LICEUL TEORETIC "MIHAIL KOGĂLNICEANU" VASLUI

LUCRARE PENTRU OBŢINEREA ATESTATULUI PROFESIONAL

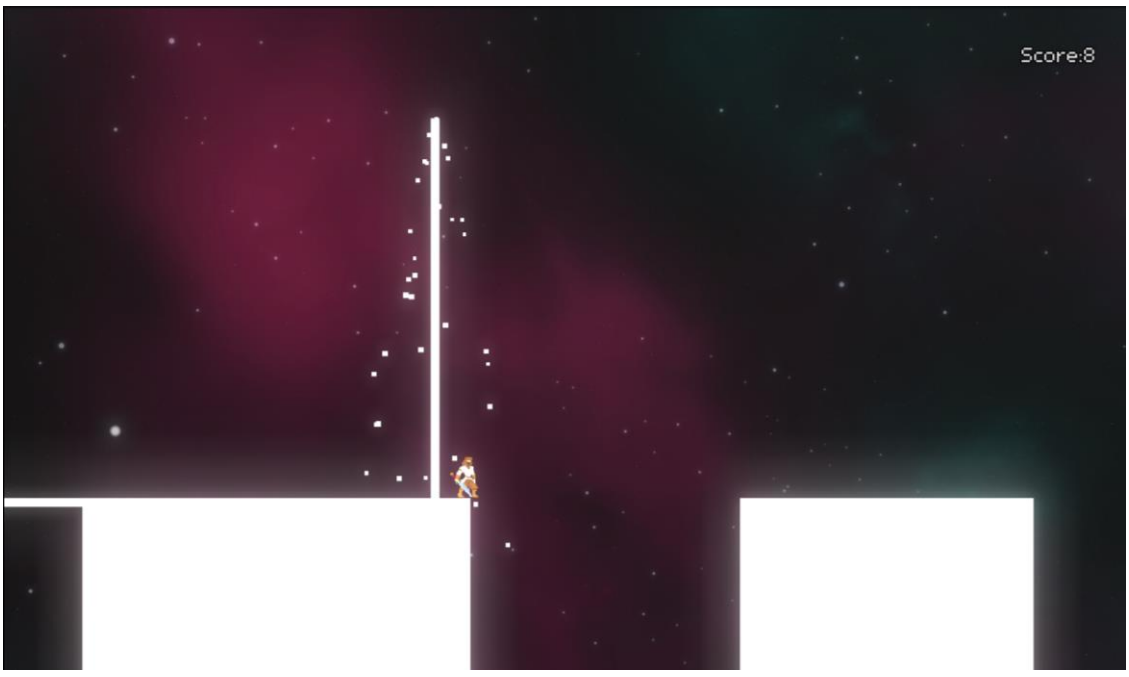
= profil matematică – informatică =

Profesor îndrumător,
Florica Ciurescu

Absolvent,
Turcu
Mihnea Alexandru

Vaslui, 2022

Dezvoltarea si proiectarea jocuri lor si aplicațiilor in Godot Engine



INTRODUCERE

Creare jocurilor cu un engine este de multe ori mai simpla si mai rapidă decat cu ajutorul unui limbaj de programare si al unui API grafic, fiind mai usoară si mai ușor de înțeles. în unele cazuri este necesară utilizarea unui astfel tip de unealtă deoarece sunt anumite utilitați care nu pot fi realizate in timp util, un altfel de exemplu ar fi exportarea aplicației pe diferite platforme.

Un dezavantaj este viteza cu care ruleaza programul deoarece engine-ul nu este optimizat special pentru proiectul tau si particularitatile pe care le contine acesta.
Intelegerea functionarii unui engine este utila celor care vor sa aprofundeze dezvoltarea de jocuri/aplicatii si programarea modularizata sau persoanelor care vor sa profeseze in industria jocurilor video sau a dezvoltarii de aplicatii.

Aceasta lucrare își propune prezentarea nu doar a consideratiilor teoretice, dar si a unui proiect realizat intr-un astfel de engine, Godot, cu explicatii detaliate si a implementarilor in limbajul GDScript.

Proiectul constă intr-un joc in care jucătorului îi este atribuit un personaj pe care trebuie sa il treaca in siguranta de pe un pilon pe celalalt.

Fiind pasionat de jocuri am hotărât sa prezint un joc și o modalitate de creare a acestuia prin intermediul engine-ului Godot pe care l-am considerat util în cadrul aplicației mele.

CONSIDERAȚII TEORETICE

Engine-ul Godot

Godot Engine e un game engine plin de utilități, cross-platform folosist pentru crearea jocurilor si aplicatiilor 2D si 3D printr-o interfață unificata. Acesta asigură un set complet de unelte uzuale, astfel utilizatorul se poate concentra pe crearea jocurilor fara sa reinventeze roata.

Jocurile sau aplicațiile pot fi exportate printr-un "click" pentru diferite platforme, inclusiv platformele majore de desktop (Linux, macOS, Windows) precum si cele pentru telefoanele mobile(Android, IOS) cat si cele bazate pe web(HTML5).

Godot este complet gratis si are sursa libera sub licența permisivă MIT. Jocurile sau aplicațiile utilizatorilor sunt ale lor si numai ale lor. Dezvoltarea engine-ului e complet independent si condusă de comunitatate, lăsând utilizatorii sa contureze engine-ul astfel încat să le întâlnească așteptările

Acest joc a fost scris in limbajele specifice engine-ului Godot si anume GDScript si TSCN.

GDScript:

Astfel arată sintaxa limbajului GDScript:

```
program = [ inheritance NEWLINE ] [ className ] { topLevelDecl
} ;

inheritance = "extends" ( IDENTIFIER | STRING ) { "."
IDENTIFIER } ;
className = "class_name" IDENTIFIER [ "," STRING ] NEWLINE ;

topLevelDecl
    = classVarDecl
    | constDecl
    | signalDecl
    | enumDecl
    | methodDecl
    | constructorDecl
    | innerClass
    | "tool"
    ;

classVarDecl = [ "onready" ] [ export ] "var" IDENTIFIER [ ":"
typeHint ]
    [ "=" expression ] [ setget ] NEWLINE ;
```

```
setget = "setget" [ IDENTIFIER ] [ "," IDENTIFIER] ;
export = "export" [ "(" [ BUILTINTYPE | IDENTIFIER { ","
literal } ] ")" ] ;
typeHint = BUILTINTYPE | IDENTIFIER ;

constDecl = "const" IDENTIFIER [ ":" typeHint ] "=" expression
NEWLINE ;

signalDecl = "signal" IDENTIFIER [ signalParList ] NEWLINE ;
signalParList = "(" [ IDENTIFIER { "," IDENTIFIER } ] ")" ;

enumDecl = "enum" [ IDENTIFIER ] "{" [ IDENTIFIER [ "="
INTEGER ]
    { "," IDENTIFIER [ "=" INTEGER ] } [ "," ] ] "}" NEWLINE ;

methodDecl = [ rpc ] [ "static" ] "func" IDENTIFIER "(" [
parList ] ")"
    [ "->" typeHint] ":" stmtOrSuite ;
parList = parameter { "," parameter } ;
parameter = [ "var" ] IDENTIFIER [ ":" typeHint ] [ "="
expression ] ;
rpc = "remote" | "master" | "puppet"
    | "remotesync" | "mastersync"  | "puppetsync";

constructorDecl = "func" IDENTIFIER "(" [ parList ] ")"
    [ "." "(" [ argList ] ")" ] ":" stmtOrSuite ;
argList = expression { "," expression } ;

innerClass = "class" IDENTIFIER [ inheritance ] ":" NEWLINE
    INDENT [ inheritance NEWLINE ] topLevelDecl { topLevelDecl
} DEDENT ;

stmtOrSuite = stmt | NEWLINE INDENT suite DEDENT ;
suite = stmt { stmt };

stmt
    = varDeclStmt
    | ifStmt
    | forStmt
    | whileStmt
    | matchStmt
    | flowStmt
    | assignmentStmt
    | exprStmt
    | assertStmt
    | yieldStmt
    | preloadStmt
    | "breakpoint" stmtEnd
    | "pass" stmtEnd
    ;
stmtEnd = NEWLINE | ";" ;
```

```
ifStmt = "if" expression ":" stmtOrSuite { "elif" expression
":" stmtOrSuite }
    [ "else" ":" stmtOrSuite ] ;
whileStmt = "while" expression ":" stmtOrSuite;
forStmt = "for" IDENTIFIER "in" expression ":" stmtOrSuite ;

matchStmt = "match" expression ":" NEWLINE INDENT matchBlock
DEDENT;
matchBlock = patternList ":" stmtOrSuite { patternList ":"
stmtOrSuite };
patternList = pattern { "," pattern } ;
(* Note: you can't have a binding in a pattern list, but to
not complicate the
grammar more it won't be restricted syntactically *)
pattern = literal | BUILTINTYPE | CONSTANT | "_" |
bindingPattern
    | arrayPattern | dictPattern ;
bindingPattern = "var" IDENTIFIER ;
arrayPattern = "[" [ pattern { "," pattern } [ ".." ] ] "]" ;
dictPattern = "{" [ keyValuePattern ] { "," keyValuePattern }
[ ".." ] "}" ;
keyValuePattern = STRING [ ":" pattern ] ;

flowStmt
    = "continue" stmtEnd
    | "break" stmtEnd
    | "return" [ expression ] stmtEnd
    ;

assignmentStmt = subscription "=" expression stmtEnd;
varDeclStmt = "var" IDENTIFIER [ "=" expression ] stmtEnd;

assertStmt = "assert" "(" expression [ "," STRING ] ")"
stmtEnd ;
yieldStmt = "yield" "(" [ expression "," expression ] ")" ;
preloadStmt = "preload" "(" CONSTANT ")" ;

(* This expression grammar encodes precedence. Items later in
the list have
higher precedence than the ones before. *)
exprStmt = expression stmtEnd ;
expression = cast [ "[" expression "]" ] ;
cast = ternaryExpr [ "as" typeHint ];
ternaryExpr = logicOr [ "if" logicOr "else" logicOr ] ;
logicOr = logicAnd { ( "or" | "||" ) logicAnd } ;
logicAnd = logicNot { ( "and" | "&&" ) logicNot };
logicNot = ( "!" | "not" ) logicNot | in;
in = comparison { "in" comparison };
comparison = bitOr { ( "<" | ">" | "<=" | ">=" | "==" | "!=" )
bitOr } ;
```

```
bitOr = bitXor { "|" bitXor } ;
bitXor = bitAnd { "^" bitAnd } ;
bitAnd = bitShift { "&" bitShift } ;
bitShift = minus { ( "<<" | ">>" ) minus } ;
minus = plus { "-" plus } ;
plus = factor { "+" factor } ;
factor = sign { ( "*" | "/" | "%" ) sign } ;
sign = ( "-" | "+" ) sign | bitNot ;
bitNot = "~" bitNot | is ;
is = call [ "is" ( IDENTIFIER | BUILTINTYPE ) ] ;
call = attribute [ "(" [ argList ] ")" ];
attribute = subscription { "." IDENTIFIER } ;
subscription = primary [ "[" expression "]" ] ;
primary = "true" | "false" | "null" | "self" | literal |
arrayDecl
    | dictDecl | "(" expression ")" ;

literal = STRING | NUMBER | IDENTIFIER | BUILTINTYPE
    | "PI" | "TAU" | "NAN" | "INF" ;
arrayDecl = "[" [ expression { "," expression } "," ] "]" ;
dictDecl = "{" [ keyValue { "," keyValue } "," ] "}" ;
keyValue
    = expression ":" expression
    | IDENTIFIER "=" expression
    ;
```

