Godot EditorScript und @tool Scripts: Umfassendes Handbuch für KI-Lernzwecke

Einführung und Grundlagen

Godot EditorScript und @tool Scripts sind **mächtige Werkzeuge für Editor-Erweiterungen** und automatisierte Workflows. Godot Engine Godot Engine Diese Funktionen ermöglichen es, Code direkt im Editor auszuführen, benutzerdefinierte Tools zu erstellen und Entwicklungsprozesse zu automatisieren. Godot Engine Zenva Während @tool Scripts kontinuierlich im Editor laufen, dienen EditorScripts der einmaligen Batch-Verarbeitung von Szenen und Assets. godotengine +2

Das Verständnis dieser Systeme ist kritisch für fortgeschrittene Godot-Entwicklung, da sie sowohl **enormes Potenzial als auch erhebliche Risiken** bergen. Die Community berichtet von häufigen Problemen wie Editor-Crashes, Memory-Leaks und ungewollten Szenen-Änderungen, die durch unvorsichtige Verwendung entstehen können. GitHub +3

Offizielle Dokumentation und technische Grundlagen

@tool Annotation: Grundlegende Funktionalität

Die **@tool Annotation** ist eine spezielle Direktive, die Scripts zur Ausführung im Editor freigibt. Sie wird am Anfang eines Scripts platziert und verwandelt normalen GDScript-Code in Editor-ausführbaren Code. godotengine +3

```gdscript
@tool
extends Node2D

# Dieser Code läuft sowohl im Editor als auch im Spiel
func _ready():
  if Engine.is_editor_hint():
    print("Läuft im Editor")
  else:
    print("Läuft im Spiel")
```

Kritische Sicherheitsaspekte der @tool Annotation:

- **Keine Schutzfunktionen**: Der Editor bietet keinen Schutz gegen Missbrauch von @tool Scripts godotengine
- **Permanente Änderungen**: Alle Änderungen im Editor sind dauerhaft und können nicht rückgängig gemacht werden godotengine

- **Eingeschränkter Zugriff**: Code von anderen Nodes kann nicht im Editor ausgeführt werden godotengine
- **Autoload-Limitation**: Auf Autoload-Nodes kann im Editor nicht zugegriffen werden godotengine +2

EditorScript Klasse: Batch-Verarbeitung

EditorScript erweitert die RefCounted-Klasse und bietet eine strukturierte Methode für einmalige Editor-Operationen: godotengine +2

gdscript

```gdscript
@tool
extends EditorScript

func _run():
    # Haupteinstiegspunkt - wird über File > Run ausgeführt
    var scene = get_scene()
    for node in get_all_children(scene):
        if node is OmniLight3D:
            # Verdopple den Bereich aller OmniLight3D Nodes
            var is_instanced_subscene_child = node != scene and node.owner != scene
            if not is_instanced_subscene_child:
                node.omni_range *= 2.0

func get_all_children(in_node, children_acc = []):
    children_acc.push_back(in_node)
    for child in in_node.get_children():
        children_acc = get_all_children(child, children_acc)
    return children_acc
```

Verfügbare EditorScript-Methoden:

- `_run()`: Haupteinstiegspunkt, wird über Ctrl+Shift+X ausgeführt godotengine
- `get_scene()`: Gibt Root-Node der aktuell bearbeiteten Szene zurück godotengine
- `add_root_node(node)`: Macht einen Node zum Root der aktuellen Szene godotengine
- `get_editor_interface()`: Zugriff auf EditorInterface (deprecated in neueren Versionen) godotengine

Tool-Vererbung und Kontext-Kontrolle

Wichtige Regel: Jedes GDScript, das ein @tool-Script erweitert, muss ebenfalls @tool sein. Die Vererbung erfolgt nicht automatisch. godotengine

gdscript

```
# Base Tool Script
@tool
extends Node
class_name BaseToolScript

# Erweiterndes Script muss ebenfalls @tool sein
@tool
extends BaseToolScript

# Kontext-sensitive Ausführung
@export var editor_only_property: String = "Editor": set = _set_editor_property

func _set_editor_property(value):
    if Engine.is_editor_hint():
        editor_only_property = value
        update_configuration_warnings()
```

