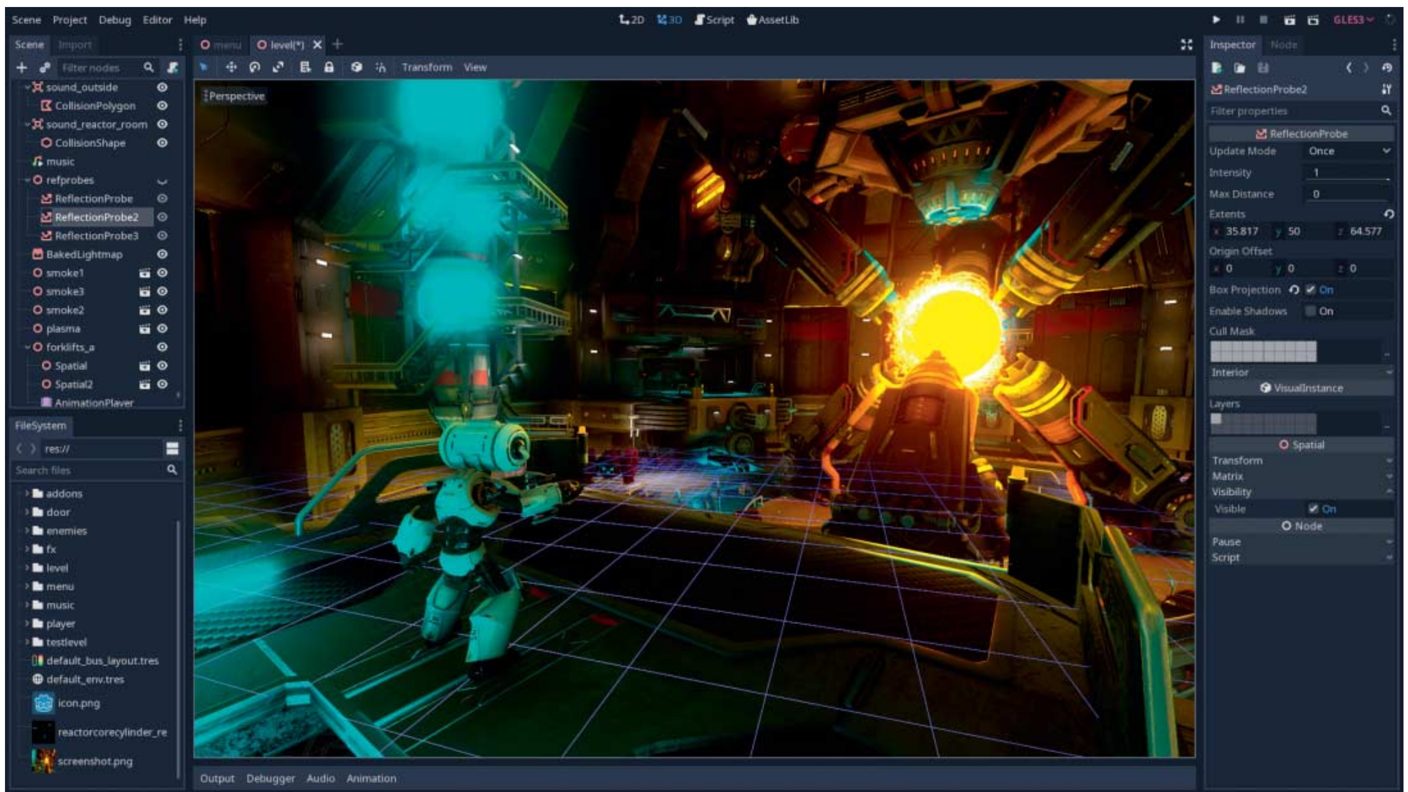


Kein Warten auf Godot

Einstieg in die Spieleprogrammierung mit der Game-Engine Godot

Von Godot fühlen sich zurzeit eher Literaten statt Entwickler angesprochen – aber das dürfte sich schnell ändern: Die Game-Engine hat den Reifegrad eines ausgewachsenen Produktes erreicht, unterstützt alle wesentlichen Plattformen und kostet nichts – auch nicht für kommerzielle Anwendungen. Wir zeigen, wie man damit einen einfachen Asteroids-Klon baut.

Von Dirk Krause

Nur 23 schmale MByte ist das komplette Programmpaket der Godot Game Engine groß. Zum Vergleich: Eine typische Unity-Installation umfasst zwischen 2 und 8 GByte Speicherplatz und dauert gern mal zwei Stündchen oder drei. Unreal ist sogar noch fetter. Trotzdem nimmt es Godot in Sachen Funktionsumfang mit den beiden Standard-Engines auf.

Eine Game-Engine ist ein Softwarepaket, das alle hardwarenahen Funktionen übernimmt, es dem Entwickler also einfacher macht, ein Spiel oder eine andere Software mit 3D-Grafik zu programmieren. Darunter fallen Bild- und -Ton-Ausgabe, aber auch Maus- und Tastatureingabe, Physik und Netzwerkfunktionen.