Fizica din motorul Godot

Introducerea in fizica motorului

În dezvoltarea jocurilor, ai nevoie de multe ori să poți afla dacă două obiecte se intersectează sau intră in contact. Aceasta este cunoscută ca **detectarea coliziunilor** Când o coliziune este detectată, deobicei vrei să se întample ceva. Asta este cunoscută ca **răspunsul coliziunilor.**

Sunt multe moduri obiecte de coliziune atât in 3D cat si 3D pentru a asigura detectarea coliziunnilor si răspunsul coliziunilor. Să incerci sa decizi care pe care dintre acestea două să le folosești poate fi derutant. Poți sa eviti probleme și să simplifici dezvoltarea daca înțelegi cum functionează fiecare și ce avantaje si dezavantaje au fiecare.

Aici vă voi prezenta:

- Cele patru tipuri de coliziuni ale Godot-ului
- Cum functioneaza fiecare obiect de coliziune

- Când și de ce sa îl folosești pe unul in locul celuilalt.

Exemplele acestui document vor folosi obiecte 2D. Fiecare Obiect 2D de ficică și de formă de coliziune are un echivalent in 3D și în cele mai multe cazuri vor funcționa in același fel.

Obiecte de coliziune

Godot oferă patru tipuri de corpuri fizice, extinzând CollisionObject2D:

- Area2D

  Nodurile `Area2D` asigură **detecție** și **influență.** Ei pot detecta când obiectele se suprapun și emit semnale când obiectele intră sau ies. Un `Area2D` poate fi de asemenea folosit pentru a suprapune propietățile fizice, precum gravitația sau amortizarea, intr-o arie definită.

Celelalte trei corpuri extind PhysicsBody2D:

- StaticBody2D

  Un corp static este un corp care nu este mutat de motorul de fizică. El participă în detecția coliziunilor dar nu răspunde la aceste coliziuni. Ele sunt cele mai des folosite pentru obiecte care sunt parte din mediul înconjurător sau care nu au nevoie de niciun comportament dinamic.

- RigidBody2D

  Acesta este nodul care implementeaza simulări de fizică 2D. Nu poți controla un `RigidBody2D` direct, dar în loc aplici forțele asupra ei (gravitație, impulsuri etc.) și motorul de fizică calculează mișcarea rezultată.

- KinematicBody2D

  Un corp ce asigură detecția coliziunii, dar nu și fizici. Toare mișcările si coliziunile acestui corp trebuie implementare in cod.

Material de fizică

Corpurile statice si cele rigide pot fi configurate să folosească un meterial fizic. Asta permite ajustarea frictiunilor si reflectărilor de pe un obiect, și setat dacă este absorbant sau/și aspru.

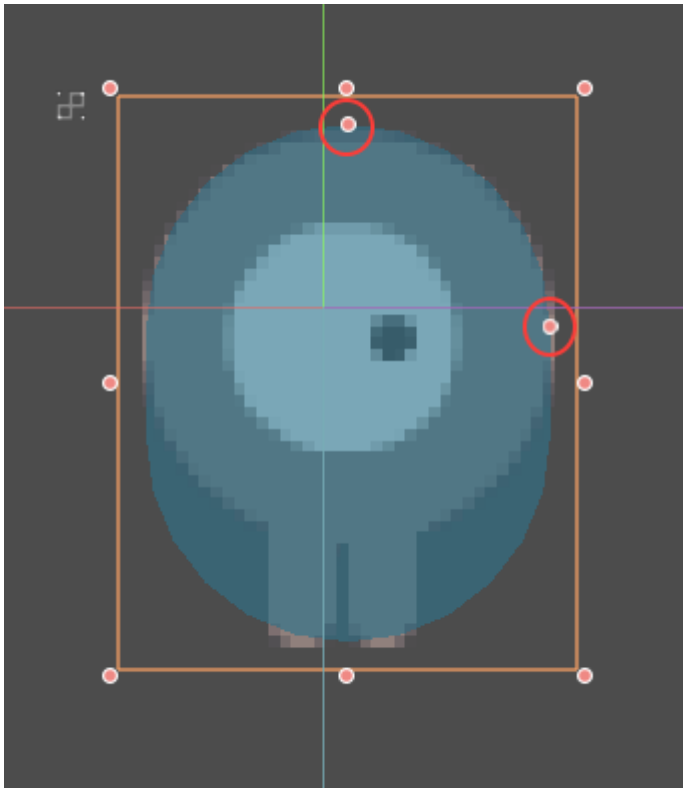
Formele de coliziune

Un corp de fizica poate să țină orice numar de Shape2D ca și copii. Aceste forme sunt folosite pentru a defini limitele de coliziune ale unui obiect si pentru a detecta contactul cu alte obiecte.

Pentru a detecta coliziunile, măcar un `Shape2D` trebuie atribuit obiectului.

Cea mai frecventa metodă de a atribui o forma este prin adaugarea a unui CollisionShape2D sua CollisionPolygon2D ca un copil al acestui obiect. Aceste noduri îți permit să desenzi forma direct in spațiul de lucru al editorului.

Important este să ai grijă să iti redimensionezi formele de coliziune în editor. Propietea “Scale” in Inspector ar trebui să rămână (1, 1). Când schimbi mărimea forma coliziunii, ar trebui folosite mereu handle-urile de mărime, nu cele specifice Node2D-ului. Redimensionarea unei forme poate rezulta in comportament al colizionilor neasteptat.



Proces de vizica callback

Motorul de fizica poate să genereze mai multe thread-uri pentru a îmbunătăți performanța, așa că poate folosi până la un frame întreg pentru a procesa fizica. Din această cauză, variabilele de stăre ale unui corp variază, cum sunt `position` sau`linear velocity` care s-ar putea să nu fie precise pentru frame-ul curent.

