Izrada brick breaker videoigre za mobilne uređaje u programskom alatu Godot

Kadežabek, Borna

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:074985

Rights / Prava: Attribution-ShareAlike 3.0 Unported/Imenovanje-Dijeli pod istim uvjetima 3.0

Download date / Datum preuzimanja: **2025-01-17**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Borna Kadežabek

Izrada brick breaker videoigre za mobilne uređaje u programskom alatu Godot

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
V A R A Ž D I N

Borna Kadežabek

Matični broj: 0016148410

Studij: Informacijski i poslovni sustavi

Izrada brick breaker videoigre za mobilne uređaje u programskom alatu Godot

ZAVRŠNI RAD

Mentor/Mentorica:

Izv. prof. dr. sc. Mario Konecki

Varaždin, rujan 2024.

Borna Kadežabek

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj rad se fokusira na kreiranje mobilne igre koristeći platformu Godot game engine. Godot je open source alat za razvoj igara, koji čini osnovu ovog završnog rada. Pored tehničkih aspekata razvoja igre, rad pokriva i dizajn vizualnog izgleda igre, kao i zvučne elemente koji su bitni za cjelokupno iskustvo igranja. U okviru rada, obrađuju se faze razvoja, uključujući programiranje mehanike igre, optimizaciju za mobilne uređaje, kao i testiranje i ispravljanje grešaka. Naglasak je stavljeni na korisničko iskustvo i intuitivnost korisničkog sučelja, što naravno doprinosi boljoj interakciji i iskustvu korisnika.

Ključne riječi: Godot; Video Igre; Mobilni Uređaji; UI; UX; Game Engine; Brick Breaker

Sadržaj

1. Uvod

Godot Engine je danas jedan od popularnijih opcija za kreiranje 2D ili 3D video igara. Postoji više različitih platformi, svaka sa svojim prednostima i nedostacima no Godot je u zadnje vrijeme stekao „kultno“ praćenje s obzirom na to da se ostale platforme plaćaju ili su vezane uz zadovoljstvo dioničara, a ne i interese programera. Moglo bi reći da se razvoj Godota zasniva na demokratskom „glasanju“ gdje korisnici, odnosno, programeri mogu zajedno usmjeriti projekt u određenom smjeru. Korisnici također mogu doprinijeti projektu preko GitHub platforme te se mogu javno izraziti o novim idejama, problemima i poteškoćama koje imaju s razvojnim okruženjem.

Kreiran 2014. godine, Godot je brzo stekao popularnost među programerima zbog svoje fleksibilnosti, jednostavnosti korištenja i snažne podrške zajednice. Ovaj alat omogućava razvoj igara za različite platforme, uključujući: Windows, macOS, Linux, Android, iOS pa čak i za web preglednike, čime postaje idealan izbor za kreiranje multiplatformskih igara i aplikacija. [1]

Jedna od ključnih karakteristika Godot Engina je njegov sistem scena. Umjesto tradicionalnog pristupa gdje je sve dio jedne velike hijerarhije, Godot koristi modularni pristup gdje je svaka scena samostalna jedinica koja može sadržavati i referencirati druge scene. Ovo omogućava programerima da lako upravljaju izuzetno kompleksnim projektima te ubrzavaju proces razvoja. Svaka scena može biti jednostavna kao jedna slika ili ponovno iskoristiva komponenta, pružajući veliku fleksibilnost u dizajnu. Iako Godot podržava nasljeđivanje, zbog same strukture i način programiranja, više se potiče kompozicija i proceduralno programiranje da se postigne modularnost i održivost projekta.

Unutar ovog rada proći ćemo kroz osnove Godota te proces kreiranja resursa za našu video igru s posebnim naglaskom na korisničko iskustvo.

2. Metode i tehnike rada

Pri izradi ovog rada primarno je korišteni alat Godot, koji je kao što je prije navedeno projekt otvorenog koda koji spada pod MIT licencu za slobodno korištenje, modifikaciju i distribuciju istog. Unutar rada se također koristi i besplatna platforma JSFXR za dizajn zvuka te i Aseprite za crtanje grafičkih elemenata igre.

Programski jezik GDScript, koji se koristi unutar ovog rada je posebno razvijeni za rad s Godotom.

3. Game Engine

Game engine je razvojno okruženje dizajnirano za razvoj video igra. Omogućuje programerima da efikasno kreiraju igre, bez potrebe za pisanjem svih osnovnih funkcionalnosti od nule. Tipične komponente engina uključuju renderiranje grafike, fiziku, detekciju kolizija, skriptiranje, upravljanje zvukom, animacije, umjetnu inteligenciju, mrežno povezivanje, izvoz aplikacija i slično. Sve te funkcionalnosti iziskuju veliko znanje i razumijevanje temeljnih značajki računalnih i informatičkih znanosti. Naravno, onda je znatno jednostavnije kreirati apstrakciju svih tih funkcionalnosti te ih zapakirati u ponovno iskoristivu aplikaciju.

Popularni game engine-i uključuju Unity, Unreal Engine, CryEngine za 3D razvoj te i Godot, Construct3 i GameMaker za razvoj 2D igara.

3.1. 2D vs. 3D

Grafika video igara može se svesti na 2 vrste: Dvodimenzionalne (2D) i trodimenzionalne (3D). Spacewar se smatra da je bila prva računalna igra koja je razvijena 1962. godine na MIT (eng. *Massachusetts Institute of Technology*), gdje su Stephen Russell, Martin Graetz, Peter Samso i ostali kreirali prvu igru. Spacewar je bila izuzetno jednostavna igra gdje su 2 igrača kontrolirala vlastitu letjelicu koja je u orbiti oko jednog planeta, igrači su onda mogli upravljati sa svojom letjelicom te gađati i srušiti protivnika, odnosno pobijediti. Atari 1972. godine razvija i objavljuje prvu komercijalno uspješnu računalnu igru Pong. [2]

O pojmu računalne igre: Ovaj izraz je u oštroj konkurenciji s video igrama, igrama za konzole i arkadnim igrama. Videoigre i igre na konzolama obično podrazumijevaju igre povezane s TV-om, dok arkadne igre podrazumijevaju igre postavljene na javnim mjestima. Računalne igre povremeno se podrazumijevaju kao igre koje se igraju na računalu. Budući da su se sva ova područja razvijala izuzetno blisko i paralelno (i budući da se sve te igre igraju na računalima), koristimo izraz računalna igra da označim sva ta područja u cjelini. [2]

2D igre su sve igre koje se nalaze unutar dvodimenzionalnog kartezijevog koordinatnog sustava označenim s (x, y) osima. Postoje više orijentacija koordinatnog sustava. Godot numerira, odnosno označuje svoj kao što se označuju slike na računalu, gdje je y os okrenuta prema dolje a lokacija $(0,0)$ se nalazi na lijevom vrhu, za razliku od ostalih alata koji prikazuju y os od dolje prema gore kao što su Blender, Autodesk Maya, Unreal Engine i slično. Sve igre su u početku bile 2D te su nakon nekog vremena pokušavale imitirati osjećaj trodimenzionalnosti koristeći različite sličice za različite orijentacije lika.

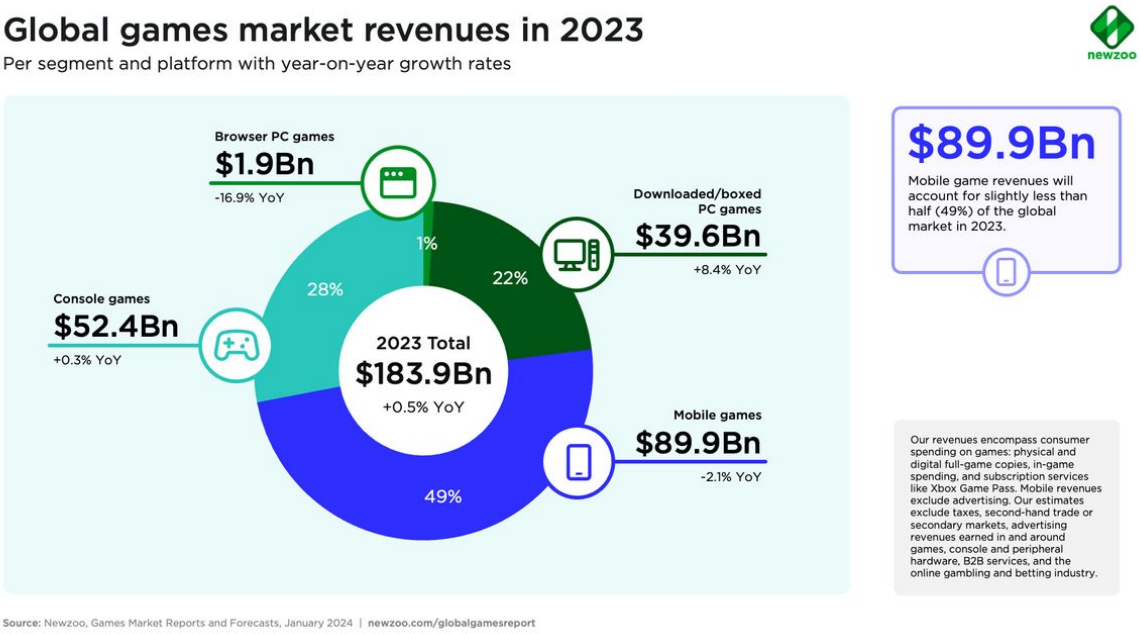
Dobro poznata implementacija imitacije 3D se koristilo u strateškim igrama (eng. *Real Time Strategy - RTS*), gdje su pojedini likovi mijenjali svoju orijentaciju preko različitih preddefiniranih sličica ovisno o njihovoj trenutnoj orijentaciji. Najčešće korisnik je gledao stanje igre iz ptičje perspektive. Prva igra koja se nalazila u 3D okruženju smatra se da je Maze War, iako se Maze War samo sastojao od kompasa i jasno označenih rubova poligona. Prva izuzetno poznata i komercijalno uspješna igra u 3D okruženju je bila Wolfenstein 3D kojega je naslijedio još poznatiji DOOM. Wolfenstein 3D i DOOM razvio je tim u tvrtci id Software. Jedino što je trodimenzionalno unutar igara je prostor, sve ostalo je imitacija 3D okruženja. Svi zidovi, igrač, neprijatelji i pokretni dijelovi su samo 2D paneli koji su spojeni tako da stvaraju iluziju 3D objekta. Svi paneli neprijatelja su orijentirani prema igraču po normali od igrača. Također pomicanje kamere kroz z os je bila onemogućena no kretanje kroz z os je bilo moguće.

Prva prava komercijalna 3D igra bila je “I, Robot”, arkadna igra koju je razvio Atari te je izdana 1984. godine. Bila je to prva arkadna video igra koja je koristila solidnu 3D renderiranu rasteriziranu grafiku s teksturiranim poligonima te ju je iskorištavala za crtanje pogleda u trećem licu. [3]

Unutar ovog rada primarno se fokusiramo na razvoj dvodimenzionalne igre. Specifično se fokusiramo na razvoj žanra Brick Breaker ili takozvani (eng. *breakout*) stil igre, gdje je u najjednostavnijem obliku cilj igre uništiti sve cigle ili blokove raspoređene na vrhu ekrana udaranjem loptice s pomoću pokretne platforme na dnu ekrana. Igrač kontrolira platformu koja služi za odbijanje i usmjeravanje loptice prema uredno posloženim objektima odnosno ciglama. Kada lopta pogodi objekt, objekt se uništava ili oštećuje.

3.2. Razvoj za mobilne uređaje

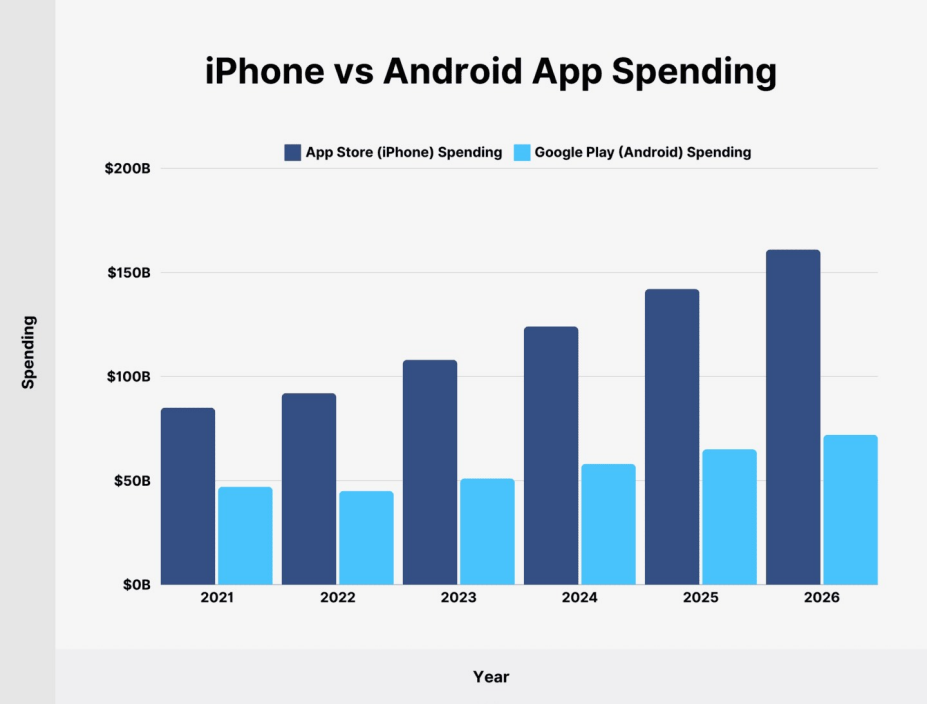
U digitalnom dobu, mobilni gaming postao je dominantni u industriji zabave. S milijardama korisnika diljem svijeta radi jednostavnosti korištenja i nabave uređaja za pristup, potražnja za zanimljivim i inovativnim mobilnim igrama danas je na vrhuncu. Razvoj igra za mobilne platforme višestruk je proces koji kombinira kreativnost i tehničko znanje.



Slika 1: Prikaz udjela prihoda od različitih segmenta gaming industrije u 2023. godini (Izvor: PCGamer) [4]

Za razvoj igara za mobilne uređaje možemo odabrati 2 platforme. To su Android platforma od Google i iOS platforma od Apple. Za ovaj rad radimo na iOS platformi koja ima znatno više poteškoća u odnosu na razvoj za Android, no ima i svoje dugoročne prednosti. Za Android je puno jednostavnije zbog toga što Android uređaji, uz striktno dopuštenje korisnika uređaja dopuštaju instalaciju stranih .apk (izvršna datoteka Android operacijskog sustava) datoteka. Za razliku od iOS koji u potpunosti zabranjuje instalaciju aplikacija koje nisu odobrene od strane Apple odnosno njihovog vlasničkog AppStore platforme koja služi za distribuciju iOS, iPadOS i macOS aplikacija na jednom mjestu. Za odobravanje aplikacije za testiranje potreban je mac uređaj s Xcode alatom čiji projekt je potrebno svakim testiranjem verificirati i validirati za korištenje. Pritom je brzo testiranje (eng. *Hot Reload*) gdje možemo uživo vidjeti promjene nad kodom nemoguće postići.