Die populärsten Engines sind zurzeit Unity und Unreal; beide kommen mit einer freien Einsteiger-Version. Bei Unity muss man aber auf eine monatliche Lizenzgebühr aufstocken, wenn man einen gewissen Mindest-Umsatz überschreitet. Unreals Mutterfirma Epic streicht einen Teil des Umsatzes ein. Ganz anders Godot: Die Software ist zu 100 Prozent Open Source, steht unter MIT-Lizenz und kann damit frei kommerziell benutzt werden. Wer will, kann die Entwickler über die Finanzierungsplattform Patreon unterstützen.

Erste Schritte

Bei der Installation gibt es die erste Überraschung: Der 23-MByte-Download enthält den gesamten Editor und braucht zunächst nichts nachzuladen – im Windows-Archiv beispielsweise steckt nur eine einzige 52-MByte-EXE, installieren muss man nichts. Godot ist eine der wenigen Game-Engines, die Linux voll unterstützen; zusätzlich neben Windows und macOS auch Android und iOS.

Nach Ausführen des Godot-Programms erscheint der Projekt-Manager. Hier geben Sie Ihrem ersten Projekt einen Namen – der dann auch der Name des Ordners wird, in dem Godot alle Daten speichert. Öffnen kann man das Projekt mit „Edit“. Der Bildschirm ist in vier Bereiche geteilt: „Szene/Import“, der die zu erzeugenden oder importierenden Spielelemen-

© Copyright by Heise Medien

Dieses Dokument ist lizenziert für HS Augsburg, uw00411t.
Alle Rechte vorbehalten. © c't - magazin für computertechnik. Download vom 10.07.2019 13:49 von www.wiso-net.de.

te umfasst, „Dateisystem“, der einen Einblick in die Ordnerstruktur gibt, einen großen Viewport in der Mitte mit Umschalttasten für „2D/3D/Skript“ und „AssetLib“ sowie ein „Inspektor“-Feld rechts.

In dem linken oberen Szene-Tab erzeugen Sie zunächst eine „2D-Szene“ als „Root Node“. Godot legt bei der Projektinitialisierung automatisch ein kleines Godot-Bild im Projektordner unten links an, das normalerweise für das Icon des ausgeführten Programms genutzt wird. Für einen ersten Probelauf nehmen Sie eben dieses Icon und ziehen es auf die Bühne in der Mitte. Godot erkennt, dass das ein PNG ist und nimmt korrekterweise an, dass man ein Sprite erzeugen will. Für ein erstes Mini-Programm reicht das schon. Um es auszuführen, drückt man F5 oder klickt den kleinen Play-Button oben rechts. Nach der Aufforderung, die Szene zu speichern, startet Godot das Programm in einem eigenen Fenster.

So weit, so unspektakulär – bis auf eine Kleinigkeit: Wenn man den Editor wieder fokussiert und das Godot-Icon verschiebt, verschiebt sich das Icon auch im laufenden Programm. Godot versucht, den Status des Editors mit dem Programm abzugleichen. Die Änderungen bleiben auch nach Beenden des Programms erhalten.

Godot ist sehr gut internationalisiert, der Editor in vielen Sprachen verfügbar. Nach der Installation startet Godot in Englisch, aber im Menü „Editor/Settings“ kann man die Sprache auf Deutsch umstellen.

Nodes, Szenen und Skripte

Zwei wichtige Grundkonzepte der Godot-Game-Engine sind Nodes und Szenen. Wählen Sie „Node2D“ oben links aus und klicken das „+“-Zeichen an (alternativ geht auch Strg+A). In der dann erscheinenden Auswahlbox finden Sie alle Elemente, die man für die Entwicklung nutzen kann – das sind die sogenannten Nodes; alternativ ziehen Sie sie wie das Bild per Drag-&-Drop in den Editor. Die Menge ist zunächst etwas überwältigend, aber zwei Funktionen erleichtern das Finden erheblich: Zum einen gibt es ein Suchfeld, das beim Eintippen die Auswahl einschränkt, zum anderen merkt sich der Editor alle ausgewählten Nodes im linken Feld, wo Sie häufig benutzte Elemente leichter wiederfinden. Der Editor hält Nodes für 2D-, 3D- und UI-Elemente vor.

Das zweite wichtige Konzept sind die Szenen – also das, was im Beispiel oben gespeichert wurde. Die dabei entstehenden .tscn-Dateien beschreiben die Nodes, die man in der Szene positioniert hat. Das Format ist ein reines Textformat, kann also theoretisch auch händisch erzeugt werden.

Wichtig: Man sollte für jedes Objekt eine eigene Szene anlegen, die dann in der Hauptszene instanziiert und ausgeführt wird. Wer sich ein wenig in Unity auskennt und jetzt an „Prefabs“ denkt, liegt genau richtig. Bei Godot kann diese Szenen beliebig verschachteln, was enormes Potenzial birgt. Die Programmierung der Nodes und Szenen kann auf vier verschiedene Arten geschehen: über GDScript, C#, C++ oder grafische Codeblöcke (Visual Scripting). Im Folgenden gehen wir hauptsächlich auf GDScript ein: Das ist die offiziell empfohlene Sprache, sie ist am engsten mit dem darunterliegenden C++-Code verwoben und daher sehr effizient.