Pentru a evita această inprecizie, orice cod care acceseaza propietățile unui corp ar trebui să fie rulate in callback-ul Node._physics_process(), care este apelat inainte de fiecare pas al fizicii la un frame rate consecvent(60 de ori pe secund implicit). Aceastei metode îi va fi trimisă un parametru delta, care este un numer de tip float egal cu timpul trecut in *secunde* de la ultimul pas. Când folosing setarea implicită de 60 Hz rată de actualizare, va fi tipic egal cu 0.01666… (dar nu mereu)

Este recomandat să utilizați întotdeauna parametrul delta atunci când este relevant în calculele fizice, astfel încât jocul să se comporte corect dacă modificați rata de actualizare fizică sau dacă dispozitivul jucătorului nu poate ține pasul.

Collision layers and masks

Una dintre cele mai puternice, dar adesea înțelese greșit, caracteristici de coliziune este sistemul de straturi de coliziune. Acest sistem vă permite să construiți interacțiuni complexe între o varietate de obiecte. Conceptele cheie sunt straturile și măștile. Fiecare CollisionObject2D are 20 de straturi diferite de fizică cu care poate interacționa.

Să ne uităm la fiecare dintre proprietățile pe rând:

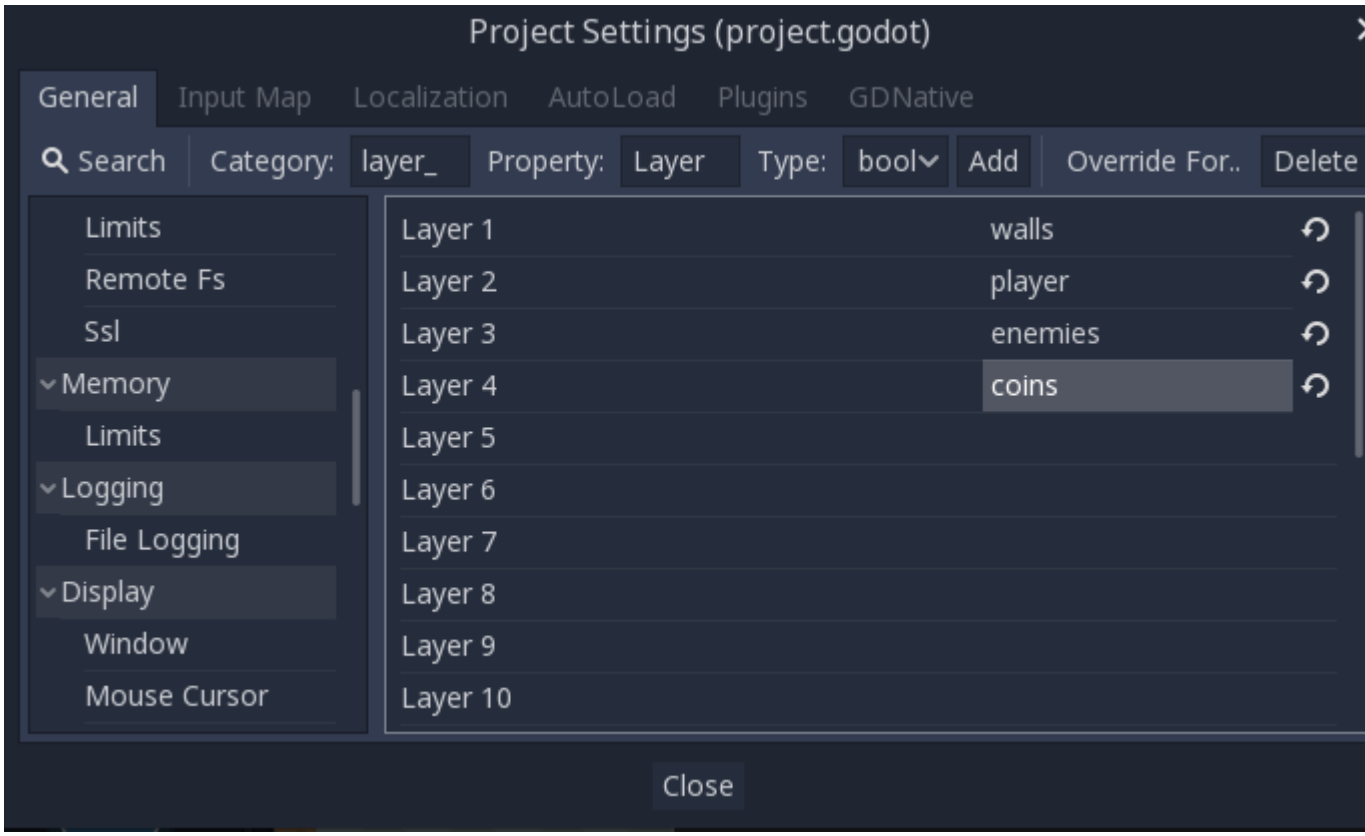
- collision_layer

  Aceasta descrie straturile în care apare obiectul. În mod implicit, toate corpurile sunt pe stratul 1.

- collision_mask

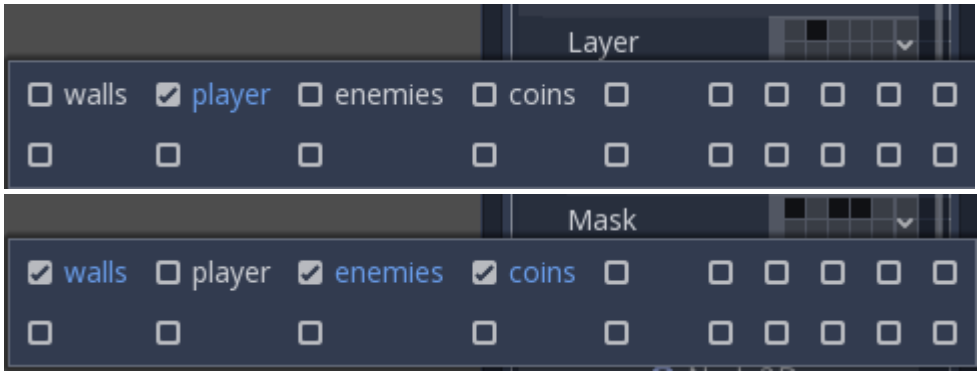
  Aceasta descrie straturile pe care corpul le va scana pentru coliziuni. Dacă un obiect nu se află într-unul dintre straturile măștii, corpul îl va ignora. În mod implicit, toate corpurile scanează stratul 1.

Aceste proprietăți pot fi configurate prin cod sau prin editarea lor în Inspector. Urmărirea a ceea ce utilizați fiecare strat tematic poate fi dificilă, astfel încât este util să atribuiți nume straturilor pe care le utilizați. Numele pot fi atribuite în Setări proiect -> Nume straturi.



GUI example

Ai patru tipuri de noduri în joc: Pereți, Jucător, Inamic și Monedă. Atât jucătorul, cât și inamicul ar trebui să se ciocnească cu Walls. Nodul Player ar trebui să detecteze coliziunile atât cu inamicul, cât și cu moneda, dar inamicul și moneda ar trebui să se ignore reciproc. Începeți prin a numi straturile 1-4 "pereți", "jucător", "dușmani" și "monede" și plasați fiecare tip de nod în stratul respectiv folosind proprietatea "Layer". Apoi setați proprietatea "Mască" a fiecărui nod selectând straturile cu care ar trebui să interacționeze. De exemplu, setările jucătorului ar arăta astfel:



Code example

În apelurile de funcții, straturile sunt specificate ca bitmask. În cazul în care o funcție permite toate straturile în mod implicit, masca stratului va fi dată ca 0x7fffffff. Codul poate utiliza notația binară, hexazecimală sau zecimală pentru măștile strat, în funcție de preferințele dvs. Echivalentul de cod al exemplului de mai sus în care au fost activate straturile 1, 3 și 4 ar fi după cum urmează:

```
# Example: Setting mask value for enabling layers 1, 3 and 4

# Binary - set the bit corresponding to the layers you want to enable (1,
3, and 4) to 1, set all other bits to 0.
# Note: Layer 20 is the first bit, layer 1 is the last. The mask for layers
4,3 and 1 is therefore
0b00000000000000001101
# (This can be shortened to 0b1101)

# Hexadecimal equivalent (1101 binary converted to hexadecimal)
0x000d
# (This value can be shortened to 0xd)

# Decimal - Add the results of 2 to the power of (layer be enabled-1).
# (2^(1-1)) + (2^(3-1)) + (2^(4-1)) = 1 + 4 + 8 = 13
pow(2, 1) + pow(2, 3) + pow(2, 4)
```

Area2D

Nodurile zonale oferă detecție și influență. Ele pot detecta când obiectele se suprapun și emit semnale atunci când corpurile intră sau ies. Zonele pot fi, de asemenea, utilizate pentru a suprascrie proprietățile fizicii, cum ar fi gravitația sau amortizarea, într-o zonă definită. Există trei utilizări principale pentru Area2D:

- Parametrii fizici superiori (cum ar fi gravitația) într-o anumită regiune.

