Introduction to 3D Game Programming with Godot

A comprehensive guide to creating 3D games with the Godot Engine

Table of Contents

Table of Contents (cont.)

1. Introduction to Godot Engine

What is Godot?

- Free and open-source game engine
- Supports both 2D and 3D game development
- Unique node-based architecture
- Uses GDScript (Python-like language)
- Completely free with no royalties or fees

Key Features for 3D Development

- **Fully integrated 3D engine**: No need for external tools
- **PBR rendering**: Modern physically-based rendering pipeline
- **Visual shader editor**: Create complex materials without coding
- **Built-in physics**: Powered by Bullet Physics
- **Cross-platform export**: Deploy to multiple platforms

Godot 4.x vs. Godot 3.x

Godot 4.x improvements:

- Vulkan rendering API support
- Improved lighting and shadows
- Enhanced physics
- Better performance for complex 3D scenes
- More advanced material system

2. GDScript Fundamentals

Basic Syntax

```
# Variables and data types
var integer_value = 42
var float_value = 3.14
var string_value = "Hello, World!"
var boolean_value = true
var array_value = [1, 2, 3]
var vector3_value = Vector3(10, 20, 30)

# Type hints (optional but recommended)
var typed_integer: int = 42
var typed_vector: Vector3 = Vector3(1, 2, 3)

# Constants
const MAX_HEALTH = 100
const GRAVITY = Vector3(0, -9.8, 0)
```

Control Flow

```
# If statements
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
else:
    print("Grade: F")

# Loops
for i in range(5):
    print(i)

var items = ["sword", "shield", "potion"]
for item in items:
    print(item)

var counter = 0
while counter < 5:
    counter += 1
```

Functions

```
# Simple function
func greet():
    print("Hello, World!")

# Function with parameters
func add(a, b):
    return a + b

# Function with type hints
func calculate_damage(base_damage: int, multiplier: float) -> int:
    return int(base_damage * multiplier)
```

Built-in Functions

```
# Called when the node enters the scene tree
func _ready():
    print("Node is ready")

# Called every frame
func _process(delta):
    # delta is the time since the last frame
    position.x += 100 * delta  # Move 100 pixels per second

# Called every physics frame
func _physics_process(delta):
    # Use for physics calculations
    apply_gravity(delta)
```

Classes and Object-Oriented Programming

```
# Player.gd
extends CharacterBody3D

# Properties
var health = 100
var speed = 5.0

# Methods
func take_damage(amount):
    health -= amount
    if health <= 0:
        die()

func die():
    queue_free()  # Remove node from scene
```

Signals (Event System)

```
# Define a signal
signal health_changed(new_health)

# Emit a signal
func take_damage(amount):
    health -= amount
    emit_signal("health_changed", health)

# Connect in code
func _ready():
    # Connect to self
    health_changed.connect(_on_health_changed)

# Signal handler
func _on_health_changed(new_health):
    update_health_bar(new_health)
```

Best Practices

1. **Use Type Hints**: Improve code readability and catch errors early
2. **Follow Naming Conventions**:
    - `snake_case` for variables, functions, signals
    - `PascalCase` for classes and nodes
    - `UPPER_SNAKE_CASE` for constants
3. **Organize Code with Classes**: Break down complex functionality
4. **Use Signals for Loose Coupling**: Avoid direct references
5. **Comment Your Code**: Especially for complex algorithms
6. **Avoid Deep Node Hierarchies**: Impact performance
7. **Use Autoloads Sparingly**: Can lead to spaghetti code

3. Mathematical Fundamentals for Games

Coordinate Systems

- **Right-handed coordinate system**
- **X-axis**: Positive to the right
- **Y-axis**: Positive upward
- **Z-axis**: Positive toward the viewer

```
# Creating a position in 3D space
var position = Vector3(x, y, z)

# Origin point
var origin = Vector3.ZERO  # Same as Vector3(0, 0, 0)
```

Vectors

```
# Creating vectors
var vec3 = Vector3(10, 20, 30)   # 3D vector

# Predefined vectors
var up = Vector3.UP              # (0, 1, 0)
var right = Vector3.RIGHT        # (1, 0, 0)
var forward = Vector3.FORWARD    # (0, 0, -1)

# Vector operations
var vec_a = Vector3(1, 2, 3)
var vec_b = Vector3(4, 5, 6)
var sum = vec_a + vec_b          # (5, 7, 9)
var doubled = vec_a * 2          # (2, 4, 6)
var dot = vec_a.dot(vec_b)       # 32
var cross = vec_a.cross(vec_b)   # (-3, 6, -3)
```