Community-Erfahrungen und häufige Probleme

Kritische Instabilitätsprobleme

Die Godot-Community berichtet von **wiederkehrenden Stabilitätsproblemen**, die den praktischen Einsatz von @tool Scripts erheblich beeinträchtigen: GitHub +3

Neustartprobleme (GitHub Issue #66381): @tool Scripts funktionieren oft nicht im Inspector, bis der Editor neu gestartet wird. Dies ist ein **bekannter Bug**, der besonders bei größeren Projekten auftritt. GitHub github

C# EditorScript-Limitations (GitHub Issue #48590): C# Tool-/EditorScripts können nur einmal ausgeführt werden. Nach der ersten Ausführung ist ein vollständiger Editor-Neustart erforderlich. GitHub github

Externe Editor-Inkompatibilität (GitHub Issue #39842): EditorScripts können nicht mit externen Editoren wie VS Code ausgeführt werden, was moderne Entwicklungsworkflows behindert. GitHub +2

Lösungsstrategien der Community

Workflow-Anpassungen:

- **Regelmäßige Editor-Neustarts** bei intensiver Tool Script-Nutzung
- **Szene schließen und neu öffnen** wenn Tool Scripts nicht reagieren GitHub github
- **Soft Reload verwenden** (Ctrl+Shift+R) für kleinere Script-Änderungen godotengine
- **Externen Editor temporär deaktivieren** für EditorScript-Ausführung GitHub github

Alternative Ansätze:

- **EditorPlugin statt Tool Scripts** für komplexe Editor-Funktionalität (Stack Exchange)
- **Separate Scripts für Editor und Game-Logik** verwenden
- **Minimale Tool Script-Komplexität** - komplexe Funktionen in EditorPlugins auslagern

Typische Fallstricke und kritische Fehler

Schwerwiegende Probleme (Editor-Crashes)

queue_free() in Setter-Funktionen: Dies ist der **häufigste Grund für Editor-Crashes**. Die Verwendung von `queue_free()` in Export-Variable-Settern führt zu sofortigen Abstürzen.

```gdscript
# FEHLERHAFT - verursacht sofortigen Crash:
@tool
extends Node2D
@export var amount: int = 0:
  set(value):
    amount = value
    if amount != 0:
      queue_free()  # CRASH!

# KORREKT - Alternative mit Deferred-Call:
@tool
extends Node2D
@export var amount: int = 0:
  set(value):
    amount = value
    if amount != 0:
      call_deferred("queue_free")  # Sicher
```

Infinite Loops in Tool Scripts: Endlosschleifen frieren den Editor ein und können nur durch Force-Quit gestoppt werden. (Godot Tutorials) (GitHub)

```gdscript
```

```gdscript
# FEHLERHAFT - friert Editor ein:
@tool
extends Node
func _process(delta):
    while true:
        pass  # Unendliche Schleife

# KORREKT - mit Escape-Bedingung:
@tool
extends Node
var iteration_count = 0
func _process(delta):
    if not Engine.is_editor_hint():
        return
    while iteration_count < 1000:  # Begrenzte Iteration
        iteration_count += 1
        if iteration_count % 100 == 0:
            await get_tree().process_frame  # Yield für Editor-Responsivität
```

Memory-Leaks und Performance-Probleme

Übermäßige Logging-Ausgaben: Kontinuierliches Logging in Tool Scripts führt zu Speicherlecks im Editor.

```gdscript
# FEHLERHAFT - verursacht Memory-Leak:
@tool
extends Node
func _process(delta):
    for i in range(1000):
        print("Debug info: ", i)  # Speicherleck!

# KORREKT - konditionelles Logging:
@tool
extends Node
var debug_enabled = false
func _process(delta):
    if debug_enabled and Engine.is_editor_hint():
        push_warning("Controlled debug output")
```

Circular References in Klassen: Selbstreferenzierende Klassen-Definitionen verursachen Speicherlecks.

```gdscript
```

```gdscript
# FEHLERHAFT - Speicherleck durch Selbstreferenz:
@tool
extends Node2D
class_name ProblematicClass
func test(arg: ProblematicClass):  # Circular reference
    pass

# KORREKT - Verwendung von WeakRef:
@tool
extends Node2D
class_name SafeClass
func test(arg: WeakRef):  # Sichere Referenz
    if arg.get_ref():
        var obj = arg.get_ref() as SafeClass
```