- Detectarea momentului în care alte organisme intră sau ies dintr-o regiune sau ce organisme se află în prezent într-o regiune.
- Verificarea altor zone pentru suprapunere.

În mod implicit, zonele primesc, de asemenea, intrarea mouse-ului și a ecranului tactil.

StaticBody2D

Un corp static este unul care nu este mișcat de motorul fizic. Participă la detectarea coliziunilor, dar nu se mișcă ca răspuns la coliziune. Cu toate acestea, poate da mișcare sau rotație unui corp care se ciocnește ca și cum s-ar mișca, folosindu-și constant_linear_velocity și proprietățile constant_angular_velocity. Nodurile StaticBody2D sunt cel mai adesea folosite pentru obiecte care fac parte din mediu sau care nu trebuie să aibă niciun comportament dinamic.

Exemplu de utilizări pentru StaticBody2D:

• Platforme (inclusiv platforme în mișcare)

• Benzi transportoare

• Pereți și alte obstacole

RigidBody2D

Acesta este nodul care implementează fizica 2D simulată. Nu controlați direct un RigidBody2D. În schimb, îi aplici forțe, iar motorul fizic calculează mișcarea rezultată, inclusiv coliziunile cu alte corpuri și răspunsurile la coliziune, cum ar fi vigurosul, rotirea etc. Puteți modifica comportamentul unui corp rigid prin proprietăți precum "Masă", "Frecare" sau "Respingere", care pot fi setate în Inspector. Comportamentul organismului este, de asemenea, afectat de proprietățile lumii, așa cum sunt setate în Project Settings -> Physics, sau prin introducerea unui Area2D care depășește proprietățile fizicii globale. Când un corp rigid este în repaus și nu s-a mișcat de ceva vreme, se culcă. Un corp de dormit acționează ca un corp static, iar forțele sale nu sunt calculate de motorul fizic. Corpul se va trezi atunci când forțele sunt aplicate, fie printr-o coliziune, fie prin cod.

Moduri rigide ale corpului

Un corp rigid poate fi setat la unul dintre cele patru moduri:

- **Rigid** - Corpul se comportă ca un obiect fizic. Se ciocnește cu alte corpuri și răspunde forțelor aplicate. Acesta este modul implicit.
- **Static** - Corpul se comportă ca un StaticBody2D și nu se mișcă.

- **Character** - Similar cu modul "Rigid", dar corpul nu se poate roti.
- **Kinematic** - Corpul se comportă ca un KinematicBody2D și trebuie să fie mișcat de cod.

Using RigidBody2D

Unul dintre beneficiile utilizării unui corp rigid este că o mulțime de comportament poate fi avut "gratuit" fără a scrie niciun cod. De exemplu, dacă ați face un joc în stil "Angry Birds" cu blocuri care se încadrează, ar trebui doar să creați RigidBody2Ds și să le ajustați proprietățile. Stivuire, care se încadrează, și viguros ar fi automat calculate de către motorul de fizica. Cu toate acestea, dacă doriți să aveți un anumit control asupra corpului, ar trebui să aveți grijă - modificarea poziției, a linear_velocity sau a altor proprietăți fizice ale unui corp rigid poate duce la un comportament neașteptat. Dacă trebuie să modificați oricare dintre proprietățile legate de fizică, ar trebui să utilizați apelul _integrate_forces() în loc de _physics_process(). În acest apel invers, aveți acces la Physics2DDirectBodyState al corpului, care permite schimbarea în siguranță a proprietăților și sincronizarea acestora cu motorul fizic. De exemplu, aici este codul pentru o navă spațială în stil "Asteroizi":

```
extends RigidBody2D

var thrust = Vector2(0, 250)
var torque = 20000

func _integrate_forces(state):
    if Input.is_action_pressed("ui_up"):
        applied_force = thrust.rotated(rotation)
    else:
        applied_force = Vector2()
    var rotation_dir = 0
    if Input.is_action_pressed("ui_right"):
        rotation_dir += 1
    if Input.is_action_pressed("ui_left"):
        rotation_dir -= 1
    applied_torque = rotation_dir * torque
```

Rețineți că nu setăm direct proprietățile linear_velocity sau angular_velocity, ci mai degrabă aplicăm forțe (tracțiune și cuplu) pe corp și lăsăm motorul fizic să calculeze mișcarea rezultată.

Când un corp rigid merge la culcare, funcția _integrate_forces() nu va fi numită. Pentru a suprascrie acest comportament, va trebui să păstrați corpul treaz prin crearea unei coliziuni, aplicarea unei forțe la acesta sau prin dezactivarea proprietății can_sleep. Fiți conștienți de faptul că acest lucru poate avea un efect negativ asupra performanței.

Contact reporting

În mod implicit, corpurile rigide nu țin evidența contactelor, deoarece acest lucru poate necesita o cantitate imensă de memorie dacă multe corpuri se află în scenă. Pentru a activa raportarea persoanelor de contact, setați proprietatea contacts_reported la o valoare non-zero. Contactele pot fi apoi obținute prin intermediul Physics2DDirectBodyState.get_contact_count() și al funcțiilor conexe. Monitorizarea contactelor prin semnale poate fi activată prin intermediul proprietății contact_monitor. Consultați RigidBody2D pentru lista de semnale disponibile.

KinematicBody2D

Corpurile CinemaematicBody2D detectează coliziunile cu alte corpuri, dar nu sunt afectate de proprietățile fizicii, cum ar fi gravitația sau frecarea. În schimb, acestea trebuie să fie controlate de utilizator prin cod. Motorul fizic nu va muta un corp cinematic.

Atunci când mutați un corp cinematic, nu trebuie să-i setați poziția direct. În schimb, utilizați metodele move_and_collide() sau move_and_slide(). Aceste metode mișcă corpul de-a lungul unui anumit vector și se va opri instantaneu dacă se detectează o coliziune cu un alt corp. După ce corpul s-a ciocnit, orice răspuns de coliziune trebuie codificat manual.

Kinematic collision response

După o coliziune, poate doriți ca corpul să ricoșeze, să alunece de-a lungul unui perete sau să modifice proprietățile obiectului pe care l-a lovit. Modul în care gestionați răspunsul la coliziune depinde de metoda pe care ați utilizat-o pentru a muta KinematicBody2D.

move_and_collide

Atunci când se utilizează move_and_collide(), funcția returnează un obiect KinematicCollision2D, care conține informații despre coliziune și corpul care se ciocnește. Puteți utiliza aceste informații pentru a determina răspunsul.

De exemplu, dacă doriți să găsiți punctul din spațiul în care a avut loc coliziunea:

```
extends KinematicBody2D

var velocity = Vector2(250, 250)

func _physics_process(delta):
    var collision_info = move_and_collide(velocity * delta)
    if collision_info:
        var collision_point = collision_info.position
```

Sau pentru a sări de pe obiectul care se ciocnește:

```
extends KinematicBody2D

var velocity = Vector2(250, 250)

func _physics_process(delta):
    var collision_info = move_and_collide(velocity * delta)
    if collision_info:
        velocity = velocity.bounce(collision_info.normal)
```

move_and_slide

Alunecarea este un răspuns comun la coliziune; imaginați-vă un jucător care se deplasează de-a lungul pereților într-un joc de sus în jos sau care rulează în sus și în jos pante într-un platformer. Deși este posibil să vă codați singur acest răspuns după utilizarea move_and_collide(), move_and_slide() oferă o modalitate convenabilă de a implementa mișcarea de alunecare fără a scrie prea mult cod.

`move_and_slide()` include automat pasul de timp în calculul său, deci nu ar trebui să înmulțiți vectorul de viteză cu delta.