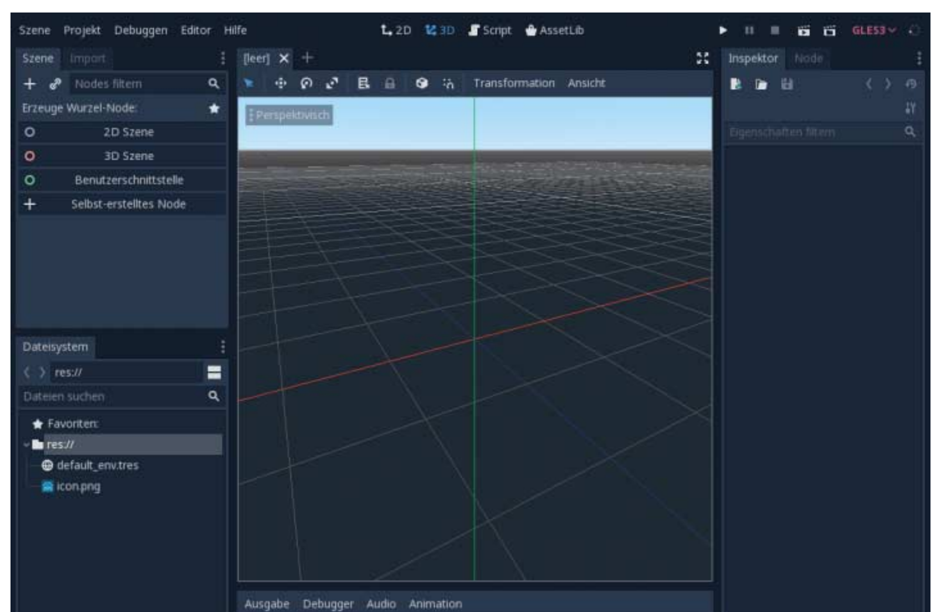
Einige Leser wird der Gedanke abschrecken, eine „neue“ Programmiersprache erlernen zu müssen, um Godot nutzen zu können. Vermutlich ist das auch einer der Gründe, warum Godot noch nicht sonderlich populär ist. Glücklicherweise wurde GDScript so stark an Python angelehnt, dass ein halbwegs erfahrener Python-Entwickler wenig Probleme haben dürfte, sich in den Dialekt einzudenken – in der Tat vergisst man manchmal, dass GDScript nicht Python ist. In der Godot-Dokumentation sind die meisten Code-Beispiele sowohl in GDScript also auch in C# hinterlegt. Wer wiederum auf Effizienz bedacht ist und sowieso schon in der C++-Welt zu Hause ist, für den gibt es C++-Bindings.

Code-Beispiel Klicker-Spiel

Einige der Grundkonzepte von Godot lassen sich gut an einem einfachen Klicker-Spiel erklären. Das Genre beschreibt Spiele, die nichts weiter machen, als durch Anklicken eines Knopfes einen Zähler hochzuzählen – trotz der Einfachheit erzeugen Titel wie „Candy Box“ einen erstaunlichen Sog.

Starten Sie mit einer neuen Szene und erzeugen Sie links im Szene-Tab einen 2D-Szene-Wurzel-Node, der als Basis für die zu bauende Klicker-Komponente dient. Für ein Klickerspiel braucht man nicht viel mehr als einen Knopf (Button) zum Klicken und ein Textfeld (Label) zum Anzeigen. Mit dem „+“-Knopf erzeugen Sie bei aktivem Klicker-Node jeweils einen Button und ein Label. Nach dieser Aktion sollte der Szene-Tab in etwa aussehen wie auf dem Screenshot auf Seite 142.

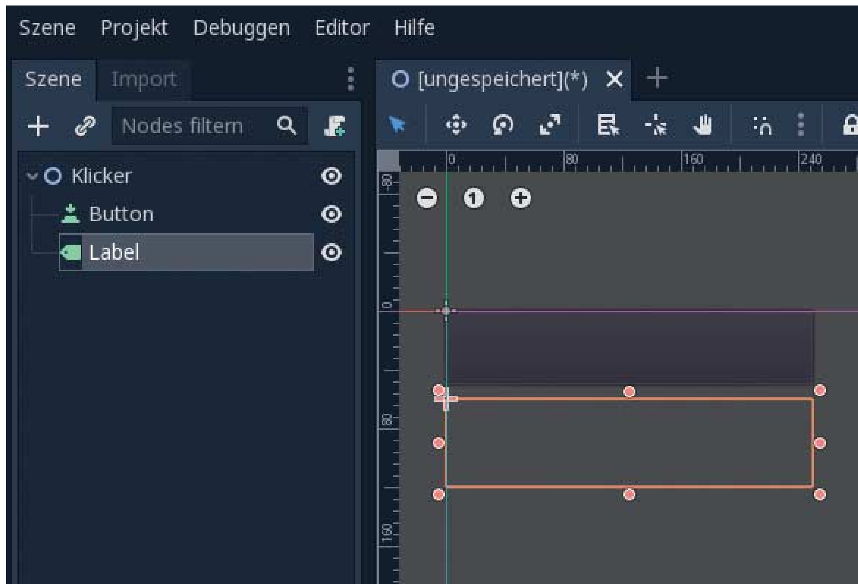
Sollte das Label dabei unter dem Button erschienen sein – kein Problem, einfach mit der Maus nach oben ziehen. Die Elemente erscheinen immer oben links und sind zunächst recht klein. Man kann im Editor Größe und Position auf zwei Arten verändern: entweder mit der Maus durch Anklicken und Verschieben der Elemente beziehungsweise deren Ränder oder – wenn es besonders exakt sein soll – durch die numerische Eingabe im Inspektor auf der rechten Seite. Dort sind sämtliche veränderbaren Eigenschaften



So sieht Godot nach dem Start eines neuen Projekts aus.

