# Godot Character Controller - Complete Technical Documentation

## Table of Contents

---

## Project Overview

### Mission Statement

This character controller is designed as a robust, production-ready foundation for 3D games in Godot. Built with composition architecture, it provides immediate professional-quality character control while maintaining the flexibility to expand into complex gameplay systems.

### Design Philosophy

**Composition Over Inheritance:** Each system operates independently, communicating through a central InputBus. This eliminates tight coupling and enables easy modification or replacement of individual components.

**Performance-First:** Every system is optimized for real-time gameplay, with caching, dirty flagging, and minimal allocation patterns throughout.

**Artist-Friendly:** All parameters are exposed via @export variables with sensible defaults, enabling rapid iteration without code changes.

### Core Architecture

```
CharacterController (Orchestrator)
├── InputBus (Communication Hub)
├── MovementSystem (Physics & Movement)
├── CameraSystem (Following & Look)
└── Configuration Resources
    ├── InputConfig (Sensitivity, Controls)
    └── MovementConfig (Speed, Physics)
```

## Camera System

### Overview

The CameraSystem provides professional third-person camera behavior with smooth following, mouse look control, and collision avoidance. Built around Godot's SpringArm3D for robust collision detection.

### Core Components

#### SpringArm3D Integration

**Purpose:** Automatic collision avoidance and camera positioning **Key Settings:**

- `collision_mask = 1` - Detects world geometry
- `margin = 0.1` - Distance buffer from walls
- `spring_length = 5.0` - Default camera distance

**Why SpringArm3D:** Unity developers often write custom raycasting for camera collision. Godot's SpringArm3D handles this automatically, providing smooth collision response without manual ray calculations.

#### Camera Following System

**Horizontal Following:**

```gdscript
@export var horizontal_follow_time: float = 0.2
@export var horizontal_max_speed: float = 100.0
```

- Uses lerp-based smoothing with configurable lag
- Prevents jarring camera snaps during rapid movement
- Max speed prevents infinite lag during teleportation

**Vertical Following with Dead Zone:**

```gdscript
@export var vertical_dead_zone: float = 1.5
@export var vertical_follow_time: float = 0.8
```

- Dead zone prevents camera bobbing during normal movement
- Slower vertical tracking for natural feel
- Activates only when character moves significantly up/down

#### Mouse Look Implementation

#### Rotation Accumulation:

- Horizontal rotation: Unlimited 360-degree rotation
- Vertical rotation: Clamped to prevent camera flipping
- Separate smoothing for each axis

#### Performance Optimization:

- Mouse sensitivity cached in InputConfig
- Rotation calculations use direct transform modification
- No unnecessary vector allocations per frame

### Camera Design Rationale

#### Why Separate Horizontal/Vertical Following?

Human vision naturally separates horizontal tracking (following action) from vertical tracking (maintaining horizon). This mirrors how cinematographers operate cameras.

#### Why Dead Zones?

Without vertical dead zones, cameras "bob" constantly as characters step over small obstacles. Dead zones provide stability while maintaining responsiveness to significant elevation changes.

#### Why SpringArm3D Over Manual Raycasting?

1. **Automatic collision handling** - No need to manage ray directions
2. **Built-in smoothing** - Natural wall collision response
3. **Performance optimized** - Engine-level collision detection
4. **Margin handling** - Prevents z-fighting with walls

### Key Parameters Explained

| Parameter | Purpose | Recommended Range |
|---|---|---|
| `horizontal_follow_time` | Camera lag for horizontal movement | 0.1-0.5s |
| `vertical_dead_zone` | Vertical movement threshold | 1.0-2.5 units |
| `camera_height_offset` | Height above character center | 1.0-2.0 units |
| `spring_length` | Camera distance from character | 3.0-8.0 units |

---

## Control System

## Overview

The InputBus centralizes all input processing, creating a clean data flow from hardware input to gameplay systems. This eliminates the common problem of systems directly polling input, which creates coupling and testing difficulties.

## InputBus Architecture

### Data Flow Pattern

```
Hardware Input → InputBus → InputState → Systems
```

**Single Direction:** Input flows in one direction only, preventing circular dependencies and making behavior predictable.