No iOS ima i svoje prednosti što se tiče jednostavnosti korištenja simulatora te potencijalno većih prihoda jer iOS korisnici imaju veću tendenciju platiti za aplikaciju ili platiti za nešto unutar same aplikacije kao što je prikazano na slici 2. [5]



Slika 2: Potrošnja korisnika na iOS i Androidu platformama (Izvor: Backlinko) [5]

Također valja nadodati da je znatno manje iOS korisnika u odnosu na Android, stoga je potrošnja prosječnog korisnika na iOS-u proporcionalno veća. Udio tržišta za Android je u 2024. godini bio 70.69% dok je udio tržišta za iOS bio samo 28.58%. [5]

Razvoj za android je isto tako izuzetno zahtjevan što se testiranja tiče jer naša aplikacija odnosno igra mora raditi na znatno više uređaja, a s obzirom na to je Android open source. Kod 2D igri je to posebno problematično jer omjer ekrana utječe na prikaz igre. Uz to, 2D igre ne rade na principu omjera pogleda gdje je svaki element udaljeni određeni postotak u odnosu na stranice ekrana ili drugog elementa, već na koordinatnom sustavu s pikselima. Razvijanje igre s različitim omjerima kao što su 19:9, 19.5:9, 20:9, 21:9. [6]

Svaki android uređaj ima i različitu rezoluciju ekrana koje mogu utjecati na performanse ili postići nenamjerni loše kvalitetni ili “pikselizirani” efekt znatno problematičnija. Generalno svi danas dostupni iPhone uređaji imaju 19.5:9 omjer. Manje popularni stariji uređaji imaju stari “tradicionalni” 16:9 omjer. [7]

4. Godot i GDScript

Kao što je prije bilo spomenuto, Godot je projekt otvorenog koda (eng. Open source) kojeg aktivno održava preko 2000 osoba te je tijekom ovog rada pronađeno puno bugova i pogrešaka unutar Godota. Sve greške i poteškoće su prijavljene te se trenutno radi na njima za daljnje verzije.

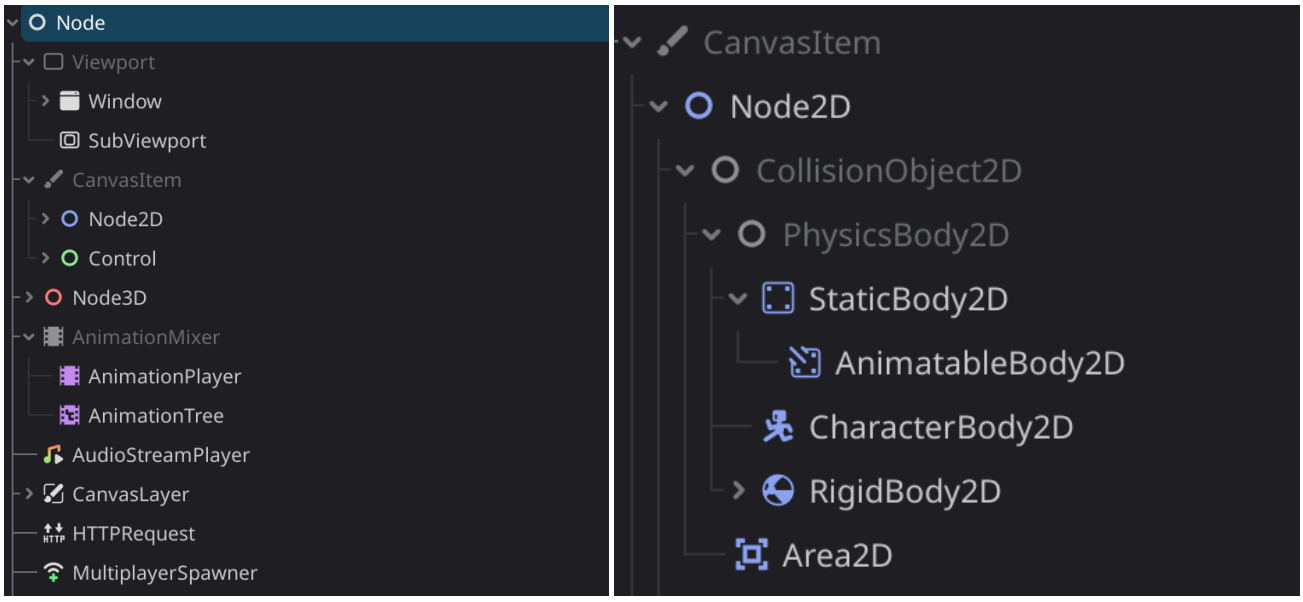
4.1. Osnovna komponenta Godota: Čvor

Najosnovnija komponenta svakog Godot projekta je čvor (eng. *Node*).

Čvor je temeljni građevni blok za logiku i kreiranje prikaza (eng. *View*) unutar Godota. Svaki čvor može biti instancirani pod nekim drugim čvorom i pritom imati pristup komunikaciji između roditelja i djeteta.

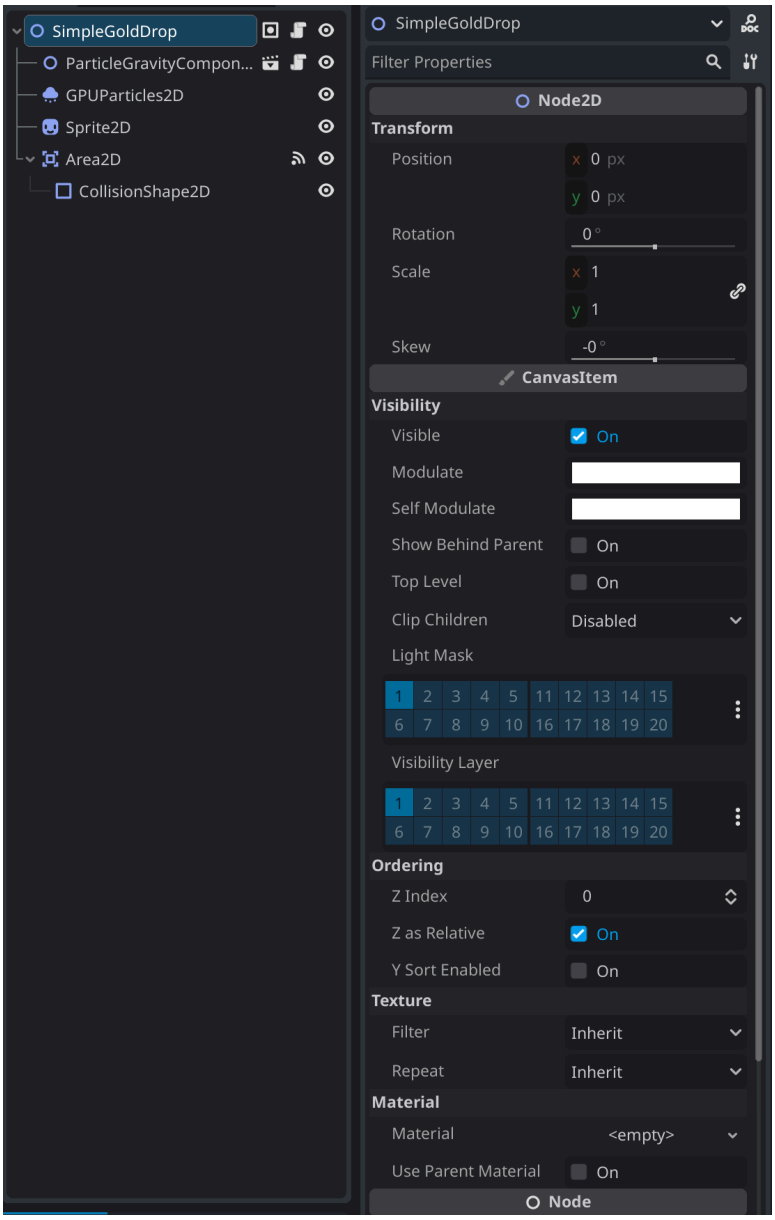
Svi čvorovi se moraju instancirati unutar scene. Postoji jedna glavna scena “root” unutar koje se nalazi naš cijeli projekt i svi njegovi trenutno aktivni pogledi. O scenama i njihovoj važnosti ćemo kasnije.

Svaki čvor ima svoju svrhu i prije implementiranu logiku. Čvor također ima i svoj tip. Tip čvora diktira njegovu svrhu. Tip čvora se ne može naknadno promijeniti, ali se može naslijediti.



Slika 3, 4: Prikaz nasljeđivanja svih čvorova unutar Godot-a

Vidimo kako se svi čvorovi unutar Godota nasljeđuju od glavnog čvora zvanog Node. To također možemo vidjeti i unutar inspektora za svaki pojedini kreirani čvor.



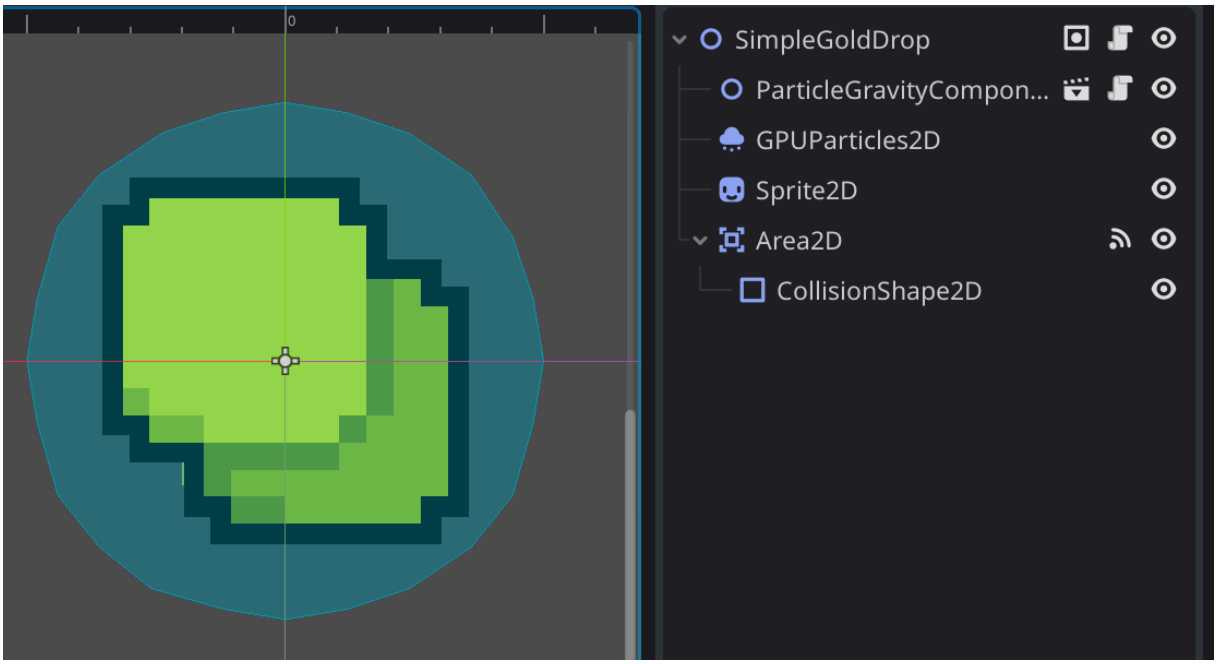
Slika 5: Prikaz inspektora odabranog čvora iz skupine čvorova

U gore navedenoj slici 5. vidimo kako svi čvorovi imaju unutar sebe definirani Node. Sve što se nalazi unutar Node su svojstva i atributi kojima manipuliramo bez pisanja koda. Time postižemo nekakvi pseudo low-code okruženje.

Umjesto se sva svojstva mjenjaju unutar koda, ona svojstva koja se najčešće koriste i mijenjaju mogu se promijeniti unutar inspektora. Inspektor sadrži osnovne i najčešće korištena svojstva svakog čvora. Kao što je prije rečeno, svaki čvor ima neka svoja svojstva i svrhu. Vidimo da u gore prikazanoj slici 5. imamo svojstva pod sekcijom Node2D i CanvasItem, te na kraju dolje Node. Time se od dolje prema gore (unutar inspektora) povezujemo s nasljeđivanjem svojstvima koja su prije bila spomenuta. Da u gore navedenoj slici koristimo Node3D umjesto Node2D, prikaz unutar inspektora ne bi sadržavao svojstva

CanvasItem, već samo Node i Node3D, jer samo čvorovi Node2D i Control nasljeđuju CanvasItem.

Kao što se vidi, svaka sekcija ima neka svoja svojstva koja onda manipuliraju svoje određeno područje. Vidimo da se Node2D primarno odnosi na poziciju i prikaz čvora, dok se CanvasItem odnosi na vidljivost i redoslijed čvora. U ovom slučaju, čvora tipa Node2D.



Slika 6: Prikaz osnovne scene unutar koje je prikazani odnos između čvorova

Unutar slike 6. se vidi kako je ime scene “SimpleGoldDrop”. Navedena scena sadrži više različitih čvorova od koje svaki ima svoju svrhu. Svaki čvor pritom doprinosi ukupnoj svrsi scene.

Svaki čvor unutar sebe može imati samo jednu asociranu skriptu. Ta skripta je onda logika tog čvora. Skripta unutar čvora uvijek nasljeđuje svojstva svog čvora i nemoguće je naslijediti više različitih svojstva. To se može zamisliti kao klase, gdje dijete klase može imati jednog i samo jednog roditelja, a roditelj može imati više djece.

4.2. Scene

Scene unutar Godota mogu se zamisliti kao ponovno iskoristivi skup više čvorova gdje zajedništvo čvorova rješava neki veći problem. Scene se također mogu zamisliti kao i ponovno iskoristive komponente koje se mogu instancirati u bilo kojem trenutku. Na primjer u slici 6. vidi se kako je svrha scene “SimpleGoldDrop” da padne zlatnik. Znatno je jednostavnije kreirati zlatnik kao komponentu, a ne ga ručno kreirati svaki put ili ga naslijediti

od entiteta koji sadržavaju nama možda nepotrebna svojstva i metode. Time se vidi prednost Godota u odnosu na neke ostale game engine koji očekuju da se grade na nasljeđivanju.

Svaka scena kao svoj root sadrži čvor. Isto kao i svaki čvor, scena ima svoj tip podataka kojega dobije od root čvora. Uvijek je najbolje za root scene koristiti tip čvora koji ima smisla za tu scenu radi jednostavnosti održavanja i čitljivosti koda.

4.2.1. Stablo scena

Stablo scena (eng. *SceenTree*) služi kao temelj za kreiranje i spremanje instanci scena. Stablo scena je skup instanciranih scena koje su izgrađene od čvorova i zasebnih nezavisnih čvorova.

Stablo scena je ukupni prikaz svih trenutnih scena i čvorova koji su vidljivi korisniku te se i aktivno koriste. Stablo scena je zamišljeno da se tijekom izvršavanja aplikacije može pretražiti, ispitati, dohvatiti i promijeniti.

4.2.1.1. Remote scena

Unutar Godota imamo više alata za debugiranje. Od tradicionalnog alata za praćenje stanja varijabli, koristi se breakpoint te (eng. *Performance profiler)* za praćenje trenutnih performansi aplikacije

Performance profiler nam služi za praćenje trenutnog stanja naše aplikacije. Od trenutnog okretanja sličica unutar jedne sekunde (eng. *Frames per second* - FPS), prikaza ukupnog broja objekta na ekranu, ukupni broj čvorova i scena koji se trenutno koriste te također, orphan čvorova koji nam ukazuje na probleme s uništavanjem scena tokom izvršavanja aplikacije.

Remote scena je jedan od alata koji je specifični za Godot, s obzirom da Godot koristi strukturu stabla za upravljanje čvorovima. To onda znači da možemo pratiti stanje trenutnog stabla scena. Stoga sse može reći da je remote scena stablo scena. Ovo je izuzetno važan i koristan alat za detekciju i rješavanje grešaka te za upućivanje na moguće pogreške koje se dešavaju unutar aplikacije bez da mi to možda i znamo, npr. kreiranje djece unutar krivog čvora i slično.

4.3. GDScript

GDScript je dinamički programski jezik koji se koristi za programiranje unutar Godota. Bazirani je na C++, ali je njegova sintaksa sličnija Pythonu. Dizajnirani je posebno za Godot. Može biti slabo il strogo tipizirani, ovisno o potrebama programera ili projekta.

Godot podržava i ostale jezike preko GDExtension-a. GDExtension omogućuje proširivanje trenutnih mogućnosti Engina, najčešće uz C++ s obzirom na to da je Godot i

pisani u C++. Također se omogućuje i pisanje GDScripta preko drugih jezika. Službeno podržane varijante su .NET i C++, ali je Godot zajednica proširila mogućnosti i na ostale programske jezike kao što su Rust, GO, Python, Swift i ostali. [8]

Godot je izuzetno jednostavan programski jezik koji ne očekuje veliko programersko predznanje te je izuzetno dobar za početnike. Prednost GDScript-a je što ne forsira koncepte nad korisnikom. Iako su klase ugrađene u GDScriptu, one se ne forsiraju kao u drugim jezicima kao što su Java i C#, svaki projekt je drugačiji. Naravno postoji određeni “korektni” i konkretni način kako se problemu pristupi te se ga riješi, najčešće uz upotrebu kompozicije radi dugoročne jednostavnosti, nadogradivosti i modularnosti projekta.