© Copyright by Heise Medien

Dieses Dokument ist lizenziert für HS Augsburg, uw00411t.
Alle Rechte vorbehalten. © c't - magazin für computertechnik. Download vom 10.07.2019 13:49 von www.wiso-net.de.



Ein Klicker-Spiel benötigt erst einmal lediglich einen Knopf und ein Textfeld.

des jeweiligen Elements gelistet. Die Größe und Position wird im Feld „Rect“ (wie Rectangle, also Rechteck) unter „Control“ bestimmt. Bewegt man das Element mit der Maus, aktualisieren sich die Werte im Inspektor dementsprechend.

Weder Button noch Label haben Text, auch das kann man im Inspektor ändern; der Button erhält ein „Klick mich!“ und das Label eine initiale „0“, damit man erst mal etwas sieht. Aus ästhetischen Gründen setzen Sie im Inspektor rechts „Align“ und „Valign“ auf „center“, sodass die Null in der Mitte steht.

Zeit, die Szene mit Strg+S als „klicker.tscn“ abzuspeichern und auszuführen. Es gibt mehrere „Run“-Befehle: Strg+B startet das Hauptprogramm (das noch nicht definiert wurde) und Strg+R startet die gerade aktive Szene. Letzteres zeigt das Programm in einem neuen Fenster mit einem jetzt klickbaren Button und dem Label – welches allerdings noch nichts Sinnvolles tut. Um das zu ändern, fügen Sie der „Klicker“-Szene ein Skript hinzu; entweder über das Rechte-Maustasten-Menü oder das Skript-Symbol im Szene-Tab. Im anschließenden Dialog informiert Godot, dass es ein Skript mit den Namen „res://klicker.gd“ anlegen wird. Der eingebaute Text-Editor funktioniert gut, ist aber auch gegen einen externen Editor austauschbar (über „Editor/Editoreinstellungen/Text Editor/External“). Auf der Godot-Website finden sich alle Einstellungen für die üblichen Verdächtigen wie Atom, Sublime Text und Visual Studio Code.

Das Skript „klicker.gd“ ist schon vorbefüllt mit Beispiel-Code, der aber im Moment noch nutzlos ist. Wichtig ist nur, dass schon ein Skript existiert, um ein sogenanntes Signal anlegen zu können – den Code erzeugt der Editor selbstständig. Ein Signal ist eine Nachricht, die szenenunabhängig gesendet und gelesen werden kann. Dafür verknüpfen Sie Button und Skript mit einem „Signal“. Neben dem „Inspektor“-Tab befindet sich der Node-Tab, in dem sich die Signale finden, die der Button schicken kann. Hier sind alle Methoden gelistet, die der jeweilige Node von Haus aus mitbringt. Wenn etwas fehlt, kann man selber Signale hinzufügen. Nach Auswahl von „pressed“ fragt Godot, wohin das Signal geschickt werden soll; zur Wahl stehen alle Nodes der Szene. Wählen Sie den „klicker“-Node aus, an dem das gerade erzeugte Skript hängt. Dort findet sich jetzt die neue Methode `_on_Button_pressed()`, die nun immer dann aufgerufen wird, wenn der Spieler auf den Button drückt:

```
func _on_Button_pressed():
  pass
```

Für unseren Klicker müssen Sie jetzt nur noch oben eine Variable definieren: `var clicks = 0`, deren Wert in der Funktion hochgezählt und ausgegeben wird.

```
extends Node2D
export var clicks = 0
func _on_Button_pressed():
  clicks += 1
  get_node("Label").text = str(klicks)
```

Ein weiteres Schmankerl: Wenn man der Variableninitialisierung ein `export` voranstellt, kann der initiale Wert der Variablen im Editor ohne Codeeingabe bestimmt werden. Um das zu zeigen, öffnen Sie eine neue Szene, initialisieren diese als 2D-Szene und ziehen die klicker.tscn zweimal hinein; im Inspektor kann jetzt der Wert bestimmt werden, und beide Klickermodule zählen jeweils für sich hoch.

Code-Beispiel Asteroids

Um weitere Konzepte von Godot an einem etwas spannenderen Beispiel zu verdeutlichen, setzen Sie im Folgenden einen Teil der Spielmechanik des bekannten Spiel Asteroids um. Starten Sie mit einer leeren 2D-Szene (also einem Node2D), die Sie zu „Player“ umbenennen. Dieser fügen Sie über das Schriftrollen-Icon rechts oben im „Szene“-Fenster vorsorglich ein Skript hinzu, das wiederum „Player.gd“ heißt – der Inhalt kommt später.