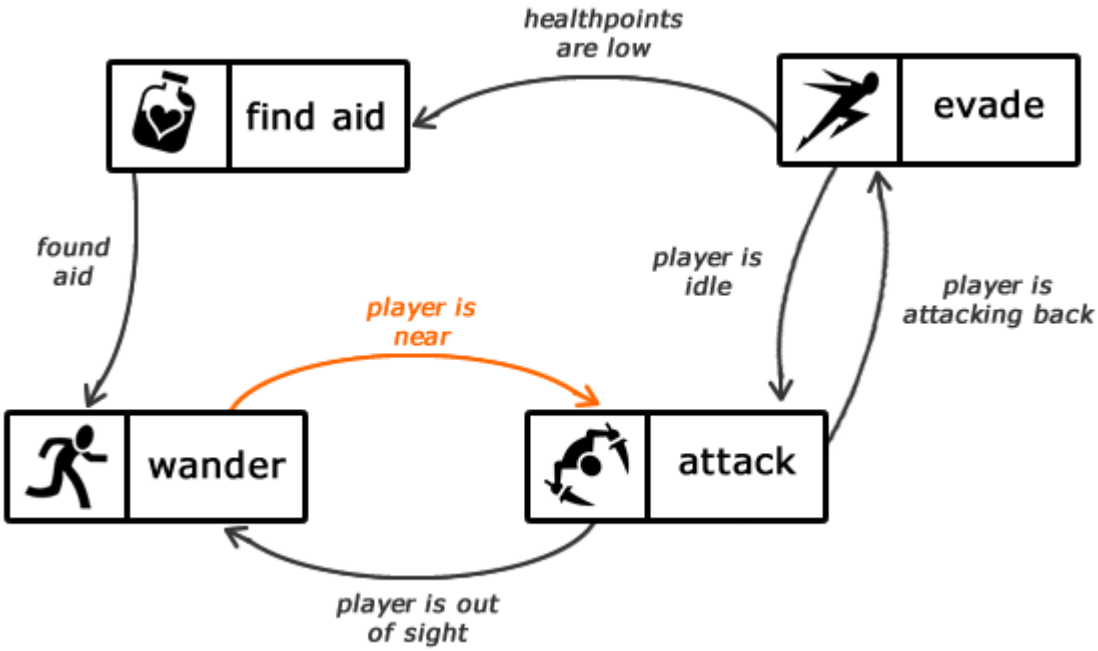
De exemplu, utilizați următorul cod pentru a face un caracter care poate merge de-a lungul solului (inclusiv pante) și să sară atunci când se află pe pământ: se extinde KinematicBody2D

```
extends KinematicBody2D

var run_speed = 350
var jump_speed = -1000
var gravity = 2500

var velocity = Vector2()

func get_input():
    velocity.x = 0
    var right = Input.is_action_pressed('ui_right')
    var left = Input.is_action_pressed('ui_left')
    var jump = Input.is_action_just_pressed('ui_select')

    if is_on_floor() and jump:
        velocity.y = jump_speed
    if right:
        velocity.x += run_speed
    if left:
        velocity.x -= run_speed

func _physics_process(delta):
    velocity.y += gravity * delta
    get_input()
    velocity = move_and_slide(velocity, Vector2(0, -1))
```

Mașini finite de stări

O mașină cu stare finită este un model folosit pentru a reprezenta și controla fluxul de execuție. Este perfect pentru implementarea de AI-uri în jocuri, producând rezultate excelente fără un cod complex.

Ce este o mașină cu stare finită? O mașină cu stare finită, sau FSM pe scurt, este un model de calcul bazat pe o mașină ipotetică făcută din una sau mai multe stări. Doar o singură stare poate fi activă în același timp, astfel încât mașina trebuie să treacă de la o stare la alta pentru a efectua acțiuni diferite. FSM-urile sunt utilizate în mod obișnuit pentru a organiza și reprezenta un flux de execuție, care este util pentru a implementa AI în jocuri. "Creierul" unui inamic, de exemplu, poate fi implementat folosind un FSM: fiecare stat reprezintă o acțiune,



cum ar fi atacul sau evitarea:

Un FSM poate fi reprezentat de un grafic, unde nodurile sunt stările, iar marginile sunt tranzițiile. Fiecare margine are o etichetă care informează când ar trebui să se întâmple tranziția, cum ar fi jucătorul este aproape de etichetă în figura de mai sus, ceea ce indică faptul că mașina va trece de la rătăcire la atac dacă jucătorul este aproape.

APLICAȚII

Pentru acest proiect am profitat la maxim de natura oject-oriented a engine-ului, in special in realizarea personajului controlat de jucator.

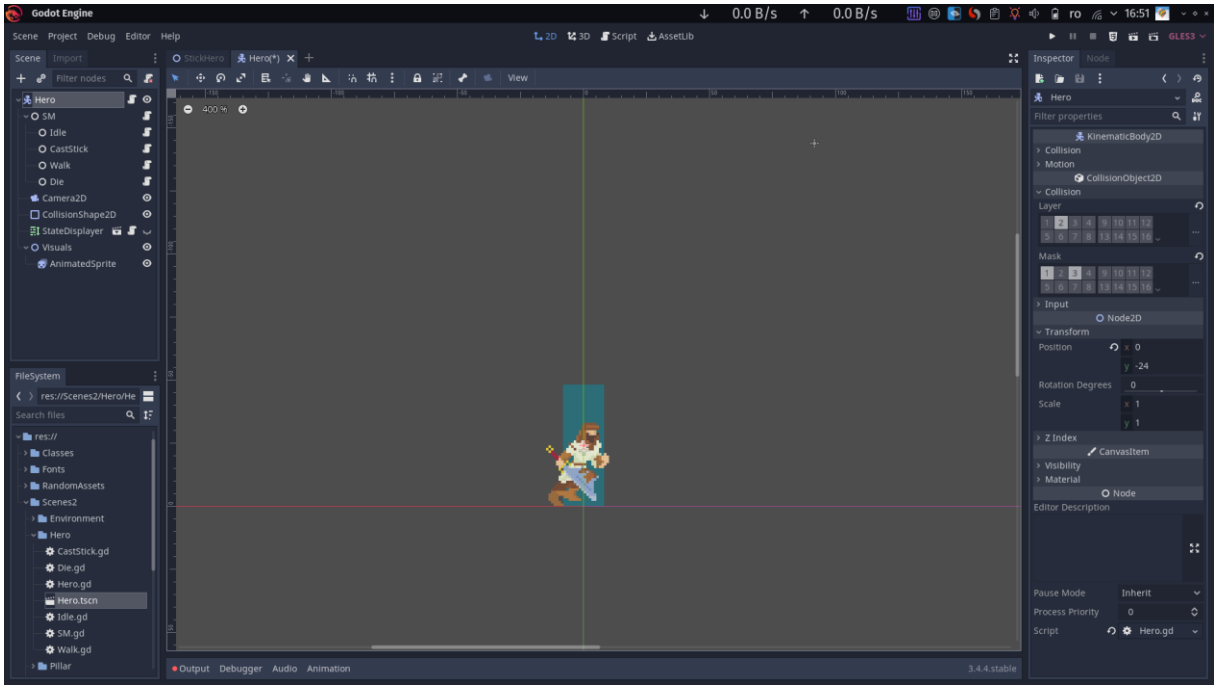
Scena Hero.tscn:

```
[gd_scene load_steps=22 format=2]

[ext_resource path="res://Scenes2/Hero/Hero.gd" type="Script"
id=2]
[ext_resource path="res://Scenes2/Hero/SM.gd" type="Script"
id=3]
[ext_resource path="res://Scenes2/Hero/Idle.gd" type="Script"
id=4]
[ext_resource path="res://Scenes2/Hero/CastStick.gd"
type="Script" id=5]
[ext_resource path="res://Scenes2/Hero/Die.gd" type="Script"
id=6]
[ext_resource path="res://Scenes2/Hero/Walk.gd" type="Script"
id=7]
[ext_resource
path="res://Classes/StateMachine/StateDisplayer/RichTextLabel.
tscn" type="PackedScene" id=8]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Idle/LightBandit_Idle_3.png" type="Texture" id=10]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Idle/LightBandit_Idle_1.png" type="Texture" id=11]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Idle/LightBandit_Idle_2.png" type="Texture" id=12]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Run/LightBandit_Run_2.png" type="Texture" id=13]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Run/LightBandit_Run_5.png" type="Texture" id=14]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Run/LightBandit_Run_4.png" type="Texture" id=15]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Run/LightBandit_Run_6.png" type="Texture" id=16]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Run/LightBandit_Run_1.png" type="Texture" id=17]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Run/LightBandit_Run_3.png" type="Texture" id=18]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Run/LightBandit_Run_7.png" type="Texture" id=19]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Run/LightBandit_Run_0.png" type="Texture" id=20]
[ext_resource path="res://RandomAssets/Bandits/Sprites/Light
Bandit/Jump/LightBandit_Jump_0.png" type="Texture" id=21]
```

```
[sub_resource type="RectangleShape2D" id=1]
extents = Vector2( 8, 24 )

[sub_resource type="SpriteFrames" id=2]
animations = [ {
"frames": [ ExtResource( 21 ) ],
"loop": true,
"name": "Die",
"speed": 5.0
}, {
"frames": [ ExtResource( 20 ), ExtResource( 17 ), ExtResource(
13 ), ExtResource( 18 ), ExtResource( 15 ), ExtResource( 14 ),
ExtResource( 16 ), ExtResource( 19 ) ],
"loop": true,
"name": "Walk",
"speed": 10.0
}, {
"frames": [ ExtResource( 11 ), ExtResource( 12 ), ExtResource(
10 ) ],
"loop": true,
"name": "Idle",
"speed": 5.0
} ]

[node name="Hero" type="KinematicBody2D"]
position = Vector2( 0, -24 )
collision_layer = 2
collision_mask = 5
script = ExtResource( 2 )

[node name="SM" type="Node" parent="."]
script = ExtResource( 3 )

[node name="Idle" type="Node" parent="SM"]
script = ExtResource( 4 )

[node name="CastStick" type="Node" parent="SM"]
script = ExtResource( 5 )

[node name="Walk" type="Node" parent="SM"]
script = ExtResource( 7 )
active = true

[node name="Die" type="Node" parent="SM"]
script = ExtResource( 6 )

[node name="Camera2D" type="Camera2D" parent="."]
position = Vector2( 91.51, -119.734 )
current = true
smoothing_enabled = true
```

```
[node name="CollisionShape2D" type="CollisionShape2D"
parent="."]
shape = SubResource( 1 )

[node name="RichTextLabel" parent="." instance=ExtResource( 8
)]
visible = false

[node name="Visuals" type="Node2D" parent="."]
position = Vector2( 0, 1.88504 )

[node name="AnimatedSprite" type="AnimatedSprite"
parent="Visuals"]
scale = Vector2( -1, 1 )
frames = SubResource( 2 )
animation = "Walk"
playing = true
```