4.3.1. Osnove GDScript jezika

Skripta zasebno ne može postojati i pokretati se osim unutar određenih uvjeta. Ti uvjeti su Autoload Singletoni o kojima ćemo kasnije govoriti. Svaka scene bi trebala sadržavati root skriptu te scene na root čvoru scene, tako da je komunikacija između roditelja i djeteta te i među djecom kontrolirana kroz jednu centralnu jedinicu, u ovom slučaju root čvor scene.

Svaka skripta može samo naslijediti jednu skriptu uz pomoć ključne riječi “extends”. Extends nasljeđuje čvor za kojeg je ta skripta asocirana te je nemoguće asocirati skriptu drugačijeg “tipa” za određeni čvor. Isto tako nemoguće je naknadno promijeniti tip skripte nakon što je definirani čvor. Za naknadne promjene prije izvršavanja aplikacije, potrebno je promijeniti tipa podataka skripte i čvora na isti tip. Također je nemoguće da skripta naslijedi dijete određenog čvora. Na primjer ako imamo čvor tipa Node, nemoguće je da skripta naslijedi tip AudioStreamPlayer bez obzira na to što je AudioStreamPlayer dijete od tipa Node.

Osnovni način izvršavanja neke logike unutar GDScript je uz pomoć “func” (eng. *Function*) ključne riječi, func služi za kreiranja metoda unutar skripte koje se mogu pozvati samo unutar asociranog čvora. Moguće je pozvati metode i izvan zadanog čvora, ali za to treba instancirati scenu kojoj bi onda javili da želimo izvršiti metodu unutar ciljanog čvora ili preko referenci na čvor. Više o odnosima roditelja i djeteta ćemo kasnije.

```
func print_word(word: String) -> void:
	print(word)

func multiply(a: int, b: int) -> int:
	return a * b
```

Unutar GDScript ne postoje javne, privatne i zaštićene metode. Sve metode su javne ako su roditelj i dijete međusobno spojeni prije kompilacije ili tokom runtime uz get_node() metodu. Metoda get_node samo radi za djecu scene, a ako bi htjeli obuhvatiti stablo scena moramo koristiti get_tree() metodu.

4.3.2. _process()

Metoda _process(delta) je ključni dio za rad igre, bez nje ne bi bilo moguće kreirati pokrete niti obračune. Bez nje, sve radnje i događaji bi ovisili o brzini procesora, što nama nije u interesu. Da zaobiđemo problem ovisnosti koristi se _process metoda koja se osvježi i izvrši svake sličice, naravno to onda opet predstavlja isti problem gdje je opet brzina izvršavanja igre ovisna o procesoru, taj problem rješavamo s deltom.

Delta argument u _process metodi je rješenje za sve glatke i konstantne pokreta i promjene unutar svih igara te je i glavni dio _process metode. Delta se odnosi na delta vrijeme između sličica, predstavljajući vrijeme proteklo od posljednje prikazane sličice. Mjeri se u milisekundama, gdje je vrijednost najčešće jednaka 16 milisekundi za 60 FPS (eng. Frames per second) odnosno bolje poznato kao kratica FPS. Ovo je ključno za izračune koji su neovisni o broju sličica u sekundi, posebno za animacije, kretanja i druge promjene koje se temelje na vremenu igri.

Kada se radi s metodama _process(delta) i _physics_process(delta), delta se prosljeđuje kao argument. Metoda _process(delta) poziva se svake nove sličice te je stoga idealna za ažuriranja opće namjene poput animacija ili pokreta koji nisu povezani s fizikom npr. kao što su fluidne mehanike i slično.

Delta vrijednost osigurava da se igra dosljedno i glatko ponaša u različitim prikazima sličica unutar sekunde. Na primjer, ako želimo pomicati objekt konstantnom brzinom, trebamo uzeti u obzir vrijeme koje je prošlo između prijašnje sličice kako biste izbjegli varijacije u brzini kretanja. Jer ako objekte pomičemo po brzini uređaja odnosno FPS-u uređaja onda vrijeme koje je prošlo i lokacija na kojoj se objekt nalazi ne korespondira brzini. Ako FPS varira, varira i brzina igre na kojoj se igra izvršava. Zbog toga se određeni aspekti starijih video igra koje nisu bile zamišljene za izvršavanje na modernim računalima ponašaju “brže” nego bi trebali.

```
extends Sprite
var speed = 100 # Brzina u pikselima po sekundi

func _process(delta) -> void: # Pomičemo sličicu konstantnom brzinom
       position.x += speed * delta
```

Vidi se po gore navedenom kodu mi pomičemo poziciju trenutnog čvora konstantnim FPS neovisnom brzinom od 100 piksela po sekundi gdje nadodajemo trenutnoj poziciji brzinu speed koja se pomnoži između sličica delta mjerenu u milisekundama.

Delta je uvijek proslijeđena kroz _process metodu. Moguće ju je ignorirati ako se ne planira koristiti sa znakom ‘_’ ispred delta argumenta. U protivnom dobiva se upozorenje da se prosljeđuje delta koja se ne koristi što može naškoditi performansama. Svaki čvor unutar scene izvršava svoju sličicu procesa od gore prema dolje. [9]

4.3.3. _ready()

Metoda _ready se poziva samo jednom i nemoguće ju je ponovno pozvati. Ready metoda se ponaša kao konstruktor kod kojeg je nemoguće proslijediti argumente. Uvijek se poziva pri ulasku scene u stablo scena odnosno takozvani (eng. *Scene tree entry*). Konstruktori ne postoje unutar Godota jer preopterećenje metodi kao kod Jave nije podržano. Trenutno jedini način pripremanja stanja čvora odnosno scene pri kreiranju scene unutar stabla scena je ili preko _ready metode ili preko vlastitih metoda koje samostalno pozovemo pri kreiranju instance scene.

```
extends Node

func _ready() -> void: # Ready se jednom izvrši pri ulasku scene u stablo
    print("Scena je spremna")
```

Kad bi instancirali scenu uz gore navedeni kod, pri pokretanju odnosno kreiranju instance, kad bi se ona dodala u stablo scena unutar konzole bi se ispisalo “Scena je spremna”.

Postoje još alternativni načini za pripremanje scene za uporabu kao što su _enter_tree() i _init().

```
extends Node

func _enter_tree(): # Izvršava se drugi
    print("Test enter tree")

func _init(): # Izvršava se prvi
    print("Test init")

func _ready(): # Izvršava se zadnji
    print("Test ready")
```

U gore navedenom kodu vidimo kako se kod izvršava unutar određenog redoslijeda. Izvršava se tako da se prvo izvrši _init metoda koja se poziva kad se objekt kreira te se nalazi unutar radne memorije. Metoda _enter_tree se tek izvršava dok se scena/čvor doda na stablo scena, tu velimo scena/čvor jer _enter_tree metoda ne mari za djecom scene, samo stavi root scenu na stablo scena bez djece. To je korisno ako se zna da određena scena samo sadrži samo jedan čvor, gdje se onda ne mora čekati da se_ready izvrši, koji očekuje da posloži djecu scene, iako su takve situacije izuzetno rijetke. Metoda _ready se zadnja poziva kad se sva djeca (čvorovi) scene stave unutar stabla scene. Većinu vremena koristimo _ready radi sigurnosti da je sve što nam treba posloženo, jer gotovo uvijek root čvor scene očekuje nekakve podatke od djece i obratno. [9]

4.3.4. Instanciranje scene

Kao što smo rekli, glavni dio Godota je kreiranje, korištenje, manipuliranje i brisanje scena unutar stabla scena. Kreirati scene možemo ručno prije kompilacije koda gdje ručno

dodajemo svaku scenu/čvor ili scenu unutar scene. Također scene možemo dodati tijekom izvršavanja koda gdje koristimo više ključnih riječi i metoda: preload(), instantiate(), add_child().

```
extends Node

var moja_scene: Node2D = preload("res://putanja/do/moje/scene.tscn")

func _ready() -> void:
	var instanca_moje_scene: Node2D = moja_scene.instantiate()
	add_child(instanca_moje scene)
```

U gore navedenom kodu kreiramo varijablu tipa Node2D koja unutar sebe sadrži scenu gdje je root čvor tipa Node s asociranom skriptom istog tipa Node. Nakon toga kreiramo instancu moja_scena gdje je varijabla instanca_moje_scene tipa Node2D od varijable moja_scena koja se onda doda na root scene/čvora tipa Node.

Ako želimo dijete staviti unutar određenog čvora možemo koristiti ključnu riječ get_tree() gdje možemo ciljano kreirati dijete unutar određenog čvora. Često je praksa kreirati prazne čvorove bez ikakve funkcije i koristi te ih nazvati "Target" za jednostavniju manipulaciju svih smislenih čvorova koji idu u taj čvor. Npr. "BrickTarget" čvor će sadržavati instance scene Brick.

```
get_tree().root.get_node(
	"Putanja/Do/Cvora/cvor.tscn"
).add_child(instanca_moje_scene)
```

S obzirom na to da su čvorovi tipovi podataka, kreirati se mogu i čvorovi tijekom izvršavanje gdje ih onda dodamo unutar scene u kojoj su kreirani. Možemo manipulirati sva svojstva inspektora unutar koda, što je znatno manje elegantnije rješenje zbog toga što puno prostora koda potrošimo na postavljanje samog čvora, ali nam je ponekad korisno za brzo prototipiranje i testiranje.

```
extends Node2D

var sprite2d

func _ready():
	var sprite2d = Sprite2D.new() # Kreiramo novi čvor tipa Sprite2D
	add_child(sprite2d) # Dodamo u root čvor scene instancu sprite2D
```

4.3.4.1. Brisanje scena

Scene se brišu uz upotrebu ključne riječi queue_free() gdje se pozivanjem queue_free metode, briše iz stabla scena čvor ili scena unutar koje je pozvano, uključujući i djecu ciljanog čvora ili scene. Ako bi pozvali queue_free() unutar root čvora scene, onda bi obrisali cijelu scenu bez mogućnosti vraćanjem istog.

Metoda queue_free, iako korektno i temeljito očisti memoriju od ciljanog čvora, ima problem da briše reference izuzetno brzo. Čime imamo problem da ako se još izvršava neki

dio čvora unutar nekog zasebnog procesa ili dretve, postoji problem da kreiramo takozvane (eng. *Orphan*) dretve ili procese. Takvi događaji su izuzetno rijetki, ali se trebaju uračunati.

Najčešći problem s korištenjem queue_free su audio zapisi. Audio zapisi (zasebni čvor u ovom slučaju) bi se trebali izvršiti do kraja bez obzira na to što je njihov roditelj uništen. Naravno to se ne događa te je potrebno koristiti ostale dijelove stabla scena gdje na primjer kreiramo zvuk unutar trajnog memorijskog prostora koji je neovisni o dešavanjima s roditeljom. Ili uz pomoć signala (event) javimo koji zvuk bi se trebao izvršiti. Sve načine kojima pristupamo ovom problemu ćemo obraditi dalje unutar rada.

4.4. Bilješke

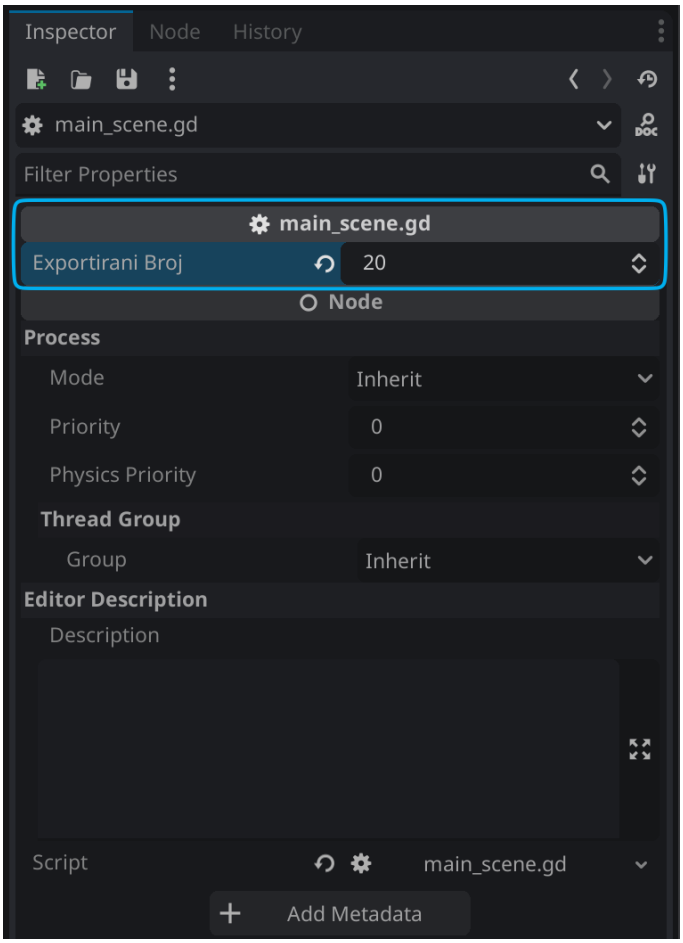
Bilješke (eng. *Annotations*) su posebni tokeni unutar GDScript koji se ponašaju kao ključne riječi, ali to nisu. Svaka bilješka započinje s ’@’ simbolom nakon kojeg slijedi vrsta bilješke te ime varijable unutar koje će se spremiti vrijednost bilješke.

Bilješke služe za konfiguriranje čvorova prije same kompilacije. Također služe za spajanje čvorova i scena koji nisu svjesne stanja scena stabla prije izvršavanja same aplikacije. Na primjer ako imamo više scena A1, A2, A3 koje su instancirane unutar neke druge scene B, gdje su A1, A2, A3 djeca B. Scene A1, A2, A3 ne znaju da su djeca B, ali mi znamo, stoga ih uz pomoć @export bilješke unutar skripte scene B ili bilo koje skripte scene A1...3 možemo referencirati A1...3 tako da znaju za međusobno postojanje.

Sve bilješke se prikazuju unutar inspektora gdje im onda možemo pridodati vrijednost ili referencu kao što je prikazano u daljnjem jednostavnom primjeru:

```
extends Node

@export_range(1, 100) var exportirani_broj: int

func _ready():
	print("Exportirani broj: ", exportirani_broj)
```

U gore navedenom kodu vidi se kako je varijabla ograničena unutar okvira od 1 do 100 gdje upisivanjem vrijednosti iznad ili ispod tog okvira rezultira mapiranjem te vrijednosti na najbliže ograničenje ograničenja definiranog unutar bilješke export_range.



Slika 7: Prikaz @export_range varijable unutar inspektora root čvora

Vidi se kako je unutar slike 7. prikazano eksportiranje varijable koja sadrži vrijednost 20. Upisivanjem vrijednosti koja je izvan okvira (1, 100) rezultira zaokruživanjem na 1 ili 100, koja god vrijednost je bliža.