Matrices and Transformations

```
# Creating a basis (3×3 matrix)
var basis = Basis()  # Identity basis
var rotation_basis = Basis.from_euler(Vector3(PI/4, 0, PI/2))

# Transform3D (4×4 Matrix)
var transform = Transform3D()  # Identity transform
var translated = Transform3D(Basis(), Vector3(10, 5, 0))

# Transforming a point
var point = Vector3(1, 2, 3)
var transformed_point = transform * point
```

Quaternions

```
# Creating a quaternion from Euler angles
var euler_angles = Vector3(PI/4, PI/2, 0)
var quat = Quaternion.from_euler(euler_angles)

# Creating a quaternion from axis-angle
var axis = Vector3(0, 1, 0)  # Y-axis
var angle = PI/2             # 90 degrees
var quat_from_axis = Quaternion(axis, angle)

# Rotating a vector with a quaternion
var rotated_vec = quat * vec3
```

Interpolation

```
# Linear interpolation (Lerp)
var a = 0
var b = 10
var t = 0.5  # Interpolation factor (0 to 1)
var result = lerp(a, b, t)  # 5

# Vector linear interpolation
var vec_a = Vector3(0, 0, 0)
var vec_b = Vector3(10, 20, 30)
var interpolated_vec = vec_a.lerp(vec_b, t)  # (5, 10, 15)
```

Practical Applications

- **Smooth Camera Follow**: Interpolate camera position
- **Projectile Motion**: Calculate trajectory with physics
- **Orbit Motion**: Circular movement around a point
- **Collision Detection**: Ray casting and intersection tests
- **Procedural Generation**: Using noise and random functions

4. Setting Up a 3D Project

Installing Godot

1. Download from godotengine.org
2. No installation required - self-contained application
3. Consider adding to PATH for command-line access

Creating a New 3D Project

1. Launch Godot
2. Click "New Project" in the Project Manager
3. Set a project name and path
4. Select "3D" as the renderer
5. Click "Create & Edit"

Project Structure

- `project.godot`: Main project file
- `default_env.tres`: Default environment resource
- `.godot/`: Internal Godot files
- `addons/`: Location for plugins
- Custom directories for organizing assets

Version Control Setup

```
# Godot 4+ specific ignores
.godot/

# Godot-specific ignores
.import/
export.cfg
export_presets.cfg

# Imported translations
*.translation

# Mono-specific ignores
.mono/
data_*/
```

5. Understanding the Godot Interface for 3D

Main Areas of the Interface

- **Scene Panel**: Hierarchical view of scene nodes
- **Inspector**: Properties of the selected node
- **FileSystem**: Project files and assets
- **3D Viewport**: Visual editor for your 3D scene
- **Bottom Panel**: Output, debugger, animation editor

Viewport Navigation

- **Orbit**: Alt + Right Mouse Button (or Middle Mouse Button)
- **Pan**: Alt + Middle Mouse Button
- **Zoom**: Mouse Wheel or Alt + Right Mouse Button + Drag
- **Focus on Object**: F key when object is selected
- **View from Specific Angle**: View menu or numeric keypad (1-7)

Viewport Display Options

- **View As**: Wireframe, solid, textured, etc.
- **Perspective/Orthogonal**: Toggle between views
- **Display Gizmos**: Show/hide helper elements

Gizmos and Manipulators

- **Move**: Red, green, and blue arrows (or W key)
- **Rotate**: Colored rings (or E key)
- **Scale**: Colored squares (or R key)
- **Local/Global Space**: Toggle between spaces (T key)

6. 3D Nodes and Scene Structure

Core 3D Nodes

- **Node3D**: Base class for all 3D objects
- **MeshInstance3D**: Displays a 3D mesh
- **Camera3D**: Defines the player's view
- **DirectionalLight3D**: Sun-like light
- **OmniLight3D**: Point light in all directions
- **SpotLight3D**: Cone-shaped light
- **RigidBody3D**: Physics-enabled body
- **CharacterBody3D**: For character movement
- **StaticBody3D**: Immovable physics body
- **CollisionShape3D**: Defines collision boundaries