**InputState Structure:**

```gdscript
class_name InputState
var movement_input: Vector2    # WASD/analog stick
var look_input: Vector2        # Mouse movement
var jump_pressed: bool         # Jump button state
var jump_just_pressed: bool    # Jump button this frame
```

### Why InputBus Over Direct Input?

1. **Testability:** Systems can be tested with mock InputState objects
2. **Recording/Replay:** InputState can be serialized for replays
3. **Input Remapping:** All mapping logic centralized in InputBus
4. **Multiple Input Sources:** Easy to support keyboard + gamepad simultaneously

## Input Processing Details

### Mouse Capture Management

```gdscript
func capture_mouse():
	Input.mouse_mode = Input.MOUSE_MODE_CAPTURED

func release_mouse():
	Input.mouse_mode = Input.MOUSE_MODE_VISIBLE
```

**Automatic capture** when game starts, **automatic release** when game loses focus.

**Input Buffering**

**Jump Buffer Timer:** Allows jump input slightly before landing **Coyote Time:** Allows jump slightly after leaving ground Both prevent frustrating input timing issues common in platformers.

**Configuration System**

**InputConfig Resource:**

- Mouse sensitivity (horizontal/vertical)
- Gamepad deadzone settings
- Input action mappings
- Accessibility options

**Why Resource-Based Config:** Godot Resources can be saved/loaded automatically, enabling persistent settings and easy designer tweaking.

### Control Design Rationale

#### Why Separate Movement and Look Input?

Movement (WASD) and camera control (mouse) operate on different timescales and have different precision requirements. Separating them allows independent tuning and processing.

#### Why Buffer Systems?

Professional games feel responsive partly due to input buffering. Players expect slight timing forgiveness, especially during fast-paced gameplay.

---

## Character Movement

### Overview

The MovementSystem handles all character physics, implementing proper 3D movement with gravity, ground detection, jumping, and camera-relative directional input.

### Physics Implementation

#### CharacterBody3D Configuration

gdscript

```gdscript
character_body.slide_on_ceiling = false      # Prevent ceiling sliding
character_body.floor_stop_on_slope = true    # Stop sliding down slopes
character_body.floor_block_on_wall = true    # Prevent wall climbing
character_body.floor_snap_length = 0.1       # Ground detection distance
character_body.floor_max_angle = deg_to_rad(45.0)  # Maximum walkable slope
```

**Why These Settings:**

- `slide_on_ceiling = false`: Prevents characters from "surfing" on ceilings
- `floor_stop_on_slope = true`: Characters don't slide down gentle slopes
- `floor_snap_length = 0.1`: Maintains ground contact over small gaps
- `45-degree max slope`: Industry standard for walkable terrain

### Movement Calculation

**Camera-Relative Movement:**

```gdscript
func calculate_ground_movement(input_state: InputState, delta: float):
  var input_vector = input_state.movement_input
  var movement_direction = (cached_camera_forward * input_vector.y +
                            cached_camera_right * input_vector.x).normalized()
```

**Why Camera-Relative:** Players expect "forward" to mean "toward camera direction," not world-space forward. This prevents disorienting movement when camera rotates.

### Gravity System

**Custom Gravity Implementation:**

```gdscript
var gravity_scale: float = 2.5  # Multiplier for responsiveness
current_velocity.y += gravity * gravity_scale * delta
```

**Why Custom Gravity:** Default physics gravity often feels floaty in games. Custom implementation allows fine-tuning jump arc and fall speed for better game feel.