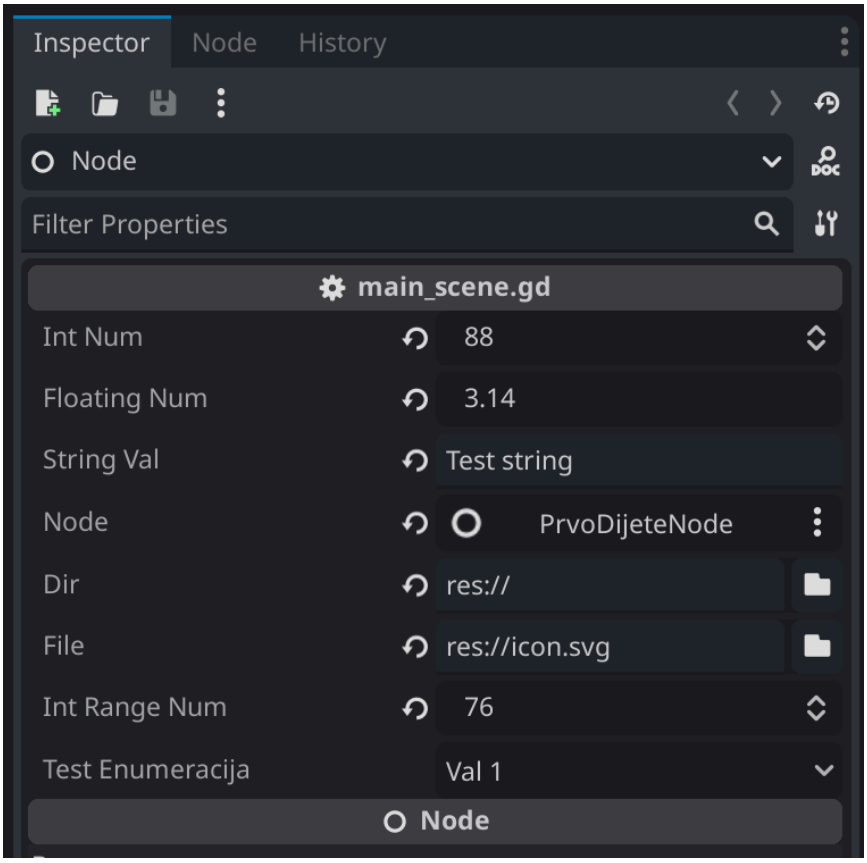
Postoje više vrsta bilješki. Postoje bilješke koje samo primaju jednostavne vrijednosti kao što su brojevi gdje koristimo @export_range gdje se može definirati prostor brojeva koje želimo odabrati

4.4.1. @export

@export bilješka je ključna bilješka koja služi za prikaz unosa unutar inspektora čvora. Sama vrijednost koju varijabla s @export bilješkom očekuje je derivirana od tipa te varijable. Na primjer: ako bi imali eksportiranu varijablu tipa Node, int, float, enum... Svaka bilješka forsira odnosno preuzima tip podataka od svoje uparene varijable koju predstavlja unutar inspektora. Zbog toga prikaz svake eksportirane varijable unutar inspektora izgleda drugačije da predstavlja tip podataka koji očekuje za unos za određenu eksportiranu varijablu. Ovaj primjer je prikazan u dolje navedenom kodu i slici 8.

```
extends Node

@export var int_num: int
@export var floating_num: float
@export var string_val: String
@export var node: Node

@export_dir var dir
@export_file var file
@export_range(1, 100) var int_range_num: int

enum TestEnum {
        val_1 = 1,
        val_2 = 2,
        val_3 = 3,
}

@export var test_enumeracija: TestEnum
```

Gore navedeni kod koji kreira više različitih vrsta @export bilješki, rezultira ovakvim izgledom inspektora:



Slika 8: Prikaz inspektora sa više različitih @export varijabli

Vidi se kako je unutar inspektora tip podataka prikazan drugačije, ovisno o tipu podataka. Isto tako, određene eksport varijable sadrže dodatne gumbove za interakciju s takvim tipom podatka, na primjer, kao što je enum_dir eksportna varijabla.

4.4.2. @onready

Kada koristimo čvorove, poželjno je želja zadržati reference na dijelove scene u varijabli. Kako je zajamčeno da se scene konfiguriraju samo prilikom ulaska u stablo aktivne scene, podčvorovi se mogu dobiti samo kada se izvrši poziv na Node._ready().

Ovo može biti malo naporno, pogotovo kada se čvorovi i vanjske reference gomilaju. Za to GDScript ima bilješku @onready koja odgađa inicijalizaciju varijable člana dok se ne pozove _ready(). [10]

@onready bilješka nam također može i služiti za referenciranje podčvorova scene tako da smo sigurni da imaju izvršenu _ready metodu. To nam dodatno olakšava posao jer spremamo referencu na čvor unutar varijable. Bez onready bilješke referenciranje podčvorova roota scene bi se izvršavalo uz simbol '$' gdje se brzo može kreirati gomila koda, jer se čvorovi imenuju opširno.

```
extends Node

@onready var ref_node = $ReferenciraniNodeScene

func _ready() -> void:
	# Rezultira istim tekstom
	print(ref_node)
	print($ReferenciraniNodeScene)
```

4.5. Signali

Signali su ključni dio i temelj rada s Godotom. Kao što je prije navedeno, Godot kao i ostali jezici i okruženja za kreiranje igara koriste MVC (eng. *Model View Controller*) čime postižemo obrazac promatrača (eng. *Observer pattern*). Obrazac promatrača se bazira na događajima koji onda aktiviraju neke predodređene funkcije koje se pokrenu kad se nešto desi, takav događaj zovemo "Event". [11]

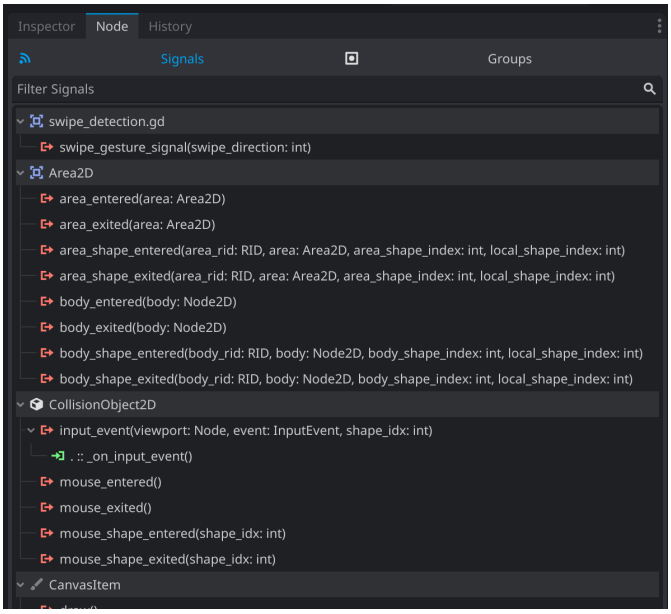
U najosnovnijem obliku, signali unutar Godota rade na principu pretplate pomoću connect(_on_ime_metode) i emit() metoda signal varijable. Po konvenciji funkciju koju pokrene Event signala započinju ime s "_on_". Signal jedino šalje obavijest i podatke ako postoje pretplatnici na taj signal (ponaša se kao mrežni preklopnik) gdje se obavijest šalje samo pretplaćenima, a ne svim čvorovima odnosno skriptama asociranog čvora koji prisluškuju eter te onda naknadno terminiraju zahtjev za obradu.

Signali i njihove interakcije moraju biti hardkodirane, iz toga proizlazi obavezni odnos roditelja i djeteta odnosno djeteta i roditelja unutar drva scena. Odnos roditelja i djeteta je moguće zaobići uz upotrebu Autoload o kojima govorimo u sljedećoj točki.

```
extends node

singal osnovni_signal
signal argumenti_signal(arg1: int)

var brojac: int = 0

func _ready() -> void:
	osnovni_signal.connect(_on_osnovni_signal)
	argumenti_signal.connect(_on_argumenti_signal)

	osnovni_signal.emit()
func _on_osnovni_signal() -> void:
	print("Osnovni signal je emitirani iz _ready metode")

func _on_argumenti_signal(brojac_arg: int) -> void:
	print("Signal s argumentom je emitirani. Brojilo je: ", brojac_arg)
	brojac = brojac + 1

func _process(_delta) -> void:
	argumenti_signal.emit(brojac)
```

Gore navedeni kod kreira 2 signala. Prvi signal je osnovni signal bez argumenta, dok drugi signal sadrži argumente kojima onda u funkciji povratnog poziva tog signala možemo pristupiti. Uz pomoć connect metode na tipu podataka signal, mi se spajamo s callback funkcijama: "_on_osnovni_signal" i "_on_argumenti_signal" da obrade poziv odašiljanja poziva za signal. To vidimo da smo uspostavili unutar _ready metode gdje smo svakom signalu priložili funkciju koja se izvrši pozivom određenog signala.

Neki node imaju ugrađene signale. Na primjer, node "Button" ima ugrađeni "_on_pressed" i "_on_hovered" signali koji se emitiraju unutar određenih uvjeta kad su oni zadovoljeni. Inspektor također sadrži naše signale koje smo mi kreirali.



Slika 9: Prikaz svih signala Area2D čvora unutar inspektora čvora

4.6. Autoload

Autoload nam služi za komunikaciju između više scena gdje nam nije baš najbolje koristiti komunikaciju između roditelja i djeteta. Autoload je singleton scena i uvijek se asocira odnosno kreira se sa skriptama i/ili scenama. Ako se autoload kreira sa skriptama, tokom runtime se kreira čvor tipa Node koja je asocirana sa danom Autoload skriptom. Novo kreirani čvor se nalazi kao prvo dijete root stabla scena.

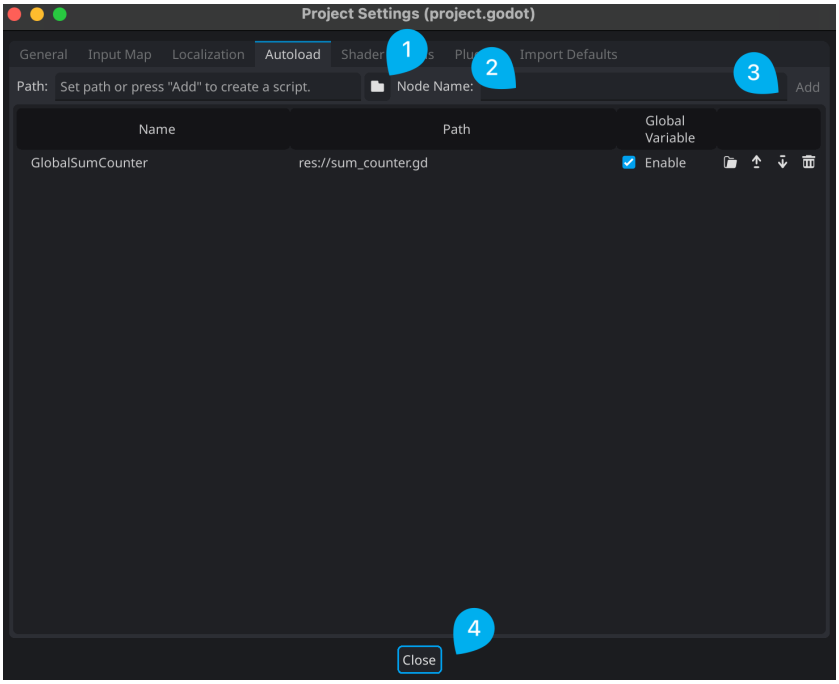
Glavni razlog za korištenja Autoload skripte je upravljanje svojstvima i metodama koje su javno dostupne svima. Import ne postoji unutar Godota te je onda Autoload singleton pod određenim imenom dostupan svim skriptama za korištenje. Autoload skripte najčešće nam služe za kreiranje svojstava koja se mogu dohvatiti, promijeniti i osvježiti iz bilo koje točke u bilo kojem vremenu unutar stabla scena u bilo koje trenutku. Godot Engine samostalno razrješava race conditions pri pristupu istom svojstvu u istom trenutku preko više različitih čvorova. [12]

Autoload kreiramo tako da se prvo kraira skriptu ili scena kojoj se planira pristupiti od bilo kojeg čvora. Najbolja praksa je kreirati i spremiti sve autoload skripte unutar jednog “Autoload” direktorija.

Kao jednostavni primjer koristit ćemo ovu jednostavnu skriptu koja samo vraća zbroj 2 dana broja. Skripta se zove sum_counter.gd:

```
extends Node

var global_string: String = “Test”

func sum_int_num(a: int, b: int) -> int:
        return a + b
```

Autoload se ručno dodaje unutar postavki projekta pod “Autoload” meni. Praksa je da se Autoload Singletonu pridoda nove ime klasa jer je Singleton, naravno obavezno ju je imenovati smisleno (alias), najčešće da ima sličnost s imenom svoje skripte iz koje se kreira. Isto tako dobra je praksa da se autoload uvijek diferencira od ostalih varijabli uz upotrebu camel case pisanja u odnosu na snake case koju GDScript forsira. Također kao dodatni način diferenciranja ponekad je korisno klasificirati autoload uz pomoć prefiksa ili sufiksa kao što je “Global”, “Manager”, “Converter” i slično, da se da naznake o cilju autoloada.



Slika 10: Prikaz postavljanja skripte za autoload unutar postavki projekta

Za pristup novo kreiranom autoloadu koristi se ime koje smo mu dali pod točkom (2) na slici 10. sad možemo pristupiti svim njegovim svojstvima i metodama. Kao primjer, kreiranom Autoloadu se pristupa kroz _ready metodu unutar glavne scene radi jednostavnosti. Pristup autoloadu radimo ovako:

```
extends Node

var global_var_singleton: int = GlobalSumCounter.sum_int_num(4,2)

func _ready():
	print(GlobalSumCounter.global_string) # Rezultira: "Test"
	print(GlobalSumCounter.sum_int_num(2, 3)) # Rezultira: 5
	print(global_var_singleton) # Rezultira: 6
```

Unutar gore navedenog koda može se vidjeti kako se GlobalSumCounter odnosno sum_counter skripta samo pozove bez inicijalizacije da postoji. Ne mora se koristiti ključnu riječ "new" da se dobi pristup svojstvima i metodama autoloada sum_counter.gd čiji je alias za pristup GlobalSumCounter.

Autoload je odlični za upravljanje svim dijelovima igre kojima treba imati konstantni pristup i znati da su unutar njih spremljeni korektni podaci. Na primjer ako imamo postavke igre i želimo znati koje su postavke spremljene. Kreiramo autoload koji ih pročita i spremi u radnu memoriju, tako da ne moramo konstantno slati upite na trajnu memoriju koje je stanje postavke.

Primjer jednostavnog autoloada koji se koristi unutar igre za ovaj završni rad je "LargeNumberConverter". Unutar igre postoji mogućnost da se brojevi bodova počinju

prebrzo pribrajati. Jedno rješenje za taj problem je umanjiti ili usporiti tempo dobivanja bodova, što umanjuje iskustvo igre. Najbolje je bilo da se brojke ne tretiraju kao brojevi već kao string, gdje bi broj 1 000 pretvorio u 1k, broj 1 200 000, pretvorio u 1.2M gdje je igraču jasno koliko bodova ima bez da se mu veliki dio ekrana zauzme s vrijednostima koje su nepotrebne.

```
extends Node

# Dostupno na: https://godotforums.org/d/35589-how-can-i-abreaviate-large-
# numbers
# Autor: turtleguy955
# 23.06.2024

func convert_num_to_typed_string(value: Variant) -> String:
	match typeof(value):
		TYPE_INT:
			if value >= 1000000:
				return "%d.%dM" % [value/1000000, (value/
100000)%10]
			elif value >= 1000:
				return "%dK" % [value/1000]
		TYPE_FLOAT:
			if value >= 1000000.0:
				return str(round(value / 1000000.0)) + "M"
			elif value >= 1000.0:
				return str(round(value / 1000.0)) + "K"
		_:
			return str(value)
	return str(value)
```

Kod je pribavljeni od Godot foruma, te je znatno modificirani da može poprimiti vrijednosti tipa float i tipa integer iz iste metode. Iako nam float nije potrebni, ovaj Autoload ima i dodatak da može poprimiti vrijednosti tipa float u slučaju da nam zatreba float tip podataka da pretvorimo u čitljivi string. [13]

Tip “Variant” unutar Godota služi nam za to kad se ne zna kojeg sve tipa podaci mogu biti ili kad podatak može biti korektni kao više različitih tipova podataka. Podatak može biti tipa int ili tipa string. Najbolja je praksa nakon toga napraviti switch te provjeriti kojega tipa je ulazni argument metode. GDScript nema switch ključnu riječ, ali zato ima match koji radi isto, Match nema break jer se automatski podrazumijeva na kraju svakog slučaja unutar match izjave kao što je prikazano u gore navedenom kodu.