Ungewollte Szenen-Änderungen

Permanente Modifikationen: Tool Scripts führen irreversible Änderungen an Szenen durch.

gdscript

```gdscript
# PROBLEMATISCH - permanente Änderungen:
@tool
extends Sprite2D
func _process(delta):
    rotation += PI * delta  # Rotiert permanent im Editor

# KORREKT - konditionelle Ausführung:
@tool
extends Sprite2D
func _process(delta):
    if not Engine.is_editor_hint():
        rotation += PI * delta  # Nur im Spiel
```

Praktische Anwendungsbeispiele

Level- und Terrain-Editoren

Prozeduraler Terrain-Generator: Ermöglicht Echtzeit-Terrain-Generierung direkt im Editor. Zenva

gdscript

```gdscript
@tool
extends Node3D
class_name TerrainGenerator

@export var size_width: int = 100
@export var size_depth: int = 100
@export var height_scale: float = 10.0
@export var regenerate: bool = false: set = _regenerate_terrain

func _regenerate_terrain(value):
    if Engine.is_editor_hint() and value:
        generate_terrain()

func generate_terrain():
    # Entferne alte Terrain-Nodes
    for child in get_children():
        child.queue_free()

    # Generiere neues Terrain
    var mesh_instance = MeshInstance3D.new()
    var array_mesh = ArrayMesh.new()

    # Noise-basierte Höhengenerierung
    var noise = FastNoiseLite.new()
    noise.seed = randi()
    noise.frequency = 0.1

    var vertices = PackedVector3Array()
    for x in range(size_width):
        for z in range(size_depth):
            var height = noise.get_noise_2d(x, z) * height_scale
            vertices.append(Vector3(x, height, z))

    # Mesh-Erstellung
    var arrays = []
    arrays.resize(Mesh.ARRAY_MAX)
    arrays[Mesh.ARRAY_VERTEX] = vertices

    array_mesh.add_surface_from_arrays(Mesh.PRIMITIVE_TRIANGLES, arrays)
    mesh_instance.mesh = array_mesh

    add_child(mesh_instance)
    if Engine.is_editor_hint():
        mesh_instance.owner = get_tree().edited_scene_root
```

Asset-Verarbeitungstools

Batch-Texture-Processor: Automatisiert die Verarbeitung von Texturen im Projekt.

gdscript

```gdscript
@tool
extends EditorScript

func _run():
    process_textures_in_directory("res://textures/")

func process_textures_in_directory(path: String):
    var dir = DirAccess.open(path)
    if not dir:
        push_error("Could not open directory: " + path)
        return

    dir.list_dir_begin()
    var file_name = dir.get_next()

    while file_name != "":
        if file_name.ends_with(".png") or file_name.ends_with(".jpg"):
            var full_path = path + "/" + file_name
            process_single_texture(full_path)
        file_name = dir.get_next()

func process_single_texture(texture_path: String):
    var texture = load(texture_path) as Texture2D
    if texture:
        # Beispiel: Erstelle automatisch Normal-Map
        var normal_texture = ImageTexture.new()
        var image = texture.get_image()
        # Normal-Map-Generierung...
        print("Processed texture: ", texture_path)
```

Custom Inspector-Erweiterungen

Random Value Generator: Erweitert den Inspector mit einem Button für Zufallswerte. godotengine

gdscript

```gdscript
@tool
extends EditorInspectorPlugin

func _can_handle(object):
  return object.has_method("generate_random_value")

func _parse_property(object, type, name, hint_type, hint_string, usage_flags, wide):
  if name == "random_seed":
    add_property_editor(name, RandomSeedEditor.new())
    return true
  return false

# Custom Editor Property
@tool
extends EditorProperty
class_name RandomSeedEditor

var property_control = HBoxContainer.new()
var spin_box = SpinBox.new()
var random_button = Button.new()

func _init():
  add_child(property_control)
  property_control.add_child(spin_box)
  property_control.add_child(random_button)

  random_button.text = "Random"
  random_button.pressed.connect(_on_random_pressed)
  spin_box.value_changed.connect(_on_value_changed)

  add_focusable(spin_box)

func _on_random_pressed():
  var new_value = randi() % 1000
  spin_box.value = new_value
  emit_changed(get_edited_property(), new_value)

func _on_value_changed(value):
  emit_changed(get_edited_property(), value)
```