An diesen Node hängen Sie das einfachste grafische Objekt, nämlich ein „ColorRect“. Das erspart Ihnen den Import von Texturen. Wer trotzdem welche verwenden möchte, ersetzt dieses ColorRect

```
extends Node2D
var direction = Vector2(0,0)
var screen_size
var d = 8
func _ready():
  screen_size = ↲
   ↳get_viewport_rect().size
func _process(delta):
  if Input.is_action_pressed(
    "ui_left"):
    rotation -= 2 * delta
  if Input.is_action_pressed(
    "ui_right"):
    rotation += 2 * delta
  var movedir = Vector2(1,0)↲
   ↳.rotated(rotation)
  if Input.is_action_pressed("ui_up"):
    direction = direction.↲
     ↳linear_interpolate(movedir,
                        0.05)
  else:
    direction = direction.↲
     ↳linear_interpolate(
       Vector2(0,0), 0.01)
  position += direction * 150 * delta
  position.x = wrapf(position.x, -d,
    screen_size.x + d)
  position.y = wrapf(position.y, -d,
    screen_size.y + d)
```

Dieses player.gd-Skript produziert die Asteroids-Bewegungsmechanik.

© Copyright by Heise Medien

Dieses Dokument ist lizenziert für HS Augsburg, uw00411t.
Alle Rechte vorbehalten. © c't - magazin für computertechnik. Download vom 10.07.2019 13:49 von www.wiso-net.de.

gegen ein texturiertes Sprite. Im Beispiel färben Sie das ColorRect blau, machen es schmaler (x: 40, y: 20) und schubsen es mittig auf die obere rechte Ecke der Szene.

Nun ziehen Sie den Player in die Main-Szene und speichern. Mit Strg+B oder dem Betätigen des Play-Buttons definieren Sie die Szene als Hauptszene. Jetzt sollte das Programm in einem eigenen Fenster ausgeführt werden, aber noch bewegt sich nichts. Das ändert sich mit dem Player.gd-Skript, welches die Asteroids-Bewegungsmechanik mit möglichst wenig Code abbildet (siehe Kasten auf Seite 142).

Dabei passiert folgendes: `_ready()` stellt die Bildschirmgröße fest, damit die Spielobjekte Asteroids-typisch am Bildschirmrand umbrochen werden können, was in den letzten beiden Zeilen passiert.

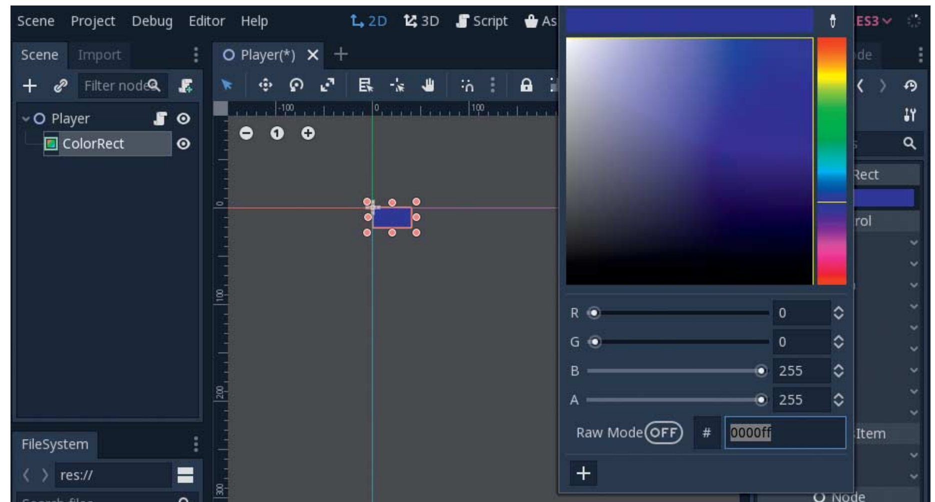
In `_process()` nutzen Sie aus, dass Godot Tasten vorbelegt, namentlich die Pfeiltasten und die „Select“-Taste, was unter Windows und macOS die Leertaste ist. Das `delta` wird genutzt, damit das Spiel unabhängig von der erreichbaren Framerate des jeweiligen Geräts gleich schnell läuft. `position` und `rotation` sind Eigenschaften des Node2D-Objektes und können ohne Präfix genutzt werden, da das Skript direkt am Node2D Objekt hängt (wer es eindeutiger mag, schreibt `self.position`).

Die Links- und Rechts-Pfeiltasten verändern die Rotation direkt. In `movedir()` wird die Stoßrichtung entsprechend der Rotation festgelegt und entweder erhöht oder verringert – wobei etwas schneller beschleunigt als gebremst wird. Anschließend wird die Position gemäß der Stoßrichtung neu gesetzt und das Raumschiff gegebenenfalls an den anderen Bildschirmrand geschubst (beispielsweise ergibt `wrapf(13,0,10)` den Wert 3).

Ein Neustart des Programms sollte jetzt schon das Asteroids-typische Bewegungsmuster ergeben. Allerdings ist der Player noch etwas einsam, weswegen Sie einen weiteren Node2D brauchen, zu „Asteroid“ umbenennen und ebenfalls mit einem Skript ausstatten. Bei uns sind die Asteroiden gelb, etwa 30 × 30 groß und kommen auf die Position –15, –15, damit sie mittig sind.