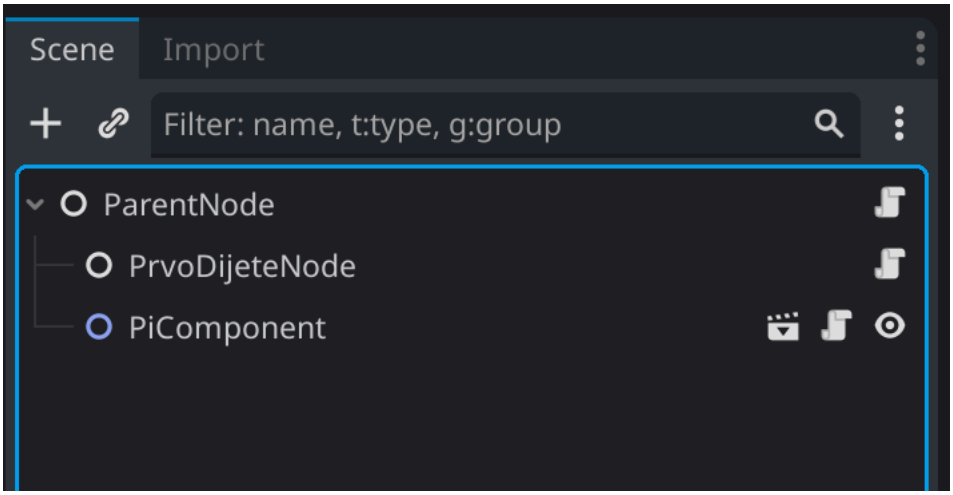
4.7. Kreiranje vlastitih komponenti

Unutar Godota, često je praksa kreirati vlastite ponovno iskoristive komponente. Uz pomoć @export bilješki je to izuzetno jednostavno. Kao što je prije bilo rečeno, scene se mogu kreirati unutar ostalih scena. Isto tako bilješke unutar jedne scene mogu biti određenog tipa podataka koji jednaki tipu podataka čvora, što znači da komponente možemo spajati međusobno iako možda nisu u direktnom odnosu kao roditelj-dijete.

Vlastite komponente se ne moraju samo spajati prije izvršavanja aplikacije. Svaka scena je u nekom smislu ponovno iskoristiva komponenta, ali ako mi zamislimo scenu kao komponentu koja je od samog početka zamišljena s glavnim ciljem modularnosti, mi možemo kreirati scenu koja može raditi u jedinstvu s ostalim scenama/čvorovima ili komponentama.

4.7.1. Prikaz jednostavne komponente

Za jednostavni primjer korištenja komponenti može se kreirati obična scena sa root tipa Node imena “ParentNode”. Kreiramo 2 djece: Prvo dijete je obični čvor koji ima asociranu skriptu samo za sebe zvan “PrvoDijeteNode”, a drugo dijete je naša nova komponenta “PiComponent” tipa Node2D. Komponenta je u ovom slučaju scena koja ima neku svrhu. Radi jednostavnosti, svrha komponente je dobaviti vrijednost π ostalim komponentama.



Slika 11: Prikaz glavne scene sa komponentom za dohvaćanje vrijednosti pi

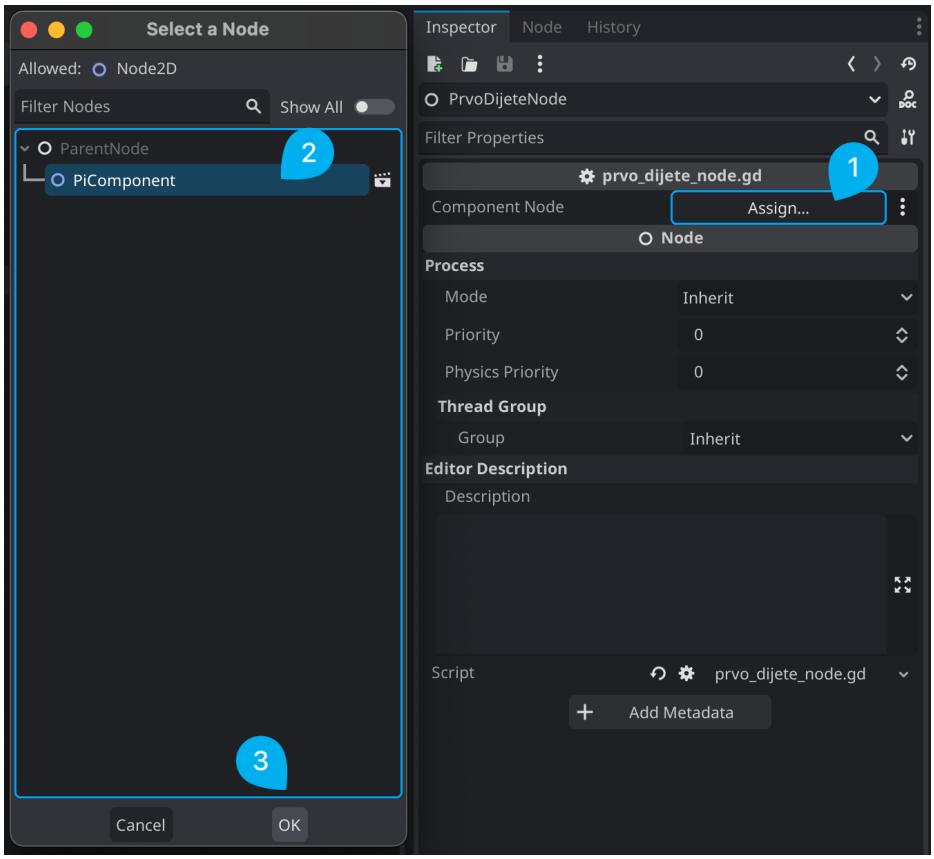
U gore navedenoj slici, prikazana je struktura glavne scene. Vidimo kako su nam tipovi čvorova prikazani različitim bojama. Također vidimo da svi čvorovi scene imaju simbol skripte, ali samo Node2D ima simbol oka zbog toga što čvorovi tipa node nisu zamišljeni za prikaz, već su samo tu za gradnju ostalih čvorova gdje je čvor tipa Node osnova svega. Isto tako samo naša komponenta PiComponent ima dodatnu posebnu ikonu (eng. *Clapperboard*) unutar scene koja označava da je taj čvor zapravo scena koja unutar sebe sadrži djecu koja nisu prikazana unutar scene u kojoj je naša komponenta (scena) instancirana.

Kod od komponente za dohvaćanje vrijednosti π izgleda ovako:

```
# PiComponent.tscn
extends Node2D

const pi: float = 3.14
```

```
func _ready():
	print("Ispis pi iz komponente: ", pi, ". Unutar _ready komponente")

func get_pi() -> float:
	return pi
```

Da bi mogli pristupiti komponenti preko svih ostalih čvorova potrebno ju je asocirati. To se može na više načina, ali je to najčešće uz upotrebu bilješke @onready i @export. U ovom slučaju to je obavljeno uz @export bilješku.



Slika 12: Prikaz asocijacije komponente preko export bilješke

Ovu asocijaciju na komponentu ćemo napraviti unutar ParentNode i PrvoDijeteNode čvorova. Da demonstriramo način rada s upitima od više čvorova.

Kod od root čvora scene odnosno ParentNode izgleda ovako:

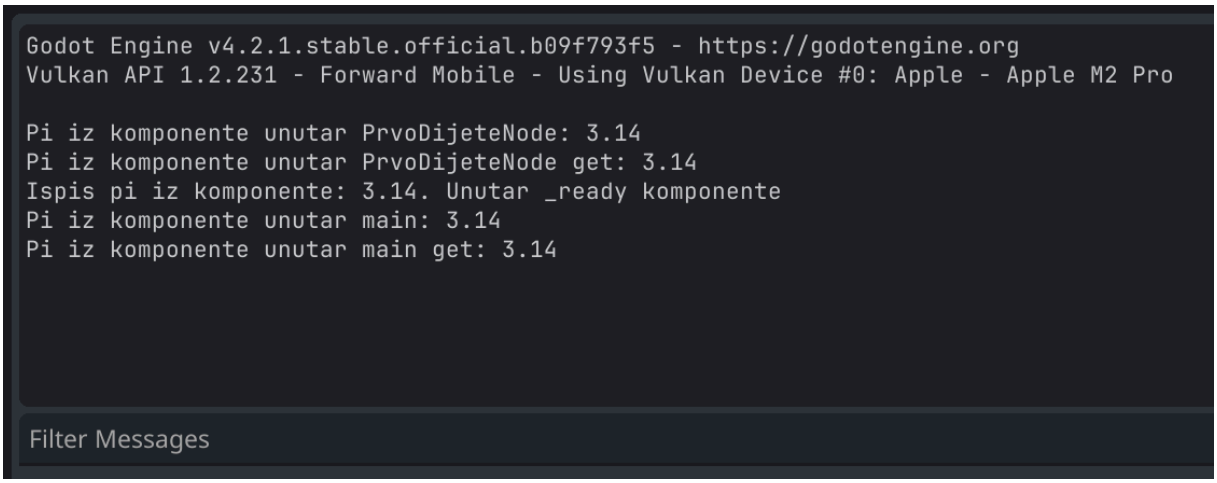
```
# ParentNode.tscn
extends Node

@export var component_node: Node2D

func _ready():
	print("Pi iz komponente unutar main: ", component_node.pi)
	print(
		"Pi iz komponente unutar main get: ",
		component_node.get_pi()
	)
```

Kod od “PrvoDijeteNode” čvora izgleda ovako:

```
extends Node

@export var component_node: Node2D

func _ready():
	print("Pi iz komponente unutar PrvoDijeteNode: ",
		component_node.pi)
	print("Pi iz komponente unutar PrvoDijeteNode: ",
		component_node.get_pi())
```

Vidimo kako je kod od čvora PrvoDijeteNode i ParentNode (root glavne scene) identični, te služi samo za pokazivanje da se djeca isto mogu međusobno asocirati isto kao i međuodnos roditelja i djeteta. Inače bi sva komunikacija morala ići kroz roditelja što nije uvijek najidealnije. Vidimo kako i kod djece postoji @export bilješka strogog tipa Node2D koja očekuje jedino čvora tipa Node2D.

Sada imamo pristup vrijednosti π direktno preko svojstva ili preko get metode koja nam vraća vrijednost π.



Slika 13: Prikaz ispisa unutar konzole od gore navedenog koda

Unutar slike 13. vidi se kako je ispis π isti, bez obzira ako se pristupa svojstvu direktno ili preko get metode. Naravno poželjnije je takve operacije raditi preko getter i setter metoda za postizanje čistog koda. Vidimo kako se PrvoDijeteNode prvo izvršilo, iako nije root scene.

Redoslijed izvršavanja čvorova je uvijek da se prvo izvrše djeca, redoslijedom od gore prema dolje. Dok se sva djeca izvrše, tek se onda roditelj počne izvršavati. To je drugačije u odnosu na redoslijed crtanja i z indeksa. Redoslijed crtanja radi na istom principu, ali zbog tog redoslijeda, čvorovi koji su dalje u nizu izvršavanja budu prvi prezentirani, odnosno na vrhu.

Slika 14: Prikaz redosljeda izvršavanja skripte čvorova

4.7.2. Prikaz naprednije komponente

Napredniji primjer ponovno iskoristive komponente koji koristimo unutar ovog rada je RandomSFX2DComponent tipa AudioStreamPlayer2D. Komponenta služi za sviranje pseudo slučajnog zvuka ovisno o zvukovima koji se postave unutar @export bilješki: audio_streams, death_streams. Da bi se razriješili monotonosti sviranja istog zvuka mi koristimo polje tipa AudioStream unutar audio_streams i death_streams da se ne svira isti zvuk više puta. Isto nad svakim zvukom se primjeni pitch modulacija koja nam je dostupna kao svojstvo unutar AudioStream čvora preko pitch_scale svojstva.

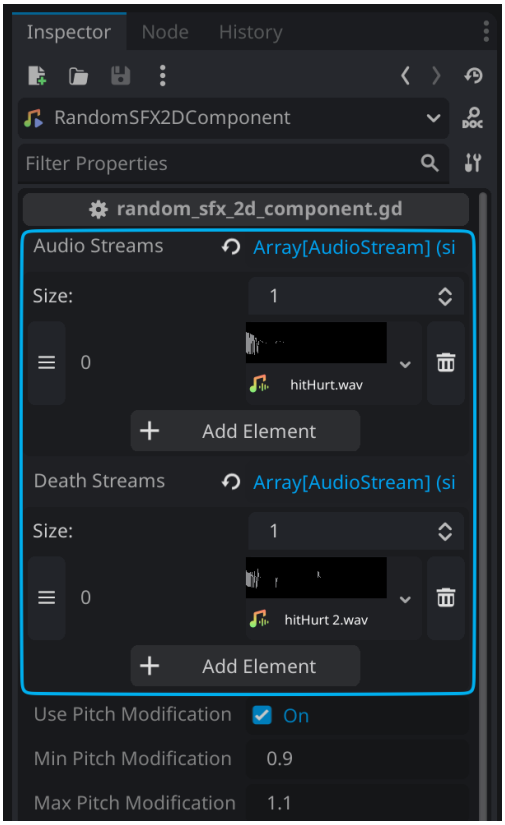
```
extends AudioStreamPlayer2D

# Firebelly Games 19.05.24. -  57. Adding SFX - Part 1 [34]
# Dostupno na: https://www.udemy.com/course/create-a-complete-2d-arena-
# survival-roguelike-game-in-godot-4/?couponCode=ST3MT72524

@export var audio_streams: Array[AudioStream]
@export var death_streams: Array[AudioStream]
@export var use_pitch_modification: bool = true
@export var min_pitch_modification: float = 0.90
@export var max_pitch_modification: float = 1.10

func play_random_stream() -> void:
	if audio_streams == null || audio_streams.size() == 0: return
	stream = get_random_audio_stream()
	set_random_pitch_modification()

	play()

func play_random_death_stream() -> void:
	if death_streams == null || death_streams.size() == 0: return
	stream = get_random_death_stream()
	set_random_pitch_modification()

	play()

func set_random_pitch_modification() -> void:
	if use_pitch_modification == true:
		pitch_scale = randf_range(
			min_pitch_modification,
```

```
                        max_pitch_modification
                )
        else: pitch_scale = 1

# Za GlobalSFXManager. Za queue_free instanci da i dalje reproducira zvuk.
func get_pitch_modification() -> float:
        if use_pitch_modification == true:
                return randf_range(
                        min_pitch_modification,
                        max_pitch_modification
                )
        else: return 1.0

func get_random_audio_stream() -> AudioStream:
        if audio_streams == null || audio_streams.size() == 0: return
        return audio_streams.pick_random()

func get_random_death_stream() -> AudioStream:
        if death_streams == null || death_streams.size() == 0: return
        return death_streams.pick_random()
```

Kod je bio znatno modificiran u odnosu na originalnu verziju tako da paše našemu slučaju korištenja.

Gore navedeni kod, radi na principu da kreiramo RandomSFX2DComponent komponentu čiji je čvor tipa AudioStreamPlayer2D, pritom dobijemo pristup @export varijablama komponente da ih možemo modificirati za slučaj korištenja te komponente u nekom određenom okruženju. Svaka metoda sadrži neku funkcionalnost koja odgovara toj komponenti. Na primjer, play_random_stream odsvira slučajno odabrani zvuk iz polja audio_streams za neki dani slučaj. Tu logiku moramo spojiti unutar scene u kojoj se komponenta nalazi te i postaviti reference na zvukove koje želimo svirati kad se neki događaj ostvari.

Kao jednostavni primjer, komponenta za reproduciranje zvuka može se postaviti da odsvira određeni zvuk kad se određeni objekt ošteti i zasebni zvuk kad se objekt uništi.



Slika 15: Postavljanje zvučnih efekata za Brick scenu preko RandomSFX2DComponent

Za izvršavanje zvučnog efekta pridodanog unutar @export audio_streams bilješke, potrebno je prvo referencirati našu audio komponentu preko @onready bilješke da ju možemo koristiti unutar Brick scene.

```
@onready var random_sfx_2d_component = $RandomSFX2DComponent
```

Sada kada nam zatreba, možemo reproducirati zvučne zapise koji se nalaze unutar @export bilješki unutar preko referenciranja random_sfx_2d_component varijable. Navedena varijabla nam je u ovom slučaju referenca na našu komponentu koja je scena, koja se nalazi unutar trenutne Brick scene.