Scene Organization Best Practices

- Use empty Node3D nodes as organizational containers
- Name nodes clearly and consistently
- Group related elements (e.g., all lights under a "Lights" node)
- Use scene instancing for reusable elements
- Consider performance implications of deep hierarchies

Scene Instancing

- Create reusable scenes for common elements
- Instance them in your main scene
- Modify instance properties while maintaining link to original
- Use scene inheritance for variations of a base scene

7. 3D Assets and Importing

Supported 3D Formats

- **glTF/GLB**: Recommended format (best compatibility)
- **FBX**: Good for complex models and animations
- **OBJ**: Simple mesh format for static models
- **COLLADA (.dae)**: XML-based format
- **STL**: Often used for 3D printing models

Importing 3D Models

1. Place model files in your project folder
2. Godot automatically imports them when detected
3. Select the imported file to configure import settings
4. Adjust settings in the Import dock

Import Settings

- **Scale**: Adjust model size to match world scale
- **Animation**: Import animations or not
- **Materials**: Create materials from model data
- **Meshes**: Configure mesh import options
- **Compression**: Balance quality and file size

Creating Basic 3D Primitives

- Built-in primitives: Cube, Sphere, Cylinder, etc.
- CSG nodes for boolean operations
- ProceduralMesh for code-generated geometry

8. Basic 3D Movement and Controls

Input Mapping

1. Project > Project Settings > Input Map
2. Add action names (e.g., "move_forward", "jump")
3. Assign keys, controller buttons, or mouse actions

Character Movement with CharacterBody3D

```
extends CharacterBody3D

var speed = 5.0
var jump_strength = 10.0
var gravity = 20.0

func _physics_process(delta):
    # Add gravity
    if not is_on_floor():
        velocity.y -= gravity * delta

    # Handle jump
    if Input.is_action_just_pressed("jump") and is_on_floor():
        velocity.y = jump_strength

    # Get movement input direction
    var input_dir = Input.get_vector("move_left", "move_right",
                                    "move_forward", "move_back")
    var direction = (transform.basis * Vector3(input_dir.x, 0, input_dir.y)).normalized()

    # Apply movement
    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()
```

First-Person Camera Control

```
extends Node3D

@onready var camera = $Camera3D

var mouse_sensitivity = 0.002
var camera_x_rotation = 0.0

func _ready():
    Input.set_mouse_mode(Input.MOUSE_MODE_CAPTURED)

func _input(event):
    if event is InputEventMouseMotion:
        # Rotate player around Y axis
        rotate_y(-event.relative.x * mouse_sensitivity)

        # Rotate camera around X axis
        camera_x_rotation = clamp(
            camera_x_rotation + event.relative.y * mouse_sensitivity,
            -PI/2,  # Look straight down
            PI/2    # Look straight up
        )
        camera.rotation.x = camera_x_rotation
```

9. Camera Systems in 3D

Camera Types

- **Fixed Camera**: Static position and rotation
- **First-Person Camera**: Attached to character's head
- **Third-Person Camera**: Follows character from behind
- **Top-Down Camera**: Views scene from above
- **Cinematic Camera**: Scripted movements for cutscenes

Third-Person Follow Camera

```
extends Camera3D

@export var target_path: NodePath
@export var distance: float = 5.0
@export var height: float = 2.0
@export var smoothness: float = 10.0

var target: Node3D

func _ready():
    if target_path:
        target = get_node(target_path)

func _physics_process(delta):
    if !target:
        return

    # Calculate desired position
    var target_pos = target.global_transform.origin
    var desired_pos = target_pos + Vector3(0, height, 0) -
                      target.global_transform.basis.z * distance

    # Smoothly interpolate position
    global_transform.origin = global_transform.origin.lerp(
        desired_pos, smoothness * delta)

    # Look at target
    look_at(target_pos + Vector3(0, height/2, 0), Vector3.UP)
```

Camera Collision

```
# Check for collisions
var space_state = get_world_3d().direct_space_state
var query = PhysicsRayQueryParameters3D.create(target_pos, desired_pos)
query.exclude = [target]
var result = space_state.intersect_ray(query)

if result:
    # If there's a collision, place camera at hit point
    global_transform.origin = result.position + result.normal * 0.2
else:
    # No collision, use desired position
    global_transform.origin = desired_pos
```