Scena Hero.tscn in editor:



Scena este alcătuita din mai multe noduri fiecare având un rol bine definit.

Nodul-rădăcină(“Hero”) este unul de tip KinematicBody2D.

Scriptul atasat acestuia este responsabil de comportamentul specific al caracterului principal mosteneste clasa predefinita KinematicBody2D.

Scriptul Hero.gd atasat scenei Hero.tscn :

```
extends KinematicBody2D

onready var sm=get_node("SM")
onready var map=get_parent().get_node("Map")
onready var collision_shape=$CollisionShape2D
onready var aspr=$Visuals/AnimatedSprite

var speed:=200
var vel:=Vector2.ZERO

func do_idle():
      sm.request_state("Idle")

func walk_left():
      vel.x=200

func _physics_process(delta):
      vel.y+=10
      vel=move_and_slide(vel, Vector2.UP)
```

Aceasta este relativ simpla si se ocupă de interacțiunea cu mediul înconjurător al caracterului prin funcția move_and_slide.

Acțiunile propiu-zise ale personajului sunt controlate printr-o “mașină de stări”.

Scriptul StateMachine.gd atașat nodul “SM”:

```
extends Node
class_name StateMachine

var states={}
var active_states=[]

func add_state(x):
      states[x.state_name]=x
      x.pr=get_parent()



func _ready():
      for x in get_children():
            add_state(x)
      for x in get_children():
            if x.active:
                  request_state(x.state_name)

func _process(delta):
      for s in active_states:
```

```
                var cur=states[s]
                var newst=cur.get_transition()

                if newst==null:
                        cur._during_state(delta)
                else:
                        if newst is String:
                                if newst.begins_with("exit") :
                                        newst.erase(0,"exit".length())
                                        _deactivate(s)
                                request_state(newst)
                        elif newst is Array:
                                for st in newst:
                                        if st=="exit":
                                                _deactivate(s)
                                        else: request_state(st)
        pass

func _deactivate(s:String):
        active_states.erase(s)
        var cur=states[s]
        cur.active=false
        cur.exit_state(active_states)

func _activate(st:String):
        var cur=states[st]
        cur.enter_state(active_states)
        if !active_states.has(st):
                active_states.push_back(st)
        cur.active=true

func request_state(st:String)->bool:
        var cur=states[st]
        if !cur.has_dependecies():
                return false
        if cur.is_in_conflict():
                return false

        for s in cur.removing_states:
                if active_states.has(s):
                        _deactivate(s)
        _activate(st)
        return cur

func request_exit_state(st:String)->bool:
        _deactivate(st)
        return true
#
func is_active(st:String):
        return active_states.has(st)
```

Aceasta este responsabila de realizarea tranzițiilor intre stările in care se află caracterul si “activarea” altor script-uri atașate nodurilor-copii ale nodului “SM” ce realizează comportamentul specific al personajului in starile respective. Aceste script-uri moștenesc clasa “State”.

Scriptul State.gd :

```
extends Node
class_name State

onready var sm=get_parent()
onready var state_name:=self.name
var pr=null

export var active:=false

var parents=0

var conflicting_states=[]
var removing_states=[]
var necessary_states=[] # strings

#copy
func _ready():
	conflicting_states=[]
	removing_states=[]
	necessary_states=[]

func get_transition():
	return null

func enter_state(old_states):
	pass

func exit_state(new_states):
	pass

func _during_state(delta):
	pass
#

func activate():
	active=true
	enter_state(false)
	sm.request_state(state_name)

func deactivate():
	active=false
	exit_state(false)
```

```
func has_dependecies()->bool:
	for x in necessary_states:
		if !sm.active_states.has(x):
			return false
	return true

func is_in_conflict()->bool:
	for x in conflicting_states:
		if sm.active_states.has(x):
			return true
	return false
```

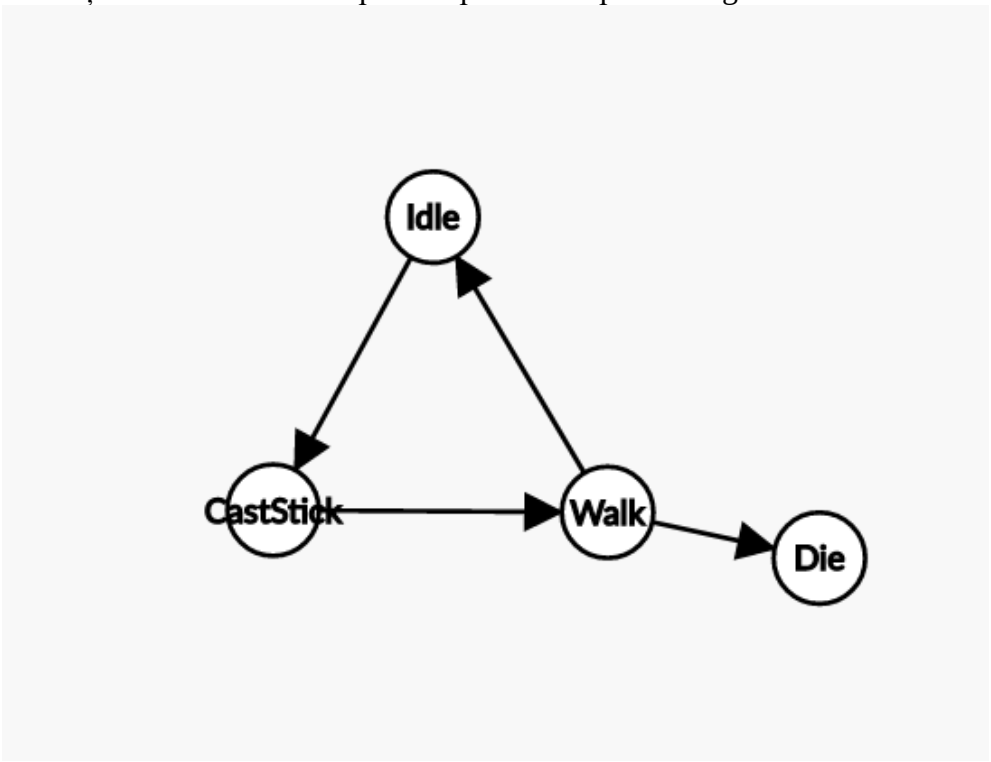
Copii nodului “SM” se numesc “Walk”, “CastStick”, “Idle” si “Die”. Si fiecare reprezinta starile posibile are personajului.

Scriptul, **Walk.gd,** ce se ocupă de mersul caracterului, arată astfel:

```
extends State

var do_idle:=false
signal idle

func _ready():
	connect("idle", self, "_idle")
	conflicting_states=[]
	removing_states=[]
	necessary_states=[]

func get_transition():
	if pr.position.y>200:
		return ["Die", "exit"]
	if do_idle:
		do_idle=false
		return ["Idle","exit"]
	return null

func enter_state(old_states):
	pr._ready()
	pr.aspr.play(name)

func exit_state(new_states):
	pass

func _during_state(delta):
	pass
```

Remarcați că anumite functii specifice clasei State sunt suprascrise in acest script(ca si in celelalte script-uri ce definesc stările caracterului). Acestea sunt cele responsabile de realizarea tranzitiei din aceasta stare in starile “Idle” si “Die”(get_transition), cea de a reda animatia de mers (enter_state), _during_state, exit_state si exit_state.