### Jump Mechanics

**Multi-Frame Jump Processing:**

1. **Jump Request:** Player presses jump button
2. **Jump Buffer:** System remembers request for several frames
3. **Ground Check:** Verifies character can jump
4. **Coyote Time:** Allows jump shortly after leaving ground
5. **Jump Execution:** Applies upward velocity

**Technical Implementation:**

gdscript

```gdscript
# Jump buffer - remember jump request
if input_state.jump_just_pressed:
  jump_buffer_timer = movement_config.jump_buffer_time

# Coyote time - remember recent ground contact
if was_on_ground and not is_on_ground:
  coyote_timer = movement_config.coyote_time

# Execute jump if conditions met
if jump_buffer_timer > 0.0 and (is_on_ground or coyote_timer > 0.0):
  execute_jump()
```

## Performance Optimizations

### Camera Direction Caching

**Problem:** Calculating camera forward/right vectors every frame is expensive **Solution:** Cache directions and only recalculate when camera rotates

gdscript

```gdscript
func update_camera_directions():
  if not camera_directions_dirty:
    return

  # Expensive calculations only when needed
  cached_camera_forward = -camera_transform.basis.z
  cached_camera_right = camera_transform.basis.x
  camera_directions_dirty = false
```

### Dirty Flagging Pattern

Systems mark data as "dirty" when it changes, preventing unnecessary recalculation. This is crucial for maintaining 60+ FPS with complex character controllers.

## Movement Design Rationale

### Why Separate Ground and Air Movement?

Ground and air movement have fundamentally different physics requirements:

- **Ground:** Immediate response, high friction, precise control
- **Air:** Momentum-based, limited control, gravity-affected

### Why Coyote Time and Jump Buffering?

These systems prevent common platformer frustrations:

- **Coyote Time:** "I was on the ground when I pressed jump!"
- **Jump Buffer:** "I pressed jump right before landing!"

Both create more forgiving gameplay without affecting skill ceiling.

---

## Architecture Deep Dive

### Composition Pattern Implementation

#### Core Principle

Each system operates independently, receiving data through InputBus and producing effects without directly coupling to other systems.

**Benefits:**

1. **Modularity:** Systems can be added/removed without affecting others
2. **Testability:** Each system can be unit tested in isolation
3. **Maintainability:** Bug fixes rarely affect multiple systems
4. **Expandability:** New features integrate without refactoring existing code

#### Communication Flow

```
InputBus → InputState → MovementSystem → CharacterBody3D (Physics)
       ↓
InputState → CameraSystem → SpringArm3D → Camera3D (Visuals)
```

**No Cross-Communication:** MovementSystem never directly calls CameraSystem and vice versa. All coordination happens through shared InputState.

### Resource-Based Configuration

#### Why Resources Over Scripts?

Godot Resources provide:

- **Serialization:** Automatic save/load functionality
- **Inspector Integration:** Native editor support
- **Type Safety:** Compile-time checking of configuration
- **Inheritance:** Base configurations with specialized variants

#### Configuration Structure

gdscript

```gdscript
# InputConfig.gd
extends Resource
class_name InputConfig

@export var mouse_sensitivity_horizontal: float = 2.0
@export var mouse_sensitivity_vertical: float = 2.0
@export var mouse_invert_y: bool = false
```

**Designer Benefits:** Artists can create configuration variants without touching code, enabling rapid iteration and A/B testing.

### Error Handling Strategy

#### Defensive Programming

Every system checks for null references and missing dependencies:

gdscript

```gdscript
func process_movement(input_state: InputState, delta: float):
  if not character_body or not movement_config:
    return  # Graceful degradation
```

#### Graceful Degradation

Systems continue operating even with missing components, preventing crashes during development iteration.

---

## Implementation Details

### Scene Structure

```
CharacterController (Node3D)
├── CameraSystem (Node3D)
│   └── SpringArm3D
│       └── Camera3D
├── CharacterBody3D
│   ├── CollisionShape3D (Capsule)
│   └── MeshInstance3D (Character Model)
├── InputBus (Node)
└── MovementSystem (Node)
```

### Initialization Sequence

1. **CharacterController._ready()** - Main initialization
2. **Configure CharacterBody3D** - Physics settings
3. **Initialize InputBus** - Input capture and configuration
4. **Setup MovementSystem** - Character reference and camera transform
5. **Configure CameraSystem** - SpringArm and target assignment

### Configuration Resources

#### InputConfig_A.tres:

```gdscript
[gd_resource type="InputConfig"]
mouse_sensitivity_horizontal = 2.0
mouse_sensitivity_vertical = 2.0
mouse_invert_y = false
```