Debugging- und Entwicklungstools

Scene Analysis Tool: Analysiert Szenen auf potenzielle Probleme. GDQuest Godot Engine

```gdscript
```

```
@tool
extends EditorScript

func _run():
    var analysis = analyze_current_scene()
    print_analysis_report(analysis)

func analyze_current_scene():
    var scene = get_scene()
    var analysis = {
        "total_nodes": 0,
        "missing_textures": [],
        "oversized_nodes": [],
        "performance_warnings": []
    }

    _analyze_node_recursive(scene, analysis)
    return analysis

func _analyze_node_recursive(node: Node, analysis: Dictionary):
    analysis.total_nodes += 1

    # Überprüfe auf fehlende Texturen
    if node is Sprite2D and node.texture == null:
        analysis.missing_textures.append(node.name)

    # Überprüfe auf überdimensionierte Nodes
    if node is Control and node.size.x > 4096:
        analysis.oversized_nodes.append(node.name)

    # Überprüfe auf Performance-Probleme
    if node is RigidBody2D and node.get_child_count() > 10:
        analysis.performance_warnings.append(
            "RigidBody2D '%s' has %d children - may cause performance issues" % [node.name, node.get_child_count()]
        )

    for child in node.get_children():
        _analyze_node_recursive(child, analysis)

func print_analysis_report(analysis: Dictionary):
    print("=== SCENE ANALYSIS REPORT ===")
    print("Total nodes: ", analysis.total_nodes)

    if analysis.missing_textures.size() > 0:
        print("Missing textures: ", analysis.missing_textures)
```

```
    if analysis.oversized_nodes.size() > 0:
        print("Oversized nodes: ", analysis.oversized_nodes)

    if analysis.performance_warnings.size() > 0:
        print("Performance warnings:")
        for warning in analysis.performance_warnings:
            print("  - ", warning)
```

Best Practices und Sicherheitsempfehlungen

Defensive Programmierung

Sichere Tool Script-Struktur: Implementierung von Sicherheitsvorkehrungen und Fehlerbehandlung.

gdscript

```
@tool
extends Node2D
class_name SafeToolScript

# Sichere Export-Variable mit Validierung
@export var safe_property: float = 1.0:
  set(value):
    if not _is_valid_property(value):
      push_warning("Invalid property value: " + str(value))
      return

    if safe_property != value:
      var old_value = safe_property
      safe_property = value

      if Engine.is_editor_hint():
        _on_property_changed_editor(old_value, value)
      else:
        _on_property_changed_runtime(old_value, value)

func _is_valid_property(value: float) -> bool:
  return value >= 0.0 and value <= 100.0

func _on_property_changed_editor(old_value: float, new_value: float):
  # Sichere Editor-Änderungen
  update_configuration_warnings()

  # Validierung der Szenen-Integrität
  if not _validate_scene_integrity():
    safe_property = old_value  # Rollback
    push_error("Property change would break scene integrity")

func _on_property_changed_runtime(old_value: float, new_value: float):
  # Runtime-spezifische Behandlung
  pass

func _validate_scene_integrity() -> bool:
  # Überprüfe Szenen-Konsistenz
  var scene_root = get_tree().edited_scene_root if Engine.is_editor_hint() else get_tree().current_scene
  return scene_root != null and is_inside_tree()
```

Performance-Optimierung

Effiziente Editor-Prozesse: Optimierung für Editor-Performance.

```
gdscript
```

```gdscript
@tool
extends Node
class_name PerformantToolScript

# Throttling für Editor-Updates
var last_update_time: float = 0.0
var update_interval: float = 0.1  # 10 Updates pro Sekunde

func _process(delta):
    if not Engine.is_editor_hint():
        return

    var current_time = Time.get_time_dict_from_system()
    var time_stamp = current_time.hour * 3600 + current_time.minute * 60 + current_time.second

    if time_stamp - last_update_time >= update_interval:
        last_update_time = time_stamp
        _editor_update()

func _editor_update():
    # Begrenzte, performante Editor-Updates
    var operations_per_frame = 10
    var completed_operations = 0

    while completed_operations < operations_per_frame:
        # Führe kleine Operation aus
        completed_operations += 1

        # Yield bei bedarf für Editor-Responsivität
        if completed_operations % 5 == 0:
            await get_tree().process_frame
```