Camera Shake Effect

```
extends Camera3D

var shake_amount = 0
var shake_duration = 0
var shake_intensity = 0
var shake_duration_initial = 0

func _process(delta):
    if shake_duration > 0:
        # Calculate random offset based on intensity
        var offset = Vector3(
            randf_range(-1.0, 1.0) * shake_amount,
            randf_range(-1.0, 1.0) * shake_amount,
            0
        )

        # Apply offset to camera
        h_offset = offset.x
        v_offset = offset.y

        # Reduce duration and intensity over time
        shake_duration -= delta
        # Scale intensity based on remaining duration
        shake_amount = shake_intensity * (shake_duration / shake_duration_initial)
    else:
        # Reset camera position
        h_offset = 0
        v_offset = 0
```

10. Physics in 3D

Physics Bodies

- **StaticBody3D**: Immovable objects (terrain, buildings)
- **RigidBody3D**: Objects affected by physics (props, debris)
- **CharacterBody3D**: Controlled characters with custom movement
- **VehicleBody3D**: Specialized for wheeled vehicles
- **Area3D**: Detect objects entering/exiting a space

Collision Shapes

- **BoxShape3D**: Rectangular prism
- **SphereShape3D**: Sphere
- **CapsuleShape3D**: Pill shape (good for characters)
- **CylinderShape3D**: Cylinder
- **ConvexPolygonShape3D**: Custom convex shape
- **ConcavePolygonShape3D**: For complex static geometry

Physics Materials

```
# Create a physics material in code
var physics_material = PhysicsMaterial.new()
physics_material.friction = 0.5
physics_material.bounce = 0.2

# Apply to a RigidBody3D
$RigidBody3D.physics_material_override = physics_material
```

Raycasting

```
func _physics_process(delta):
    if Input.is_action_just_pressed("shoot"):
        var space_state = get_world_3d().direct_space_state
        var camera = $Camera3D
        var from = camera.global_position
        var to = from + camera.global_transform.basis.z * -100

        var query = PhysicsRayQueryParameters3D.create(from, to)
        var result = space_state.intersect_ray(query)

        if result:
            print("Hit: ", result.collider.name)
            print("Position: ", result.position)
            # Apply damage, spawn effects, etc.
```

Physics Layers and Masks

1. Project Settings > Layer Names > 3D Physics
2. Define up to 32 collision layers
3. Configure which layers interact with each other

```
# Set object to be on layer 2
$RigidBody3D.collision_layer = 2

# Set object to collide with layers 1 and 3
$RigidBody3D.collision_mask = 1 | 4  # (2^0 | 2^2)
```

11. Lighting and Materials

Light Types

- **DirectionalLight3D**: Sun-like light affecting entire scene
- **OmniLight3D**: Point light radiating in all directions
- **SpotLight3D**: Cone-shaped light
- **ReflectionProbe**: Captures surroundings for reflections

Lighting Properties

- **Color**: Light tint
- **Energy**: Light intensity
- **Range**: Distance light travels
- **Attenuation**: How light fades with distance
- **Shadow**: Enable/disable and configure shadows

Environment Settings

1. WorldEnvironment node
2. Create or select an Environment resource
3. Configure:
    - Background (color, sky, etc.)
    - Ambient light
    - Fog
    - Glow/bloom
    - Adjustments (brightness, contrast, etc.)

PBR Materials

- **Albedo**: Base color
- **Metallic**: How metallic the surface is
- **Roughness**: Surface smoothness/roughness
- **Normal Map**: Surface detail without extra geometry
- **Emission**: Self-illumination
- **Rim**: Edge lighting effect
- **Clearcoat**: Additional glossy layer
- **Anisotropy**: Directional reflections

Creating a Basic PBR Material

```
# Create a new StandardMaterial3D
var material = StandardMaterial3D.new()

# Set properties
material.albedo_color = Color(0.8, 0.2, 0.2)
material.metallic = 0.7
material.roughness = 0.3
material.emission_enabled = true
material.emission = Color(0.8, 0.2, 0.2)
material.emission_energy = 0.5

# Apply to mesh
$MeshInstance3D.material_override = material
```