#### MovementConfig_A.tres:

```gdscript
[gd_resource type="MovementConfig"]
base_speed = 5.0
sprint_multiplier = 1.8
jump_height = 4.5
gravity_scale = 2.5
```

### Integration Points

#### Adding New Systems

1. Create system script extending Node
2. Add @export reference in CharacterController
3. Initialize in `initialize_character_controller()`
4. Process in `_process()` or `_physics_process()`

#### Extending Input

1. Add new fields to InputState
2. Update InputBus to populate fields
3. Systems automatically receive new input data

---

## Performance Considerations

### Frame Rate Targets

- **60 FPS minimum** on mid-range hardware
- **120 FPS capable** on high-end systems
- **30 FPS fallback** for low-end devices

### Memory Management

- **No per-frame allocations** in movement processing
- **Object pooling** for temporary calculations
- **Resource caching** for expensive computations

### Optimization Techniques

#### Dirty Flagging

```gdscript
var camera_directions_dirty: bool = true

func mark_camera_directions_dirty():
  camera_directions_dirty = true

func update_camera_directions():
  if not camera_directions_dirty:
    return
  # Expensive calculations only when needed
```

#### Vector Caching

```gdscript
# Cache expensive calculations
var cached_camera_forward: Vector3 = Vector3.FORWARD
var cached_camera_right: Vector3 = Vector3.RIGHT
```

#### Early Returns

```gdscript
func process_movement(input_state: InputState, delta: float):
  if not character_body:
    return  # Skip processing if no character
```

### Performance Monitoring

Key metrics to track:

- **Input lag:** Target 0-1 frames maximum
- **Movement calculation time:** <0.3ms per frame
- **Camera update time:** <0.2ms per frame
- **Total character processing:** <0.5ms per frame

---

## Future Expansion

### Immediate Extensions (2-3 days each)

#### Animation Integration

**Required Components:**

- AnimationTree with StateMachine
- Animation states: Idle, Walk, Run, Jump, Fall
- Transition conditions based on movement state

**Integration Points:**

- MovementSystem exposes movement speed and state
- AnimationSystem reads state and controls AnimationTree
- No changes to existing architecture

#### Point-and-Click Navigation

**Required Components:**

- NavigationInputHandler (extends InputBus)
- NavMesh scene setup
- Click-to-move destination system

**Integration Points:**

- InputBus switches between keyboard and navigation handlers
- MovementSystem receives target positions instead of input vectors
- Camera system unchanged

### Advanced Extensions (1+ weeks each)

#### Combat System

**Architecture:**

- CombatSystem (new component)
- Action state management
- Hit detection and response
- Animation integration

#### Multiplayer Support

**Architecture:**

- NetworkSync components
- Client prediction
- Server reconciliation
- Input lag compensation

#### AI Character Support

**Architecture:**

- AIInputHandler (replaces InputBus)
- Behavior trees
- Pathfinding integration
- Decision-making systems

## Extension Guidelines

### Maintaining Architecture

1. **Follow composition pattern** - New systems as separate components
2. **Use InputBus pattern** - Central communication hub
3. **Avoid direct coupling** - Systems communicate through shared data
4. **Preserve performance** - No per-frame allocations

### Adding New Systems

1. Create new component script
2. Add to CharacterController with @export
3. Initialize in setup phase
4. Process in appropriate frame callback
5. Document configuration parameters

### Configuration Management

1. Create Resource classes for complex settings
2. Export parameters for designer access
3. Provide sensible defaults
4. Document parameter ranges and effects

---

## Conclusion

This character controller represents a professional-grade foundation for 3D games in Godot. Its composition architecture, performance optimizations, and extensive configurability make it suitable for both rapid prototyping and production development.

The separation of concerns between input, movement, and camera systems provides the flexibility to expand into complex gameplay mechanics while maintaining the responsiveness and polish that players expect from modern games.

Whether you're building a simple platformer or a complex action-RPG, this controller provides the solid foundation necessary to focus on your game's unique features rather than fundamental character control mechanics.