Auch dieses werfen Sie in die Main-Szene und füllen Asteroid.gd mit folgendem Code auf:

```
extends Node2D
var velocity = Vector2(0,0)
var screen_size
var d = 8
var rotationVel
func _ready():
  randomize()
  screen_size = ↲
   ↳get_viewport_rect().size
  velocity = Vector2(randf()-0.5,
                     randf()-0.5)
  rotation = randf()*3.14
  rotationVel = randf()/10
  position = Vector2(
    rand_range(0,screen_size.x),
    rand_range(0,screen_size.y))
func _process(delta):
  position += velocity * 200 * delta
  position.x = wrapf(position.x, -d,
    screen_size.x + d)
  position.y = wrapf(position.y, -d,
    screen_size.y + d)
  rotation += rotationVel
```



Statt eines texturierten Sprites benutzt unser Demo-Asteroids ein blaues Rechteck („ColorRect“).

Das Skript ähnelt dem Player-Code, nur dass der Asteroid immer der gleichen, initial zufällig gewählten Richtung folgt. Die Funktion `randf()` liefert eine Zufallszahl zwischen 0 und 1, `rand_range(a, b)` zwischen `a` und `b`.

Ein Asteroid macht noch kein Weltraumspiel, aber statt den Asteroiden noch ein paar Mal in die Main-Szene zu werfen (was ginge), erzeugen Sie jetzt die Herde in einer Schleife, und zwar in Main.gd:

```
extends Node2D
func _ready():
  var asteroid = ↲
   ↳preload("res://Asteroid.tscn")
  for i in range(15):
    var a = asteroid.instance()
    add_child(a)
```

Das lädt die Szene, erzeugt 15 weitere Instanzen, ruft deren `_ready()` auf und fügt sie zur Szene hinzu.

Nach einem Neustart ist schon ordentlich Gewimmel auf dem Schirm. Allerdings kollidiert noch nichts. Um das zu erreichen, brauchen Sie einen ‚Area2D‘-Node, den Sie zunächst an den Player in der Datei Player.tscn hängen (nicht an die Instanz in Main). Eine Warnung erinnert daran, dass man für eine funktionierende Kollision einen `CollisionShape2D`-Node braucht, welchen Sie an den ‚Area2D‘ anhängen müssen. In diesem wiederum muss man erst ein „New RectangleShape2D“ auswählen, das dann die Kollisionsform ausmacht. Anschließend legt man das grafische Objekt auf das ColorRect. Wer es genau haben und Zahlen eingeben will, kann unter RectangleShape2D „edit“ auswählen und eben das tun.

Genau das Gleiche erledigen Sie jetzt beim Player; vereinfacht wird das dadurch, dass man beim Hinzufügen der Nodes links die vorher verwendeten sofort ohne Suchen findet.

Um die Kollisionen ans Laufen zu bekommen, bieten sich Signale an. Die möglichen Signale, die Area2D zur Verfügung stellt, findet man in dem Tab neben dem Inspektor rechts. Wählen Sie dazu das Area2D des Asteroiden aus, nicht das des Players. Das oberste Signal ist das gewünschte „area_entered“:

```
func _on_Area2D_area_entered(area):
  pass
```

Im anschließenden Dialog wählen Sie unten rechts „Connect“ aus. Jetzt sind

© Copyright by Heise Medien

Dieses Dokument ist lizenziert für HS Augsburg, uw00411t.
Alle Rechte vorbehalten. © c't - magazin für computertechnik. Download vom 10.07.2019 13:49 von www.wiso-net.de.

zwei Sachen passiert: Godot hat das Area2D mit dem Asteroid-Skript verknüpft und netterweise ganz unten auch schon den zugehörigen Code im Skript angehängt.

Das obige ersetzen Sie durch

```
func _on_Area2D_area_entered(area):
  queue_free()
```

Die Funktion `queue_free()` gibt die Ressourcen des Objektes frei, das heißt, der Asteroid wird gelöscht, sobald das Signal gesendet wird.

Das Gleiche machen Sie beim Player, nur sieht dort das Kollisions-Skript wie folgt aus:

```
func _on_Area2D_area_entered(area):
  get_tree().reload_current_scene()
```

Das ist im Wesentlichen ein schnelles Ende durch ein Neuladen der Szene – quasi ein Restart. Hier käme im Normalfall ein End-Dialog, Score oder Ähnliches hin.

Das Spiel ist jetzt sogar schon spielbar: Man muss den Asteroiden ausweichen, um möglichst lange am Leben zu bleiben. Sie könnten einen Timer und einen Score einbauen und dann hier aufhören. Aber Asteroids ohne Schießen? Niemals.

Ergo brauchen Sie eine weitere Szene mit dem Namen „Bullet“ und dem obligatorischen, neu anzuhängendem Skript „Bullet.gd“, das als Bullet.tscn gespeichert wird. Diesmal wird das ColorRect rot und etwa (–5, –1, 10, 2) groß. Ebenfalls addieren Sie wie gehabt Area2D mit CollisionShape. Es muss eigentlich nichts weiter tun, als mit hoher Geschwindigkeit in eine Richtung zu fliegen – dafür reicht das folgende Skript:

```
extends Node2D
var screen_size
var velocity = Vector2(1,0)
func _ready():
  screen_size = ⤶
   ↳get_viewport_rect().size
func _process(delta):
  position += velocity * 1000 * delta
  position.x = wrapf(position.x, 0,
                   screen_size.x)
  position.y = wrapf(position.y, 0,
                   screen_size.y)
```