Jednostavni primjer korištenja komponente gdje u ovom slučaju odsviramo kratak zvučni efekt kad se ošteti Brick objekt:

```
func damage_brick_and_check_hp(body) -> void:
	brick_data.damage(1)

	if brick_data.get_health() <= 0:
		GlobalSFXManager.play_queue_free_sfx(
			random_sfx_2d_component.get_random_death_stream(),
			random_sfx_2d_component.get_pitch_modification()
		)
		queue_free()
	else:
		random_sfx_2d_component.play_random_stream()

	check_brick_hp()
```

Prvo se fokusiramo na else blok koda koji se izvrši kad Brick ima više od 0 poena zdravlja. Poeni se nalaze unutar prilagođenog resursa o kojima se bude više govorilo unutar sljedeće točke. Reprodukciju audio zapisa radi se preko play_random_stream metode kojoj imamo pristup preko čvora komponente za upravljanjem audio zapisima. Zvuk se odsvira samo ako instanca Brick ima više od 0 poena zdravlja.

Veći problem je odsvirati zvuk dok moramo izbrisati scenu ako su poeni zdravlja jednaki 0 ili ispod. Problem je isti kao što je bio i navedeni unutar točke 4.3.4.1. gdje je problem dokraja odsvirati zvučni zapis čiji roditelj više ne postoji. Rješenje tomu je korištenje autoload Singletona zvan “GlobalSFXManager” scenu tipa AudioStreamPlayer2D koja preuzme izvršavanja ciljanog zvučnog zapisa na sebe. S obzirom na to da se nalazi izvan memorije u kojoj se nalazi čvor koji je predodređeni za brisanje, zvučni zapis unutar Autoloada nije pogođeni s destruktivnim problemom queue_free metode jer se autoload nalazi izvan opsega brisanja ciljane scene ili čvora.

```
GlobalSFXManager.play_queue_free_sfx(
        random_sfx_2d_component.get_random_death_stream(),
        random_sfx_2d_component.get_pitch_modification()
)
```

Gore navedeni kod prosljeđuje metodi play_queue_free_sfx autoload Singletonu GlobalSFXManager vrijednosti trenutnog zvuka koji se želi izvršiti pomoću get metode unutar komponente, te i pitch modulacija radi konzistentnosti. Jednostavni autoload je strukturirani ovako:

```
extends AudioStreamPlayer2D

func play_queue_free_sfx(
        sound_stream: AudioStream,
        pitch_mod: float
) -> void:
        stream = sound_stream
        pitch_scale = pitch_mod

        play()
        pitch_scale = 1.0
```

Jednostavna metoda prima zvučni zapis tipa AudioStream te pitch modulaciju tipa float te ih odsvira unutar čvora tipa AudioStreamPlayer2D s pomoću metode play(). Svojstvo stream je isto svojstvo AudioStreamPlayer2D koje postavlja određeni zvučni zapis za reproduciranje preko play() metode.

Svojstvo pitch_scale koje se odnosi na pitch modulaciju obavezno se mora postaviti natrag na vrijednost 1.0 jer bi se inače pitch_scale svojstvo zauvijek pribrajalo samome sebi te bi pitch modulacija rasla u beskonačnost. To je zbog toga što je ovo autoload koji se nikada ne briše iz memorije, odnosno uvijek ostaje u stablu scena, stoga moramo paziti na takve situacije.

4.8. Animacije

Animacije su ključni dio svake igre zbog dojma koji daju igraču kroz takozvano korisničko iskustvo (eng. *User Experience* - UX). Animacije su stoga ključni dio Godota i GDScript jezika čijim korištenjem proširujemo iskustvo igrača.

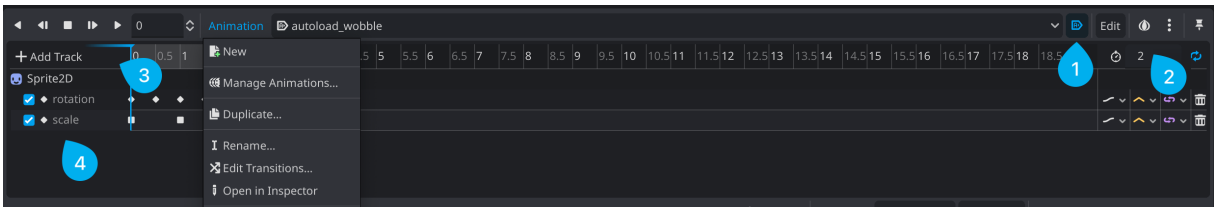
Unutar Godota, animacije se mogu kreirati na 2 načina: AnimationPlayer čvora i Tweenova. AnimationPlayer je čvor unutar scene koji se koristi uz upotrebu korisničkog sučelja (eng. *Graphical user interface* - GUI), dok su animacije uz upotrebu Tweena kreirane preko koda unutar čvora bez korisničkog sučelja.

4.8.1.AnimationPlayer

Animation Player čvor nam primarno služi za kreiranje linearne interpolacije svih svojstava svakog čvora zasebno spremljenog unutar određene animacije. Svaki čvor ima svoja svojstva od kojih je građeni. Pristupiti im preko @onready bilješke nam nije potrebno jer se AnimationPlayer čvor referencira na početku kreiranja root čvora određene scene. Referenciranje svakog svojstva čvora se radi uz pomoć pritiska na ključ unutar inspektora. Da bi nam se pokazali ključevi trebali bi biti unutar Animation tab koji se uvijek nalazi pri dnu ekrana.

Kao što je prije bilo navedeno, moguće je kreiranje više animacija i manipulacija istog unutar istog čvora. Svaka animacija ponovno kreira sve svoje ključeve za svaki čvor unutar sve djece scene, uključujući i root čvor scene, bez obzira na to ako se nalazi unutar nekog root čvora scene. Nije preporučeno manipulirati roditelja (root scene u ovom slučaju) od AnimationPlayer čvora.

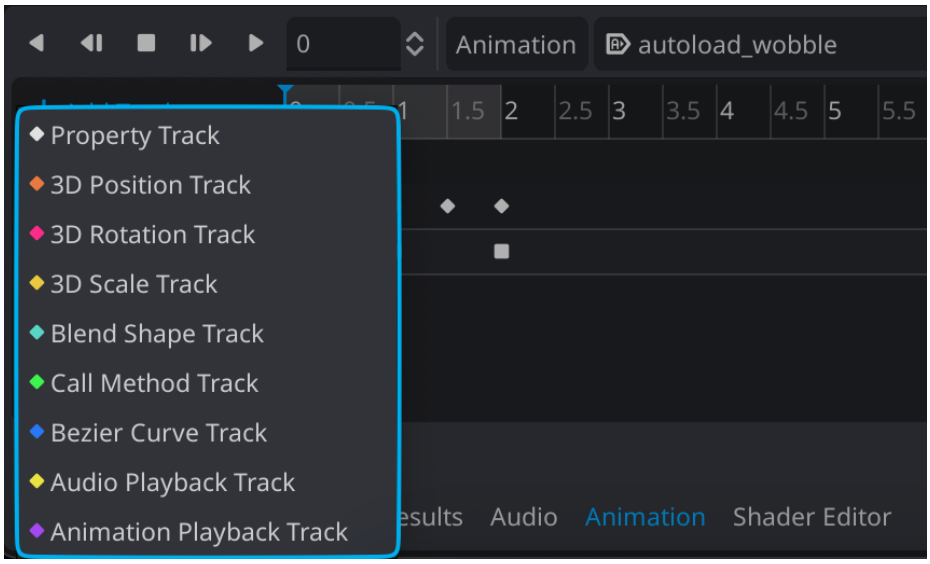
Klikom na AnimationPlayer otvara nam se novi tab prozor na dnu ekrana unutar kojeg možemo kreirati novu animaciju te joj pridodati ime.



Slika 16: Prikaz AnimationPlayer prozora

Unutar čvora AnimationPlayer vidimo označeno na slici 16. pod oznakom (1) gumb za “Autoplay on load”, što znači da će nam animacija započeti reproducirati prilikom učitavanja root čvora (scene) u stablo scena. Pod oznakom (2) nam je označeno ukupno trajanje animacije te i gumb za ponavljanje animacije. Pod oznakom (3) nam je gumb za kreiranje novih traka za animacije. Sve trake koje se mogu kreirati su prikazane na slici 17. Pod

oznakom (4) na slici 16. prikazane su sve trake unutar trenutne animacije. Također vidimo kako je kreirani property track koji cilja Sprite2D čvor unutar scene u kojoj se nalazimo. Također vidimo da unutar property track ciljamo ključeve svojstava: “rotation” i “scale” koje onda uz ključne točke kroz određeno vrijeme mijenjamo. Ova je animacija za “ljuljanje” cijelog objekta da se čini prirodnijim tijekom igranja. Stoga nam je potrebno da se animacija izvršava beskonačno uz pomoć oznake (2) na slici 16.



Slika 17: Prikaz svih mogućih tipova traka unutar AnimationPlayer čvora

Također se mogu kreirati i trake tipa “Call Method Track” koje mogu pozvati bilo koje metode od ciljanog čvora, u ovom slučaju to je čvor tipa Sprite2D ili bilo kojeg drugoga ako odaberemo unutar trenutne scene u kojoj se nalazi AnimationPlayer čvor. Ne samo da se mogu pozvati metode koje su ugrađene unutar tog čvora, već se mogu pozvati prilagođene metode za upravljanje animacijom ili pripremanje određenih dijelova scena na nama prilagođeni način.

Točke unutar trake za animacije dodaju se preko desnog klika na traku te onda “insert key-frame”. Dodavanjem nove točke pod inspektorom za tu odabranu novokreiranu točku dobijemo mogućnost mijenjanja vrijednosti za taj tip podataka. Ako je svojstvo rotacija onda je tip podataka u stupnjevima, a ako je tip podataka pozicija onda je tip podataka koji unosimo u točku tipa Vector2.

4.8.2.Tween

Za razliku od AnimationPlayer čvora. Tween služi za linearnu interpolaciju vrijednosti koje ne znamo unaprijed. Iako je moguće mijenjati vrijednosti kroz neko vrijeme unutar AnimationPlayer čvora čije vrijednosti još ne znamo unaprijed, vrlo brzo nam postaje previše i prekompleksno za održavanje. Tu uskače u pomoć Tween.

Tweenovi se uvijek izvršavaju redoslijedno jedan za drugim bez preskakanja osim u određenim uvjetima gdje je dopušteno paralelno izvršavanje više Tweenova. [14]

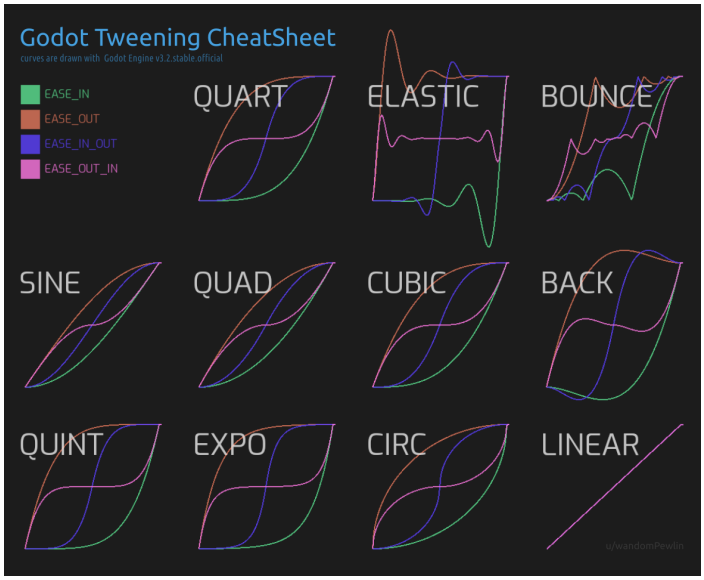
U prijašnjim verzijama Godota, Tween je bio zasebni čvor unutar scene, slično kao i AnimationPlayer čvor. U novijim verzijama se može kreirati samo preko koda. Tween je dobio ime od “in-betweening” što je animacijska tehnika za kreiranje specifičnih točki unutar određenog vremena gdje onda računalo interpolira vrijednosti kroz put između tih 2 točki. [14]

Za kreiranje Tweena koristimo metodu create_tween koju pridodamo varijabli koja onda postaje tip podataka Tween.

```
func _ready() -> void:
	var tween = create_tween()

	tween.tween_property(
		self, “global_position”, Vector2(120, 90), 1.0
	)
	tween.tween_property(
		self, “rotation”, 90, 1.0
	)
```

Unutar gore navedenog koda mi “Tweenamo” trenutnu globalnu poziciju root-a scene od njezine trenutne globalne pozicije (x, y) do globalne pozicije tipa podataka Vektor2 u vrijednosti od (120, 90) kroz 1 sekundu vremena što je postavljeno na zadnjem argumentu tween_property metode. Nakon što se to izvrši kroz točno 1 sekundu vremena će se rotacija root scene okrenuti za 90 stupnjeva.

Da se dobi prirodniji osjećaj kretanja tweenovi također mogu imati tranzicije i olakšanja (eng. *Easing*) asocirana na sebi. Tako da onda protok vremena više nije linearni nego je ograničeni tokom grafa koji se odabere.



Slika 18: Prikaz svih kombinacija olakšanja i tranzicija (Izvor: r/godot) [15]

Da asociramo tranzicije i olakšanja određenom Tweenu, može se to napraviti tako da koristimo set_ease i set_trans metode na Tweenu. Metode primaju samo predodređene enumeracije za argumente.

```
func _ready() -> void:
	var tween = create_tween()

	tween.tween_property(
		root, "global_position", Vector2(120, 90), 1.0
	).set_ease(Tween.EASE_IN).set_trans(Tween.TRANS_QUART)

	tween.tween_property(
		root, "rotation", 90, 1.0
	).set_ease(Tween.EASE_OUT).set_trans(Tween.TRANS_BACK)
```

Gore navedeni kod rezultira pomakom prema poziciji (120, 90) kao što je prikazano na slici 18. pod "QUART" tranzicijom označeno sa zelenom bojom gdje pomak sve brže i brže ubrzava te se onda naglo zaustavi. Nakon što se pomak prema određenoj poziciji izvrši, root scena se rotira prema tranziciji "BACK" s olakšanjem tipa "EASE_OUT" označena narančastom bojom na slici 18. koja uspori rotaciju prema kraju trajanja tweena rotacije.

Tweenove je moguće pokrenuti istovremeno uz set_parallel(true) gdje se onda svi tweenovi izvršavaju istovremeno do trenutka kad se set_parallel metoda postavi na false.

```
func _ready() -> void:
	var tween = create_tween().set_parallel(true)

	tween.tween_property(
		root, "global_position", Vector2(120, 90), 1.0
	)
	tween.tween_property(
		root, "rotation", 90, 1.0
	)
```

Gore navedeni kod rezultira rotacijom od 90 stupnjeva i pomaka prema globalnoj poziciji od (120, 90) od 1 sekunde istovremeno jer je set_parallel metoda na tween varijabli postavljena na true.

4.9.Dodatna terminologija

4.9.1. Resursi

Do sad smo se fokusirali na čvorove koji služe za logiku aplikacije. Postoji još jedan tip podataka koji je isto tako važan, a to su resursi. Za razliku od čvorova koji crtaju, simuliraju fiziku, poslažu korisničko sučelje… Resursi su spremnici podataka za čvorove. Samostalno ne rade ništa, ali u tandemu sa ostalim elementima Godota kao što su čvorovi, resursi mogu sadržavati podatke o nečemu važnomu za određeni čvor. [16]

Bilo što Godot učita s diska je resurs. Pod to spadaju teksture, slike, animacije, fontovi i ostale datoteke koje Godot koristi.