12. Basic Game Mechanics Implementation

Health System

```
extends CharacterBody3D

signal health_changed(new_health)
signal died

var max_health = 100
var current_health = max_health

func take_damage(amount):
    current_health = max(0, current_health - amount)
    emit_signal("health_changed", current_health)

    if current_health <= 0:
        die()

func heal(amount):
    current_health = min(max_health, current_health + amount)
    emit_signal("health_changed", current_health)

func die():
    emit_signal("died")
    # Handle death (play animation, game over, respawn, etc.)
```

Inventory System

```
class Item:
    var id: String
    var name: String
    var stackable: bool
    var max_stack: int

class InventorySlot:
    var item: Item
    var quantity: int

var inventory = []
var inventory_size = 20

func _ready():
    # Initialize empty inventory
    for i in range(inventory_size):
        inventory.append(InventorySlot.new())

func add_item(item_id, quantity=1):
    var item_data = get_item_data(item_id)
    # Check if stackable and exists in inventory
    # Add to empty slots if needed
```

Interaction System

```
extends Area3D

signal interaction_available(object)
signal interaction_unavailable

var current_interactable = null

func _ready():
    connect("body_entered", _on_body_entered)
    connect("body_exited", _on_body_exited)

func _on_body_entered(body):
    if body.has_method("interact"):
        current_interactable = body
        emit_signal("interaction_available", body)

func _input(event):
    if event.is_action_pressed("interact") and current_interactable:
        current_interactable.interact()
```

Save/Load System

```
const SAVE_PATH = "user://save_game.dat"

func save_game():
    var save_file = FileAccess.open(SAVE_PATH, FileAccess.WRITE)

    # Create a dictionary with all the data to save
    var save_data = {
        "player": {
            "position": {
                "x": $Player.global_position.x,
                "y": $Player.global_position.y,
                "z": $Player.global_position.z
            },
            "health": $Player.current_health,
            "inventory": get_serialized_inventory()
        },
        "game_state": {
            "level": current_level,
            "score": player_score
        }
    }

    # Save as JSON
    save_file.store_line(JSON.stringify(save_data))
```

13. Performance Optimization

Level of Detail (LOD)

- Create multiple mesh versions with different polygon counts
- Switch based on distance from camera
- Use the LOD node in Godot 4.x

Occlusion Culling

- Use OccluderInstance3D nodes to define occluders
- Place strategically in your level
- Helps avoid rendering objects that are not visible

Instancing

```
extends MultiMeshInstance3D

func _ready():
    # Create a multimesh with 100 instances
    multimesh = MultiMesh.new()
    multimesh.transform_format = MultiMesh.TRANSFORM_3D
    multimesh.mesh = preload("res://assets/grass_blade.mesh")
    multimesh.instance_count = 100

    # Position each instance
    for i in range(100):
        var position = Vector3(
            randf_range(-10, 10),
            0,
            randf_range(-10, 10)
        )
        var basis = Basis().rotated(Vector3.UP, randf() * TAU)
        var transform = Transform3D(basis, position)
        multimesh.set_instance_transform(i, transform)
```

Other Optimization Techniques

- **Texture Optimization**: Compression, atlases, mipmaps
- **Shader Complexity**: Simplify calculations, use vertex shaders
- **Physics Optimization**: Simplified collision shapes, physics layers
- **Profiling**: Use built-in profiler, monitor FPS, check memory leaks

14. Resources and Further Learning

Official Resources

- Godot Documentation
- Godot YouTube Channel
- Godot GitHub Repository
- Godot Asset Library

Community Resources

- Godot Subreddit
- Godot Discord
- Godot Forum
- GDQuest - Tutorials and courses
- KidsCanCode - Godot Recipes

Recommended Books

- "Godot Engine Game Development in 24 Hours" by Ariel Manzur and George Marques
- "Godot Engine Game Development Projects" by Chris Bradfield

Asset Resources

- Kenney - Free game assets
- OpenGameArt - Free game art
- Mixamo - Character animations
- Sketchfab - 3D models (free and paid)

Practice Projects

1. **3D Platformer**: Character movement, jumping, collecting items
2. **First-Person Explorer**: Camera controls, interaction, inventory
3. **Vehicle Simulator**: Physics, controls, terrain
4. **Simple FPS**: Weapons, enemies, health system
5. **Puzzle Game**: Interaction mechanics, state management

Thank You!

Questions?