Basic Concepts - Authority and RPC

By default, everything in RadGame has the Server as authority. The only (so far) Exception is the player_input script that is part of each player character - the client that's controlling the character has the multiplayer authority over his input script.

Movement

The player movement is handled in player.gd, which should have the server as authority. Here's how:

```
func _physics_process(delta) -> void:
    # movement
    if not is_dead:
        handle_movement(delta)
```

```
func handle_movement(delta: float) -> void:
    # jumping
    if input.jumping and is_on_floor():
        velocity.y = jump_velocity
    if not is_on_floor():
        velocity.y -= gravity * delta
    input.jumping = false
    var direction = (transform.basis * Vector3(input.direction.x, 0,
input.direction.y)).normalized()
    if direction:
        velocity.x = direction.x * speed
        velocity.z = direction.z * speed
    else:
        velocity.x = move_toward(velocity.x, 0, speed)
        velocity.z = move_toward(velocity.z, 0, speed)
    move_and_slide()
```

In handle_movement, the direction gets read from the player_input.gd script, over which the player has authority. the player_input node in the player scene is actually a multiplayer synchronizer, and syncs the "direction" variable to the server, so movement can then happen on the server. here's

how the input script does it:

```
func _process(_delta):
	movement_direction()
```

```
func movement_direction():
	# unrotated direction from input
	var direction_ur = Input.get_vector("move_left","move_right","move_forward","move_back")
	# set strafing
	var strafing_left = false
	var strafing_right = false
	if direction_ur.x < 0:
		strafing_left = true
	if direction_ur.x > 0:
		strafing_right = true
	rpc_id(1,"set_strafing",strafing_left,strafing_right)
	# set backpedaling
	var backwards = false
	if direction_ur.y > 0:
		backwards = true
	rpc_id(1,"set_backpedaling",backwards)
	# rotate input according to camera orientation
	direction = Vector2(cos(-$"../camera_rotation".rotation.y)*direction_ur.x -\
			sin(-$"../camera_rotation".rotation.y)*direction_ur.y, \
			sin(-$"../camera_rotation".rotation.y)*direction_ur.x +\
			cos(-$"../camera_rotation".rotation.y)*direction_ur.y)
	if Input.is_action_just_pressed("jump"):
		jump.rpc()
```

So the direction comes straight from the input and gets synced via the synchronizer, while states such as strafing and backpedaling use RPCs to sync animations.

Spells

while there is targeting and interaction functionality in player_input.gd, we don't care about these right now. let's find where spells happen. In actionbar_slot.gd there is this:

```
func _on_pressed() -> void:
	# the rpc call to fire the spell is sent from player_input, as the server does not load
	# the actionbar ui elements of players, and using player_input is somewhat intuitive
```

```
	References.player_reference.get_node("player_input").\
		request_enter_spell_container(slot_spell_id)
```

and back in the player_input.gd :

```
func request_enter_spell_container(spell_id: String):
	rpc_id(1,"enter_spell_container",spell_id)



@rpc("authority","call_local")
func enter_spell_container(spell_id: String):
	$"../spell_container".spell_entrypoint(spell_id)
```

so the first call is a local function that allows the input script, with player authority, to send a request to the server to enter a spellcontainer. the second one calls this in spell_container.gd:

```
func spell_entrypoint(spell_id: String) -> int:
	# determine whether spell is present in container
	var node_name = "spell_"+spell_id
	var spell_node = get_node_or_null(node_name)
	if spell_node == null:
		print("spell %s not known"%spell_id)
		return 1  # spell not known
	result = spell_node.trigger()
	return result
```

the second to last line, `result = spell_node.trigger()` actually triggers the spell. Spells when triggered should send combat events via the combat script.

Revision #3
Created 2024-10-05 21:56:32 UTC by Admin
Updated 2024-10-06 09:19:07 UTC by Admin