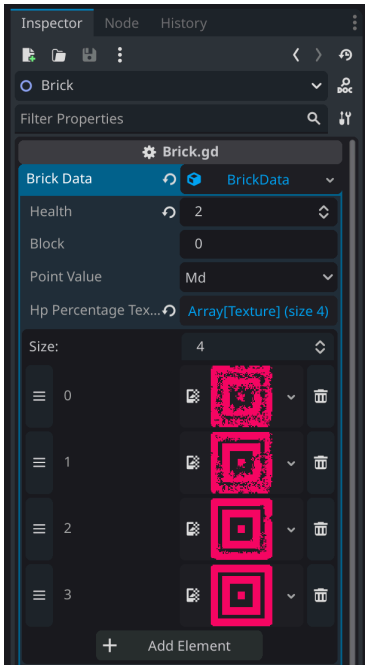
```
extends Resource
class_name BrickData

@export var health: int = 5
@export var block: float = 0.0
@export var point_value: PlayerStats.Points = PlayerStats.Points.MD
@export var hp_percentage_textures: Array[Texture] = []

func get_health() -> int:
	return health

func damage(amount: int) -> void:
	health = health - amount
	if health < 0: health = 0

func get_size_of_hp_textures() -> int: return hp_percentage_textures.size()

func get_points_from_brick() -> int: return point_value
```

Gore navedeni primjer resursa koristi se unutar igre zove se BrickData. BrickData naziv se sada ponaša kao klasa te se može koristiti za definiranje tipa podatak varijabli ili bilješki. BrickData služi za definiranje bilo kojem čvoru da on ima mogućnost unutar sebe raditi “operacije” za Brick entitet. U ovom slučaju definira se @export varijable koje nam služe za preddefiniranje nekih vrijednosti za instance svih scena te i neke getter i setter metode za dohvaćanje tih podataka.

Ako bi htjeli koristiti BrickData tip podataka za definiranje neke @export bilješke moramo ju kreirati sa tim tipom podataka unutar nekog čvora:

```
@export var brick_data: BrickData
```

Vidi se kako se u gore navedenom isječku koda koristi BrickData class_name kao tip podataka koji @export varijabla očekuje. Prikaz bilješke izgleda ovako:



Slika 19: Prikaz inspetkora sa bilješkom tipa BrickData

Vidimo kako su nam u slici 19. svojstva bilješki skripte od tipa podataka: BrickData jer unutar @export varijable kreiramo novi resurs koji je samo referencirani na varijablu brick_data koji unutar sebe sadrži svojstva i metode od BrickData resursa odnosno tipa podataka sa class_name imenom BrickData.

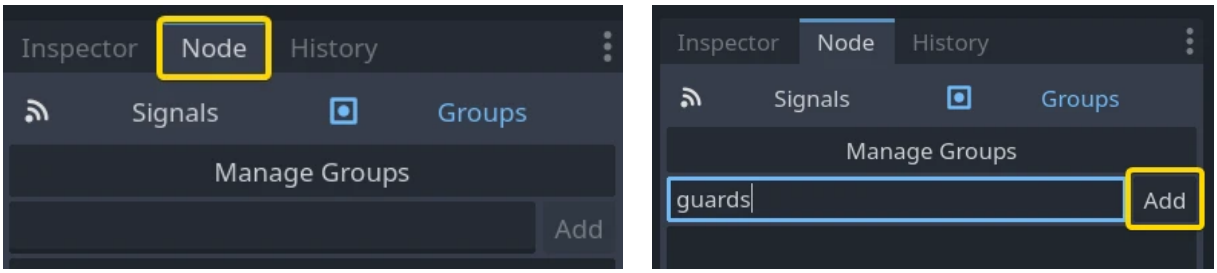
Sada možemo brick_data varijabli pristupiti unutar bilo kojeg dijela skripte gdje se treba koristiti te imati pristup lokalnoj instanci resursa tipa BrickData pod imenom brick_data. Naravno onda imamo pristup i metodam i ostalim svojstvima iz resursa BrickData za tu lokalnu instancu. Na primjer kao što su gore navedene metode: get_health, damage… koje se samo odnose na tu lokalnu instancu Brick entiteta odnosno lokalnu instancu BrickData resursa. Možemo si zamisliti da je BrickData sad postala tip podataka ili alias za naš novo kreirani resurs.

4.9.2. Grupe

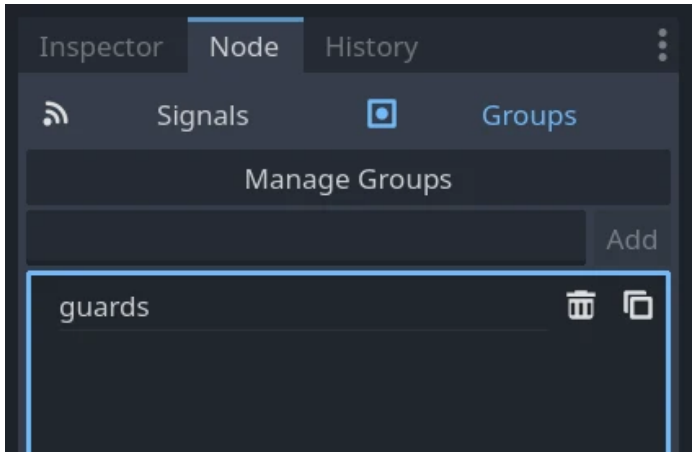
Grupe u Godotu rade kao oznake u ostalim softverima. Možemo pridodati čvoru koliko god grupa želimo. Nakon toga, možemo koristiti stablo scena za:

- Dobivanje popisa čvorova u grupi radi organizacije.
- Pozivanje metode nad svim čvorovima u grupi.
- Slanje obavijesti svim čvorovima u grupi.

Ovo je korisno za organiziranje velikih scena i odvajanje funkcionalnosti koda. [17] Grupe se kreiraju tako da određenom čvoru ili sceni pridodamo grupu nekog imena.



Slika 20, 21: Prikaz dodavanja grupe čvoru (Izvor: Godot Docs) [17]



Slika 22: Prikaz dodane grupe čvoru (Izvor: Godot Docs) [17]

Slika 22. prikazuje kako smo dodali čvoru grupu zvanu “guards”. Grupe možemo za više toga koristiti. Na primjer, može se ih koristiti za obavještavanje više čvorova ili scena da se nešto desilo. Ako bi stavili “guards” grupu na neprijateljsku scenu koja javi događaj svim ostalim scenama unutar iste grupe da se je igrač detektirao. Za javljanje svim ostalim neprijateljima da je igrač detektirani radimo ovako:

```
# https://docs.godotengine.org/en/stable/tutorials/scripting/groups.html
func _on_player_spotted():
	get_tree().call_group("guards", "enter_alert_mode")
```

Gdje je onda “enter_alert_mode” metoda unutar svih čvorova te grupe koja se aktivira, a “guards” ime grupe koju smo prije kreirali.

4.9.3.Timer čvor

Timer čvor je jedan od ključnih čvorova za rad sa Godotom. Timer je bitan za rukovanjem i radom vremenski određenim događajima. Omogućuje stvaranje mjerača vremena koji mogu pokrenuti ili zaustaviti određene radnje nakon određenog razdoblja. Čvor se može dodati sceni bilo kroz editor kao čvor u sceni ili putem skripte.

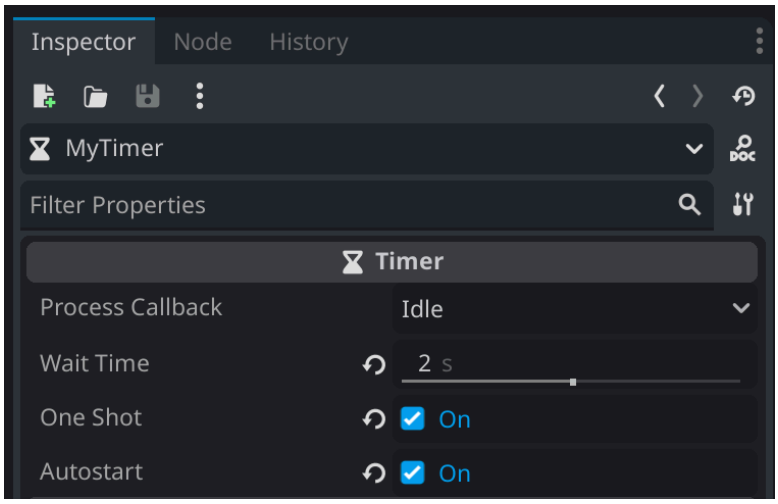
Ako želimo dodati timer čvor kroz kod možemo to ovako:

```
func _ready() -> void:
	var my_timer = Timer.new()
	my_timer.wait_time = 2
	my_timer.autostart = true
	my_timer.one_shot = true
	my_timer.timeout.connect(_on_timer_timeout)

	add_child(my_timer)

func _on_timer_timeout() -> void:
	print(“Timer je završio nakon 2 sekunde”)
```

Spajanje na izvršavanju našeg ručno kreiranog Timer čvora se radi preko pristupa timeout svojstvu i spajanje na signal koji se izvrši pri završetku wait_time svojstva od 2

sekunde. Nakon isteka vremena od wait_time, spajamo se na nama prilagođenu funkciju "_on_timer_timeout" koja se izvrši na timeout od Timer čvora.

Također, Timer čvor se može dodati i ručno unutar scene gdje mu se može pristupiti ili preko koda ili preko inspektora da odradi neku funkcionalnost na timeout signalu.



Slika 23: Prikaz postavki Timer čvora unutar inspektora čvora

Vidi se kako je puno jednostavnije postaviti Timer čvora na određene postavke u odnosu da ručno moramo unutar koda konfigurirati čvora. Naravno na isti način se spaja na signal koji se izvrši pri isteku vremena (wait_time) čvora. Naravno s obzirom na to da se čvor nalazi unutar scene, ciljani signal za timeout događaj može se kreirati unutar "Node" tab pored inspektor taba, što će rezultirati sa manje koda za konfiguraciju signala čvora.

4.9.4. Dictionary

Rječnici su asocijativni spremnici koji sadrže vrijednosti referencirane jedinstvenim ključevima (eng. *Key value pair*). Rječnici će sačuvati redoslijed umetanja prilikom dodavanja novih unosa. U drugim programskim jezicima ova se struktura podataka često naziva hash mapom ili asocijativnim nizom. [18]

Rječnik se može definirati stavljanjem popisa parova kao ključ:vrijednost par (eng. *key:value pair*) odvojenih zarezima unutar vitičastih zagrada.

```
# [https://docs.godotengine.org/en/stable/classes/class_dictionary.html]
var prazni_dict = {} # Kreiramo prazni rječnik.

var dict_val_2 = "druga vrijednost"
var dict_val_3 = 3
var moj_dict = {
        "kljuc": "prva vrijednost",
        "kljuc_2": dict_val_2,
        "kljuc_3": dict_val_3,
}
```

Unutar gore navedenog koda kreira se rječnik zvani “moj_dict” koji sadrži 3 ključeva gdje prvi ključ sadrži vrijednost tipa String unutar asociranog ključa sa vrijednošću “prva vrijednost” dok se ostali ključevi: kljuc2, kljuc3 referenciraju na vrijednosti od varijabli dict_val_2 i dict_val_3.

Za pristup rječniku koristimo:

```
func _ready() -> void:
	# Ključevima pristupamo prema string imenu ključa
	var neki_kljuc = moj_dict[“Neki kljuc”]
```

Rječnik tip podataka također može sadržavati i kompleksnije tipove podataka, kao što je na primjer polje:

```
var my_dict = {
	"Polje": [1, 2, 3, 4] # Pridodajemo polje brojeva ključu “Polje”
}
```

Dodati novo polje odnosno rječniku dodati novi “key:value” par radimo tako da samo “postavimo” ključ koji ne postoji. Primjer koda:

```
func _ready() -> void:
	var moj_dict = {}
	for key in moj_dict:
		# Ispisuje prazno jer nema ničega
		print(“DICT: ”, moj_dict[key])

	moj_dict[“prva_vrijednost”] = 1
	moj_dict[“druga_vrijednost”] = “2”
	for key in moj_dict:
		# Ispisuje 1, “2” jer smo ih dodali gore
		print(“DICT: ”, moj_dict[key])

	# Ispisuje drugu vrijednost direktno što rezultira sa izlazom “2”
	print(“Druga vrijednost: ”, moj_dict[druga_vrijednost])
```

Rječnike unutar ovog završnog rada koristi se za čitanje i spremanje postavki unutar igre.

5. Kreiranje korisničkog sučelja

Dizajn korisničkog sučelja (eng. *User Interface* - UI) u Godotu se radi uz pomoću CanvasLayera čvora. CanvasLayer čvor pruža osnovu za stvaranje dinamičkih i interaktivnih korisničkih sučelja koja se mogu prilagoditi različitim scenama i rezolucijama igre. CanvasLayer čvor je neovisni o događanjima ispod sebe, možemo si zamisliti da postoji na zasebnom z-inexu u odnosu na ostale čvorove kao što su Node2D i slično, što ga čini idealnim za dizajniranje elementa korisničkog sučelja koji moraju ostati dosljedni i ne ometani u različitim rezolucijama i stanja igre.

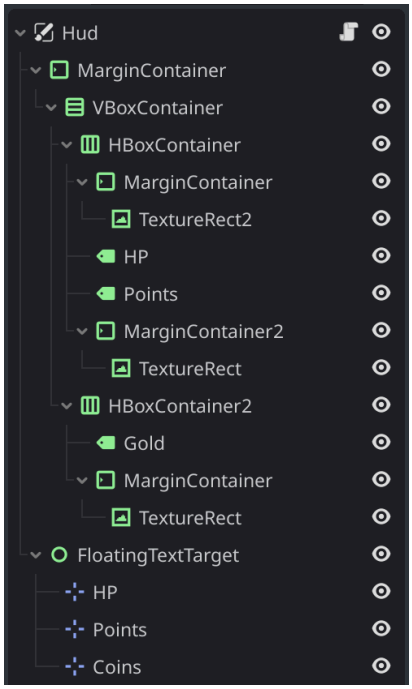
Za dizajniranje korisničkog sučelja u Godotu, prvo moramo razumjeti njegovu hijerarhijsku strukturu. Čvor CanvasLayer služi kao korijen za elemente korisničkog sučelja kao što su gumbi, oznake, spremnici, panele i slično. Ova separacija CanvasLayer čvora od ostatka igre osigurava da nam na korisničko sučelje ne utječu pokreti i transformacije kamere u igri, održavajući nama lako predvidljivo sučelje.

Godotovo pseudo low-code okruženje što se tiće povuci i ispusti sučelje i GDScript omogućuju da se interakcije i animacije skriptiraju sa dodatnom logikom, te pritom i obogaćujući korisničko iskustvo.

5.1. CanvasLayer čvor

Korištenjem CanvasLayera, mogu se stvoriti sučelja koja povećavaju interes korisnika i održavaju funkcionalnost u različitim scenarijima igre. Kao što smo rekli, CanvasLayer čvor nam služi za separaciju 2D igre i korisničkog sučelja da ne utječu jedan na drugog. [19]

Implementacija UI elemenata uključuje dodavanje kontrolnih čvorova kao djece čvora CanvasLayer. Kontrolni čvorovi su svestrani i pružaju razne ugrađene funkcije kao što su usidra, postavke margina i mogućnosti responzivnog dizajna. Na primjer, dodavanje čvora tipa Button sloju CanvasLayer omogućuje dijete CanvasLayer čvora ostane fiksiran na zaslonu bez obzira na transformacije scene igre, odnosno svega što se nalazi ispod igre.

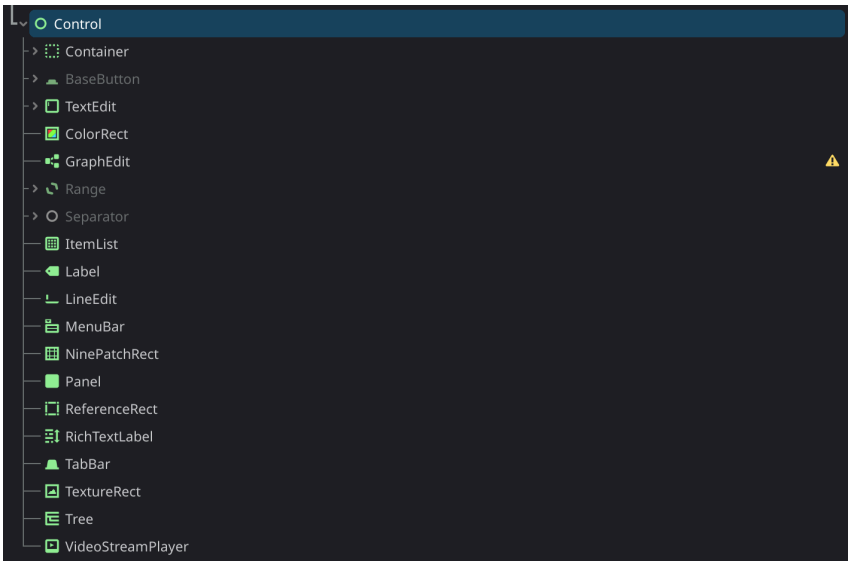


Slika 24: Primjer CanvasLayer čvora

U gore navedenoj slici 24. vidimo kako su svi čvorovi koji su djeca CanvasLayer čvora koji je root scene sa imenom “Hud” (eng. *Heads Up Display*) zelene boje koja označuje da su oni control tipa. Control tip čvora označuje čvor koji bi trebao spadati pod CanvasLayer. Isto tako Control čvorovi imaju posebna svojstva koja su posebno stvorena za upravljanje korisničkim sučeljem.