Welche Richtung das ist und wo es losgeht, hängt von Rotation und Ort des Players ab. Das Bullet-Objekt wird nicht vom Player verwaltet – wäre das so, würden auch Drehungen des Players das Bullet-Objekt mit rotieren. Stattdessen soll es im globalen Koordinatensystem der Main-Szene geradeaus fliegen. Eine elegante Möglichkeit, das zu realisieren, ist die Erzeugung eines eigenen Signals `shoot()`, dem Objekt, Postion und Rotation übergeben werden, damit die Main-Szene den Torpedo auf den Weg bringt:

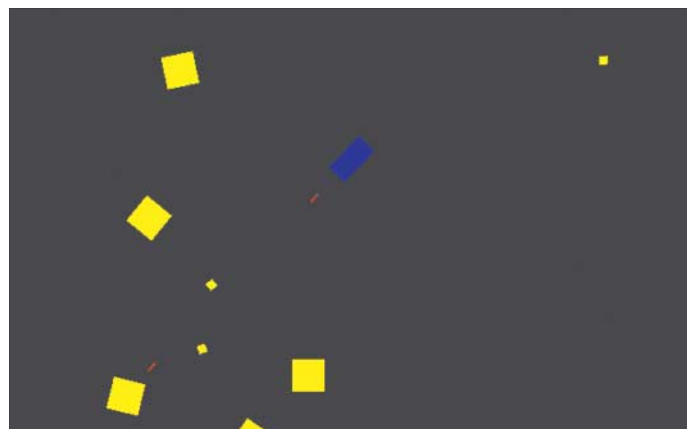
```
signal shoot(bullet, rotation,
             location)
var bullet = ⤶
  ↳preload("res://bullet.tscn")
```

Diese beiden Zeilen kommen in Player.gd; die erste erzeugt das Signal, die zweite lädt die Bullets nach (sic!).

Danach erweitern Sie den „process“-Block um eine weitere Taste (`"ui_select"` meint per Default die Leertaste):

```
if Input.is_action_just_pressed(
    "ui_select"):
  var thisBullet = bullet.instance()
  emit_signal("shoot", thisBullet,
              rotation, position)
```

Dieser Code erzeugt eine Bullet-Instanz und übergibt Rotation und Position des Player-Objekts in das Signal.



Rudimentär, aber spielbar: das fertige Asteroids-Projekt

Gefeuert wird aus oben genannten Gründen in Main.gd:

```
func _on_Player_shoot(bullet,
                      rotation,
                      position):
  add_child(bullet)
  bullet.rotation = rotation
  bullet.position = position
  bullet.velocity = bullet.velocity.⤶
   ↳rotated(rotation)
```

Das mit Bullet.gd verknüpfte „area_entered“-Signal wird aufgerufen, wenn das Bullet einen Asteroiden trifft; überschreiben Sie es wie folgt:

```
func _on_Area2D_area_entered(area):
  area.get_parent().queue_free()
  queue_free()
```

Dies löscht den getroffenen Asteroiden und danach das Bullet-Objekt selbst.

Damit das Ganze nicht buchstäblich zum Rohrkrepierer wird – das Bullet-Objekt wird ja an der gleichen Stelle wie der Player erzeugt und kollidiert ergo auch sofort mit ihm –, setzt man sogenannte Kollisionsmasken, zu finden unter „Collision“ in Area2D.

Ein letztes Problem bleibt noch: Einmal gefeuert, fliegen die Geschosse unendlich lang weiter, bis sie auf etwas treffen. Abhilfe schafft, die Lebenszeit der Bullets zu begrenzen. Dazu fügen Sie der Bullet-Szene einen Timer hinzu und verknüpfen das „timeout“-Signal mit dem Skript. Das überschreiben Sie wie folgt:

```
func _on_Timer_timeout():
  queue_free()
```

Den Timer setzen Sie auf „Autostart“ und „One Shot“. Das heißt: Bei Erzeugung des Objekts startet der Timer und nach einer Sekunde löst er einmalig aus und löscht das Objekt.

Godot beherrscht auch Vererbung – das nutzen Sie für eine letzte Verbesserung, nämlich die Einführung eines zweiten Asteroidentypen. Das Zerstören des ersten Typen („big“) soll nicht nur das Objekt verschwinden lassen, sondern mehrere neue, kleinere Asteroiden erzeugen („small“). Dazu duplizieren Sie die Datei Asteroids.tscn und nennen sie jeweils Asteroids-big.tscn und Asteroids-small.tscn. Beim anschließenden Öffnen benennen Sie die Root-Nodes entsprechend um und lösen das Asteroids.gd-Skript davon ab. Dann erzeugen Sie jeweils ein neues Skript; aber bei der Erzeugung wählen Sie bei „Inherits“ eben das alte Asteroid.gd

© Copyright by Heise Medien

Dieses Dokument ist lizenziert für HS Augsburg, uw00411t.
Alle Rechte vorbehalten. © c't - magazin für computertechnik. Download vom 10.07.2019 13:49 von www.wiso-net.de.

„Wir diskutieren manchmal Monate“

Der Hauptentwickler der Godot Engine, Juan Linietsky, ist nicht nur langjähriger Programmierer, sondern auch technischer Berater und Musiker. Im c’t-Interview erklärt er, warum er eine neue Game-Engine entwickelt hat.

c’t: Warum braucht die Welt eine neue Game Engine?