Personajul mai are alte 3 astfel de stari reprezentate de scripturile Idle.gd(activa atunci cand caracterul stationeaza), Die.gd(prezenta atunci cand jocatorul pierde nivelul), CastStick.gd(prezenta atunci cand jucatorul apasa space si ridica “batul”.
Tranzițiile intre aceste stari pot fi reprezentate printr-un graf:



Caracterul incepe in “Idle”, din “Idle” poate sa punta un bat trecand in “CastStick”, bat pe care merge, activand starea “Walk”. Dacă bățul are mărimea potrivita acesta poate sa meargă până se oprește, intrând in Idle altfel cade de pe băț si moare, activand starea “Die”.
Acesta este codul pentru aceste stari :

Die.gd :

```
extends State

var play_tscn=preload("res://Scenes2/Play/Play.tscn")
#copy
func _ready():
	conflicting_states=[]
	removing_states=[]
	necessary_states=[]

func get_transition():
	return null

func enter_state(old_states):
	get_tree().root.get_child(0).add_child(play_tscn.instanc
e())
	pass

func exit_state(new_states):
	pass

func _during_state(delta):
```

```
	pass
```

CastStick.gd:

```
extends State

onready var
stick_tscn=preload("res://Scenes2/Stick/Stick.tscn")
var stick_fallen:=false

#copy
func _ready():
	conflicting_states=[]
	removing_states=[]
	necessary_states=[]

func get_transition():
	if stick_fallen==true:
		stick_fallen=false
		return "Walk"
	return null

func _stick_fallen():
	stick_fallen=true

func enter_state(_old_states):
	pr.aspr.play(name)

	var stick=stick_tscn.instance()
	var shape_extent=pr.collision_shape.shape.extents
	stick._ready()
	stick.global_position=Vector2(pr.position.x+shape_extent
.x+stick.collision_shape.shape.extents.x+0.4-
50,stick.position.y)
	stick.connect("stick_fallen", self, "_stick_fallen")
	pr.get_parent().add_child(stick)

func exit_state(_new_states):
	pass

func _during_state(_delta):
	pass
```

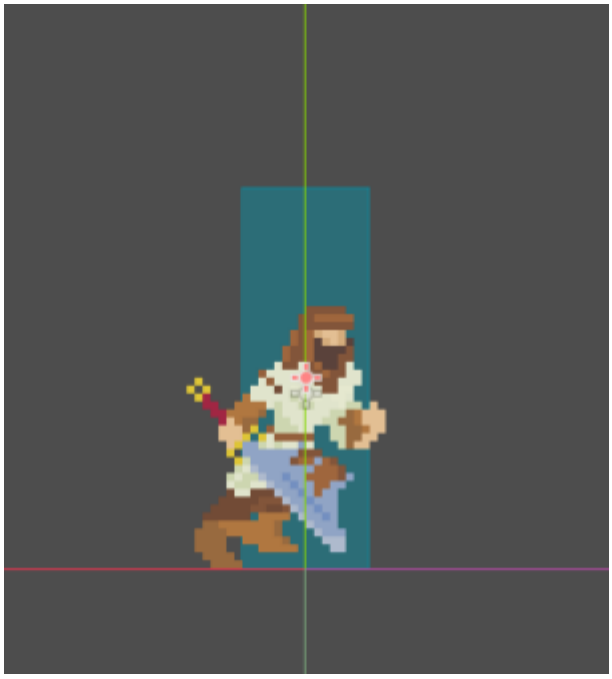
Idle.gd :

```
extends State

func _ready():
	conflicting_states=[]
	removing_states=["Walk"]
	necessary_states=[]
```

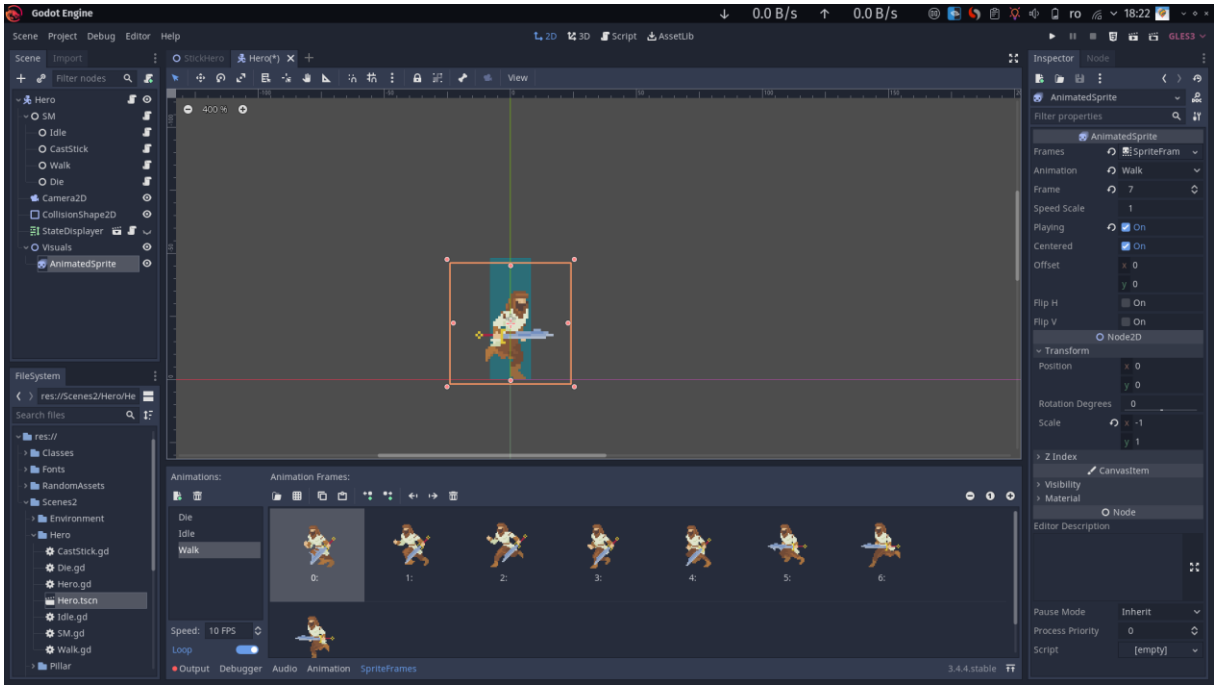
```
func get_transition():
	if Input.is_action_just_pressed("stick"):
		return ["CastStick","exit"]

	return null

func enter_state(old_states):
	pr.aspr.play(name)
	pr.vel=Vector2.ZERO
	pass

func exit_state(new_states):
	pass

func _during_state(delta):
	pass
```

Nodul “Camera2D” reprezinta camera jocului si este parentată eroului pentru a-l urmări pe acesta.

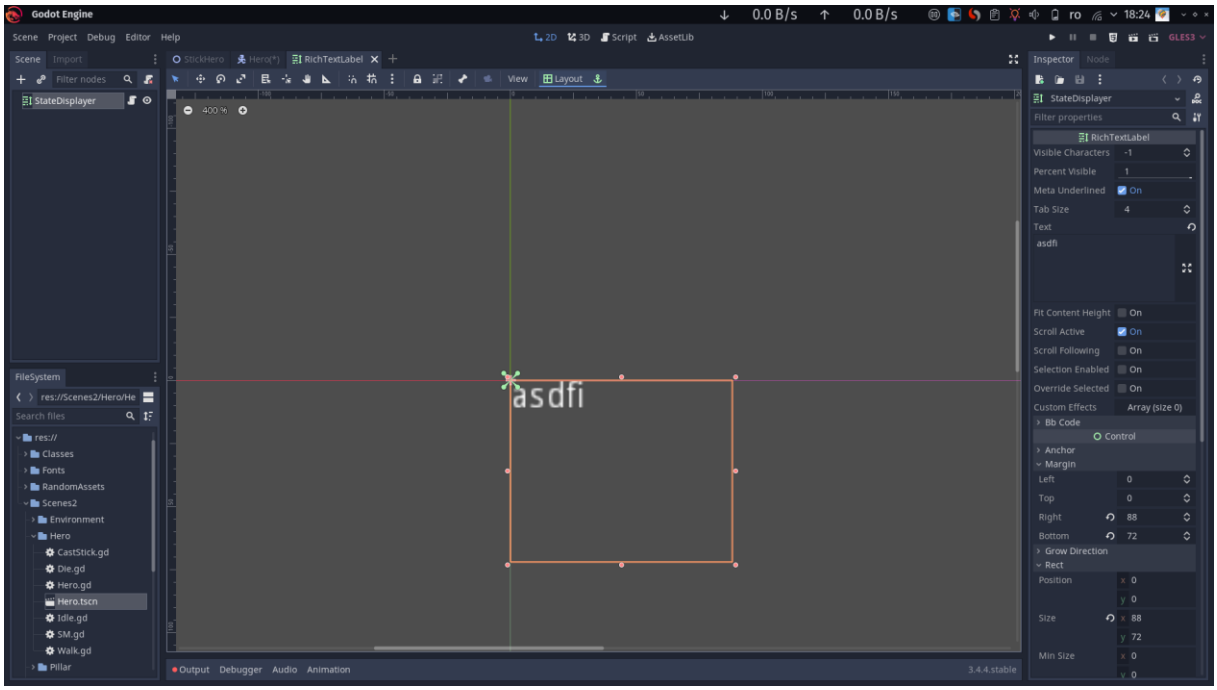
Nodul “CollisionShape2D” este forma de coliziune a personajului, fiind setată să fie un dreptunghi:



Nodul “Visuals” are parentat la el nodul “AnimatedSprite” ce contine animatiile pentru caracter:



Se mai remarca un nod care este defapt o scena numita **“StateDisplayer”** :



Aceasta afișeaza stările active in mașina de stari.

