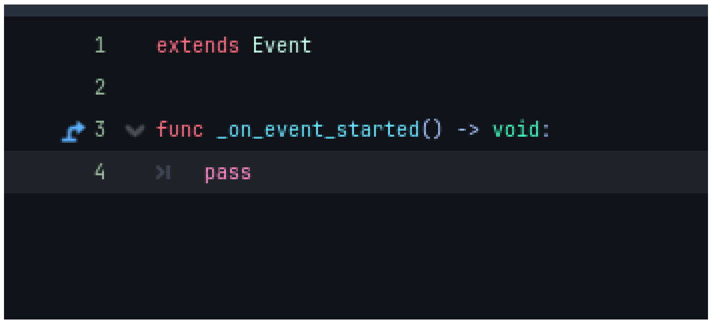


This event will now be activated when the game starts. To add functionality to the event, you will have to create a script for the event.

Right click on the Event Node in the Scene Tree and select Extend Script.



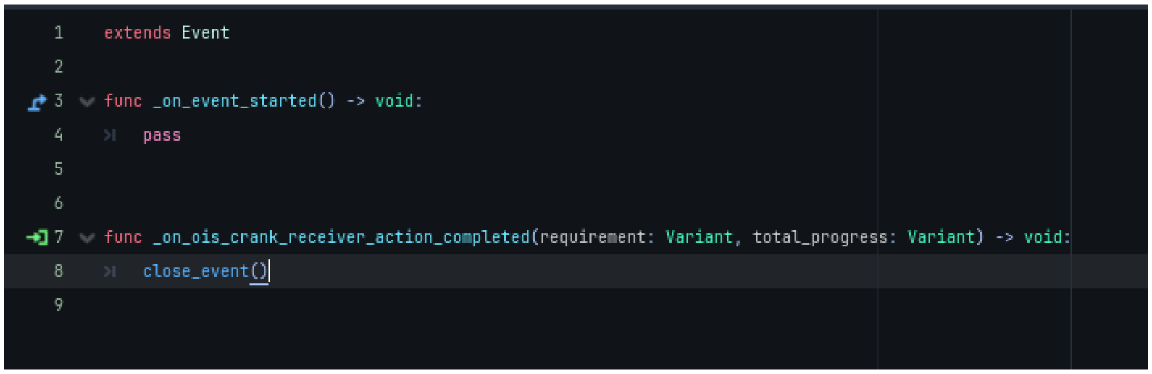
Which will create an empty .gd script that extends Event. Add the _on_event_started() function which is automatically called once an event starts to set up what you want to happen whenever the event starts.



```
extends Event

func _on_event_started() -> void:
	pass
```

To end the event, you will need to set that up yourself as well. An event can end because of many different things that can happen throughout your game. In any case, you will call the close_event() function whenever you want to end an event. In this example, the event is ended when a crank action is completed:



```
extends Event

func _on_event_started() -> void:
	pass


func _on_ois_crank_receiver_action_completed(requirement: Variant, total_progress: Variant) -> void:
	close_event()

```