Juan Linietsky: Als ich 2002 angefangen habe, gab es schlicht keine bezahlbare Game Engine und schon gar keine, die Open Source war. Mit den Jahren wuchs Godot dann immer mehr, und es stellte sich heraus – die Welt braucht eine neue Game Engine! (lacht)

c’t: Sie sagen, dass Godot stark gewachsen ist. Ist es nicht sehr schwierig, ein so großes Team zu koordinieren?

Godot-Entwickler Juan Linietsky

Linietsky: Es ist kompliziert. Godot hat über 1000 Contributers auf GitHub, und das alleine zu managen ist quasi unmöglich. Glücklicherweise ist Rémi Verschelde seit März 2018 an Bord. Er koordiniert den engeren Kreis der Entwickler und macht Dinge wie beispielsweise vernünftiges Tagging der Issues. Er übernimmt viele Community-Aufgaben und kuratiert. Er macht sozusagen die Magie.

c’t: Sie leben und arbeiten in Buenos Aires. Gibt es dort eine große Spieleentwicklerszene?

Linietsky: Statistisch gesehen kommen die meisten Spielentwickler aus Osteuropa; aber tatsächlich kommt einiges von hier: Cocos2D wird in Argentinien entwickelt, der Hauptentwickler von Ogre3D ist ebenfalls Argentinier. Ein Grund dafür dürfte sein, dass Buenos Aires, die reichste Stadt Lateinamerikas, schon vor vielen Jahrzehnten eine Software-Industrie aufgebaut und viel in IT-Studiengänge investiert hat.

c’t: Wie verhindern Sie Bloat, also das Anwachsen der Software-Basis?

Linietsky: Durch Neuschreiben. Nach jedem mit Godot realisierten Projekt haben Ariel Manzur (Co-Autor von Godot) und ich uns den Code angeschaut – und ihn dann neu geschrieben. Das haben wir dann drei-, vier-, vielleicht fünfmal gemacht. Deshalb ist der aktuelle Stand inzwischen wirklich gut. Anders als in großen Firmen nehmen wir uns viel Zeit. Wir diskutieren manchmal Monate, bevor wir uns an die Entwicklung neuer Features machen. Ich denke, deshalb ist Godot auch so klein und effizient.

c’t: Godot ist eine der wenigen Game Engines, die Linux unterstützt. Sie entwickeln sogar auf Linux, richtig?

Linietsky: Genau, ich nutze Ubuntu zur Entwicklung. Viele Entwickler arbeiten mit Linux, und sprangen zu Beginn auf, weil Godot sehr stabil auf Linux lief. Mit der größeren Verbreitung sind die meisten unserer User jetzt aber auf Windows unterwegs.

c’t: Entwickeln Sie selbst noch Spiele?

Linietsky: Leider nein, ich schätze man kann wohl doch nicht alles gleichzeitig machen (lacht). Vielleicht in einigen Jahren wieder, wenn Godot stabiler ist.

aus. Jetzt verbinden Sie wie vorher das „entered“-Signal in die jeweiligen Skripts. Das generische Asteroids.gd benötigt kein Signal mehr, da dieses jetzt nur noch die gemeinsam benötigte Funktionalität abbilden muss. Daher können Sie jetzt die existierende `_on_Area2D_area_entered`-Methode in die Asteroids-small.gd umkopieren:

```
extends "res://Asteroid.gd"
func _on_Area2D_area_entered(area):
  queue_free()
```

Die Big-Variante lädt die kleineren Asteroiden, erzeugt eins bis vier neue und löscht sich dann selbst:

```
extends "res://Asteroid.gd"
var a = preload(
  "res://Asteroid-small.tscn")
func _on_Area2D_area_entered(area):
  for i in range(randi() % 4 + 1):
    var s = a.instance()
    get_parent().add_child(s)
    print(area.get_parent().position)
    s.position = ↵
      ↳area.get_parent().position
  queue_free()
```

Zum Schluss verkleinern Sie noch ColorRect und CollisionShape der Small-Szene. Das Endergebnis sieht dann aus wie auf dem Screenshot auf Seite 144.

Fazit

Ob professioneller Spiele-Entwickler, Hobbyist oder blutiger Anfänger: Die Godot Engine sollte man sich unbedingt anschauen, nicht nur weil sie kostenlos ist. Der Profi, der lang genug mit Unity gearbeitet hat, wird sich wehmütig daran erinnern, wie schlank und schnell Unity einmal war – Godot ist inzwischen das bessere Rapid Prototyping Tool.

Für Anfänger und Hobbyisten ist die Lernkurve der Godot Engine flach genug, um einen guten Einstieg zu bekommen. Die Dokumentation ist recht vollständig und sehr solide, außerdem gibt es viele Beispiele. Wichtig ist, dass man beim Recherchieren von Drittquellen auf die Versionsnummer achtet – die aktuelle Version 3.1 unterscheidet sich erheblich von älteren Versionen. Mehr Praxisinfos über Godot, zum Programm-Export und der Entwicklung von 3D-Projekten finden Sie in einer der nächsten c’t-Ausgaben.

(jkj@ct.de) c't

***Listings:* ct.de/ytc4**

© Copyright by Heise Medien

Dieses Dokument ist lizenziert für HS Augsburg, uw00411t.
Alle Rechte vorbehalten. © c't - magazin für computertechnik. Download vom 10.07.2019 13:49 von www.wiso-net.de.