Scriptul StateDisplayer.gd:

```
extends RichTextLabel

onready var sm=get_parent().get_node("SM")
```

```
func _process(delta):
	text=String(sm.active_states)
```

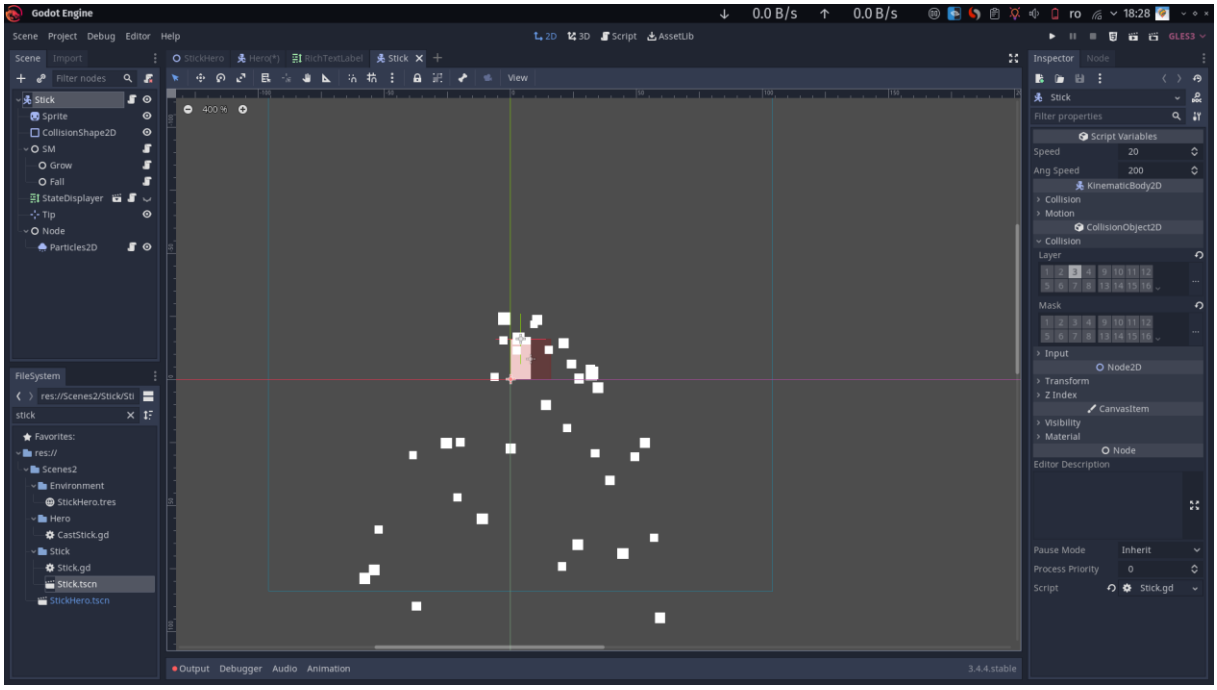
Atunci cand eroul intra in starea “CastStick” prin apasarea tastei SPACE in starea “Idle” acesta adaugă un “băț” in Scena principală:

Scena Stick.tscn:

```
[gd_scene load_steps=10 format=2]

[ext_resource path="res://Scenes2/Stick/Stick.gd" type="Script" id=1]
[ext_resource path="res://Scenes2/Stick/SM.gd" type="Script" id=2]
[ext_resource path="res://Scenes2/Stick/Grow.gd" type="Script" id=3]
[ext_resource path="res://Scenes2/Stick/Fall.gd" type="Script" id=4]
[ext_resource path="res://Classes/StateMachine/StateDisplayer/RichTextLabel.tscn"
type="PackedScene" id=5]
[ext_resource path="res://RandomAssets/Squares/white.png" type="Texture" id=6]
[ext_resource path="res://Scenes2/Stick/Particles2D.gd" type="Script" id=7]

[sub_resource type="RectangleShape2D" id=1]
extents = Vector2( 8, 8 )

[sub_resource type="ParticlesMaterial" id=2]
lifetime_randomness = 0.6
flag_disable_z = true
spread = 180.0
gravity = Vector3( 0, 97, 0 )
initial_velocity = 50.0
initial_velocity_random = 0.2
orbit_velocity = 0.0
orbit_velocity_random = 0.0
scale = 5.0
scale_random = 0.5

[node name="Stick" type="KinematicBody2D"]
collision_layer = 4
collision_mask = 0
script = ExtResource( 1 )

[node name="Sprite" type="Sprite" parent="."]
scale = Vector2( 0.25, 0.5 )
texture = ExtResource( 6 )
offset = Vector2( 16, -16 )

[node name="CollisionShape2D" type="CollisionShape2D" parent="."]
modulate = Color( 1, 0, 0, 1 )
position = Vector2( 8, -8 )
shape = SubResource( 1 )
disabled = true

[node name="SM" type="Node" parent="."]
```

```
script = ExtResource( 2 )
[node name="Grow" type="Node" parent="SM"]
script = ExtResource( 3 )
active = true
[node name="Fall" type="Node" parent="SM"]
script = ExtResource( 4 )

[node name="StateDisplayer" parent="." instance=ExtResource( 5 )]
visible = false

[node name="Tip" type="Position2D" parent="."]
position = Vector2( 3.994, -16 )

[node name="Node" type="Node" parent="."]

[node name="Particles2D" type="Particles2D" parent="Node"]
position = Vector2( 3.99426, -16 )
amount = 1000
lifetime = 46.9
speed_scale = 2.0
randomness = 1.0
fixed_fps = 60
local_coords = false
process_material = SubResource( 2 )
script = ExtResource( 7 )
```

Scena Stick.tscn in editor:



Se remarcă folosirea aceleiași implementări a mașinii de stări și în cadrul acestei scene care are ca stari “Grow” si “Fall”. Atunci cand este introdus in scena principala Bățul se află in starea “Grow” iar atunci cand jucatorul nu mai apasa tasta SPACE aceasta intra in starea “Fall”.

Scriptul Grow.gd:

```
extends State


func _ready():
	conflicting_states=[]
	removing_states=[]
	necessary_states=[]

func get_transition():
	if !Input.is_action_pressed("stick"):
		return ["Fall","exit"]
	return null

func enter_state(old_states):
	pass

func exit_state(new_states):
	pass

func _during_state(delta):
	pr.scale.y+=pr.speed*delta
#
```

Scriptul Fall.gd:

```
extends State

#copy
func _ready():
	conflicting_states=[]
	removing_states=[]
	necessary_states=[]

func get_transition():
	if pr.rotation_degrees==90:
		return ["exit"]
	return null

func enter_state(old_states):
	pr.particles.emitting=false
	pass

func exit_state(new_states):
```

```
        pr.collision_shape.disabled=false
        pr.emit_signal("stick_fallen")

func _during_state(delta):
        pr.rotation_degrees+=pr.ang_speed*delta
        pr.rotation_degrees=min(pr.rotation_degrees,90)
```

Pe langa “eroul” acestui joc am realizat o harta generata procedural. Aceasta consta in niste stalpi pe care acesta poate sa mearga si niste marcare invizibile care au rolul.

BIBLIOGRAFIE

- https://docs.godotengine.org/en/stable/
- https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867
- https://www.aleksandrhovhannisyan.com/blog/finite-state-machine-fsm-tutorial-implementing-an-fsm-in-c/
- https://gamefromscratch.com/
- https://www.gdquest.com/

CUPRINS