Undo/Redo-Integration

EditorScript mit Undo-Unterstützung: Implementierung von Undo/Redo-Funktionalität.

```gdscript
```

```
@tool
extends EditorScript

func _run():
    var undo_redo = EditorInterface.get_undo_redo()
    undo_redo.create_action("Batch Node Modification")

    var scene = get_scene()
    if not scene:
        push_error("No scene loaded")
        return

    # Sammle alle Änderungen vor Ausführung
    var modifications = []
    for node in get_all_children(scene):
        if node is Light2D:
            modifications.append({
                "node": node,
                "property": "energy",
                "old_value": node.energy,
                "new_value": node.energy * 2.0
            })

    # Führe Änderungen mit Undo-Unterstützung aus
    for mod in modifications:
        undo_redo.add_do_property(mod.node, mod.property, mod.new_value)
        undo_redo.add_undo_property(mod.node, mod.property, mod.old_value)

    undo_redo.commit_action()
    print("Modified %d Light2D nodes with undo support" % modifications.size())
```

Konfigurationswarnungen

Intelligente Warnungen: Implementierung von kontextsensitiven Warnungen. godotengine

```
gdscript
```

```gdscript
@tool
extends Node2D
class_name ConfigurableToolScript

@export var required_resource: Resource
@export var min_scale: float = 0.1
@export var max_children: int = 5

func _get_configuration_warnings():
  var warnings = []

  # Ressourcen-Validierung
  if required_resource == null:
    warnings.append("Required resource is not assigned")

  # Skalierungs-Validierung
  if scale.x < min_scale or scale.y < min_scale:
    warnings.append("Scale is below minimum threshold (%.2f)" % min_scale)

  # Kinder-Validierung
  if get_child_count() > max_children:
    warnings.append("Too many children (%d). Maximum recommended: %d" % [get_child_count(), max_children])

  # Performance-Warnungen
  if get_child_count() > 0:
    var complex_children = 0
    for child in get_children():
      if child is RigidBody2D or child is CharacterBody2D:
        complex_children += 1

    if complex_children > 3:
      warnings.append("Many physics bodies may impact performance")

  return warnings
```

Zusammenfassung und Empfehlungen

Godot EditorScript und @tool Scripts sind **mächtige aber instabile Werkzeuge**, die sowohl enormes Potenzial als auch erhebliche Risiken bergen. Godot Engine +3 Die Community-Erfahrungen zeigen, dass erfolgreiche Nutzung eine **defensive Programmierung** und **umfassende Sicherheitsvorkehrungen** erfordert. GitHub +3

Kernempfehlungen für KI-Systeme:

1. **Immer Engine.is_editor_hint() verwenden** für Editor-spezifische Logik godotengine Stack Exchange

2. **Niemals queue_free() in Setter-Funktionen** verwenden
3. **Regelmäßige Editor-Neustarts** bei intensiver Tool Script-Entwicklung
4. **EditorPlugin bevorzugen** für komplexe Editor-Funktionalität Stack Exchange
5. **Extensive Validierung** und Fehlerbehandlung implementieren
6. **Backup vor EditorScript-Ausführung** - keine Undo-Funktionalität vorhanden godotengine GitHub

Häufigste Probleme nach Schweregrad:

- **Kritisch**: Editor-Crashes durch queue_free(), Infinite Loops
- **Hoch**: Memory-Leaks, Performance-Probleme
- **Mittel**: Ungewollte Szenen-Änderungen, Debugging-Schwierigkeiten
- **Niedrig**: Autoload-Zugriffsprobleme, Context-Verwirrung GitHub github

Die erfolgreiche Nutzung dieser Systeme erfordert ein tiefes Verständnis der Godot-Editor-Architektur und eine **konservative Herangehensweise** mit umfassender Fehlerbehandlung. GitHub github KI-Systeme sollten bei der Verwendung von @tool Scripts und EditorScript **maximale Vorsicht** walten lassen und stets defensive Programmierungsmuster implementieren.