5.1.1. Control čvor

Control čvor je osnovni čvor za upravljanje korisničkim sučeljem. Svi ostali čvorovi zelene boje unutar stabla scene nasljeđuju svoja svojstva od Control čvora.



Slika 25: Prikaz svih čvorova koji nasljeđuju od Control čvor

Control čvor sadrži granični pravokutnik koji definira svoje opsege, položaj usidrenja u odnosu na nadređenu kontrolu ili trenutni okvir za prikaz i pomake u odnosu na usidrenu točku. Pomaci se ažuriraju automatski kada se promijeni čvor, također, kod promjene kojih od njegovih roditelja ili veličina zaslona. [20].

Control čvor ima puno preddefiniranih signala kao što su mouse_entered, focus_entered, gui_input i ostali koji služe za upravljanjem korisničkim sučeljem. Korisničkim sučeljem se upravlja na isti način kao i ostatak GDScripta, odnosno uz pomoć signala. Svaki događaj koji se desi unutar Control čvora ili njegove djece se može uhvatiti te onda odraditi preddefinirana metoda za taj specifični događaj kojeg osluškujemo.

Za jednostavni primjer može se kreirati čvor tipa Button koji kad se pritisne pošalje nam poruku da je pritisnuti. Za ovaj primjer nam je potreban čvor tipa Button te i signal koji se nalazi unutar čvora: _on_pressed:

```
extends Control

var br_pritisnuti: int

func _ready() -> void:
	print("Scena je spremna!")
	br_pritisnuti = 0

func _on_pressed() -> void:
	br_pritisnuti = br_pritisnuti + 1
	print("Gumb je pritisnuti" + br_pritisnuti + " puta!")
```

Ako ne želimo spajati signale kroz grafičko korisničko sučelje, odnosno preko inspektora. Što rezultira manje jasnijem kodu ako je puno signali koji se presretaju unutar jedne skripte. Možemo ručno spojiti signal gumba preko @onready bilješke.

```
extends Control

@onready var gumb: Button = $Button

var br_pritisnuti: int


func _ready() -> void:
	gumb.on_pressed.connect(_on_gumb_je_stisnuti)
	print("Scena je spremna!")
	br_pritisnuti = 0

func _on_gumb_je_stisnuti() -> void:
	br_pritisnuti = br_pritisnuti + 1
	print("Gumb je pritisnuti" + br_pritisnuti + " puta!")
```

Gore navedeni kod radi isto što i prije navedeni kod. Jedina razlika je to što mi kreiramo vlastitu metodu "_on_gum_je_stisnuti" na koju se signal spoji nakon njegovog okidanja. Obavezno je spajanje signala od gumb koji je tipa Button iz čvora koji se zove Button s definicijom signala "on_pressed" i connect što očekuje referencu na metodu koju želimo izvršiti nakon okidanja signala.

5.1.2. Container čvor

Container čvor služi za spremanje i upravljanje ostalim čvorovima tipa Control. Container čvor je osnovna klasa od koje svi ostali spremnici nasljeđuju.

Container odnosno spremnik je osnovna klasa za sve spremnike korisničkog sučelja. Container čvor automatski raspoređuje svoje podređene Control čvorove na određeni način. Ova se klasa može naslijediti za izradu prilagođenih vrsta spremnika. [21]

Najčešće korišteni spremnici koji nasljeđuju od Container čvora su: MarginContainer, GridContainer, BoxContainer, HBoxContainer, VBoxContainer i PanelContainer. Oni služe za aranžiranje svoje djece na određeni način unutar svojih definiranih granica.

Container čvorovi uvijek imaju “Container Sizing” definirani unutar svojih svojstava koji omogućuje naprednije uređivanje djece Control tipa.

Slika 26: Prikaz container sizing-a unutar HBoxContainer tipa čvora

Slika 27: Prikaz primjera HUD sučelja sa spremnicima

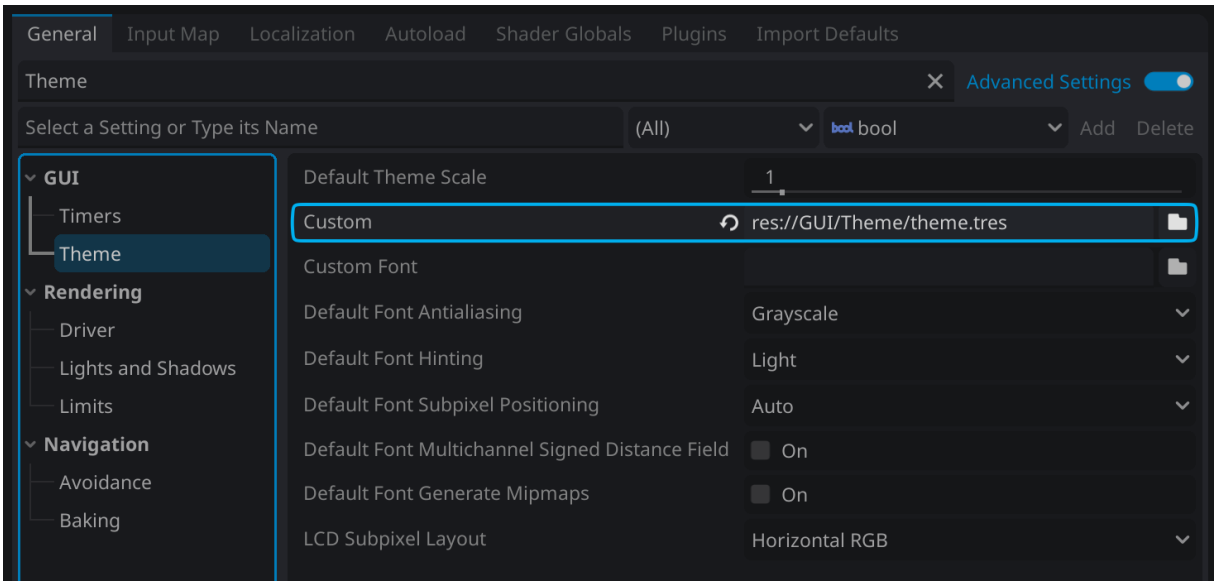
Gore navedena slika 27. prikazuje odabrani HBoxContainer koji sadrži djecu tipa MarginContainer, Label i TextureRect. Svi oni su posloženi po horizontu radi HBoxContainer koji svoju visinu definira po visini svog najvećeg djeteta. U ovom slučaju, sva djeca su približno iste veličine. HBoxContainera da svoju djecu posloži tako da su djeca usidrena samo na lijevoj i desnoj strani je preko Horizontal svojstva koji spada pod “Container Sizing”

koji je prikazani na slici 26. no samo to nije dosta. Svako dijete HBoxContainer čvora moramo “forsirati” da sebe usidri na lijevu, odnosno desnu stranu unutar svoje granice definirane od trenutne HBoxContainer veličine. Trenutno je pod Label čvor “HP” odabrani Fill s Expand svojstvom. S obzirom na to da je to tekstualni čvor koji se piše s lijeva na desno ne mora se striktno forsirati da se on usidri na lijevu stranu jer se uvijek bude. Isto tako Label čvor ima dodatna svojstva za uređivanje teksta što rezultira manje nespretnim baratanjem sa slaganje više container tipa čvorova da zajedno rade.

5.2. Kreiranje teme za korisnička sučelja

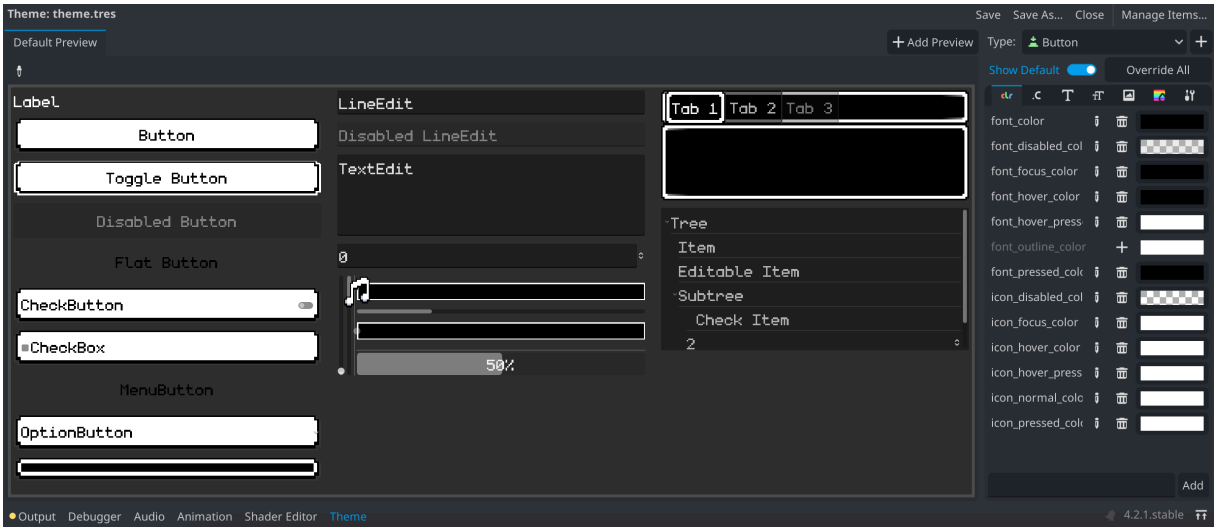
Teme su resursi koji se koristi za stiliziranje Control i Window čvorove. Dok se pojedinačne kontrole mogu stilizirati pomoću nadjačavanja njihove lokalne teme preko theme_override svojstva, resursi teme omogućuju pohranu i primjenu istih postavki na sve Control čvorove koji su iste vrste (npr. stiliziranje svih gumba na isti način). Jedan resurs teme može se koristiti za cijeli projekt. Resurs teme dodijeljen kontroli primjenjuje se na samu kontrolu, kao i na svu njezinu izravnu i neizravnu djecu. [22]

Glavnu temu kreiramo tako da kreiramo novog resursa sa “.tres” ekstenzijom te za koji odaberemo da je tipa “Theme”. Da bi mogli koristiti glavnu temu globalno moramo ju dodati unutar postavki projekta.



Slika 28: Prikaz dodavanja teme untar postavki projekta

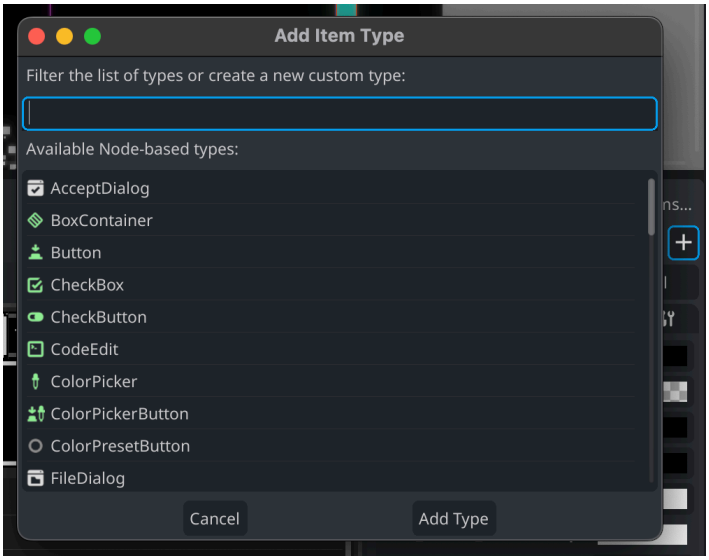
Pri kreiranju i dodavanja globalne teme nam se otvara novi tab prozor na dnu editora zvan “Theme”. Unutar novo kreiranog prozora mi možemo urediti našu temu.



Slika 29: Prikaz već konfigurirane Theme panele

Unutar Theme panele moe se kreirati prebrisivaje stila kontrolnih čvorova (eng. T*heme Override*) za sve kontrole. Vidi se na slici 29. kako su nam predstavljene najčešći čvorovi tipa Control koje možemo uređivati. U gore lijevom kutu vidi se Label i Button kontrolu koji su interaktivni. To je izuzetno važno da može testirati korisničko sučelje bez da se mora pokretati aplikaciju i onda testirati tijekom izvršavanja aplikacije. Također, vidi se kako imamo definirani vlastiti font.

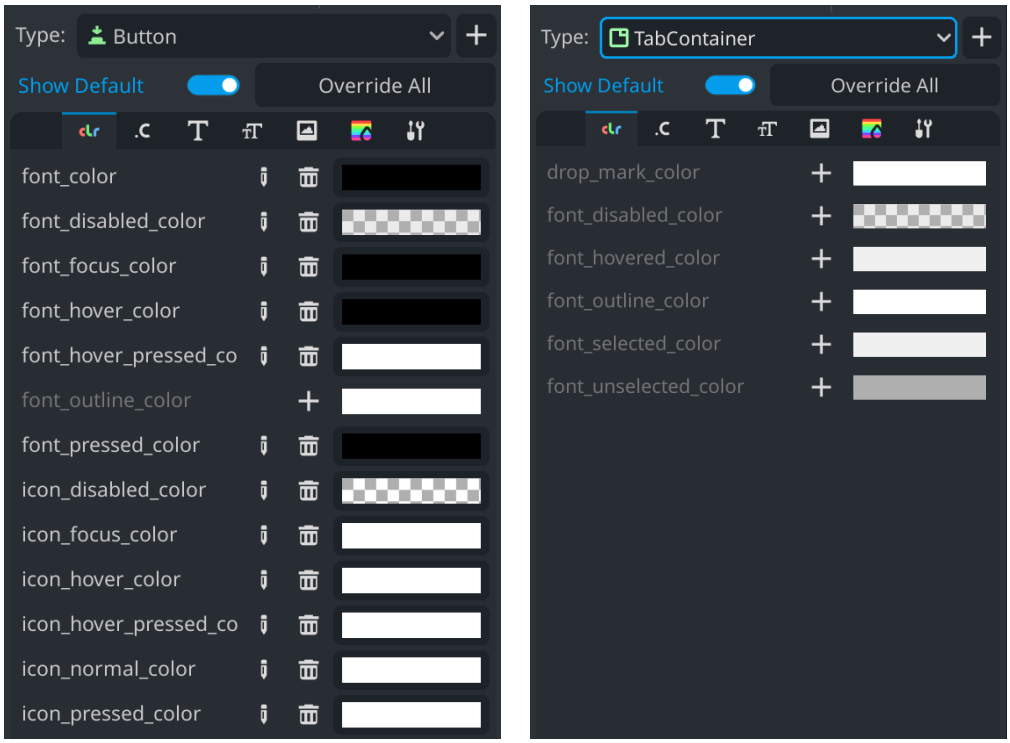
Za kreiranje theme override klikne se na gornje desni '+' gumb i odaberemo Control čvor koji želimo prebrisati s vlastitim stilom.



Slika 30: Kreiranje novog theme override unutar glavne teme

Kreiranjem novog theme override možemo manipulirati svojstvima stila tog čvora. Generalno za svaki tip čvora su nam dana preddefinirana svojstva koja se u velikoj većini slučajeva želi koristiti. Na primjer, kod gumbi imamo stanje izgleda gumba kod fokusiranja na

gumb, klik na gumb, izgled gumba dok je onesposobljeni i slično. TabContainer čvor u odnosu na Button čvor ima drugačija svojstva izgleda koje možemo promijeniti.



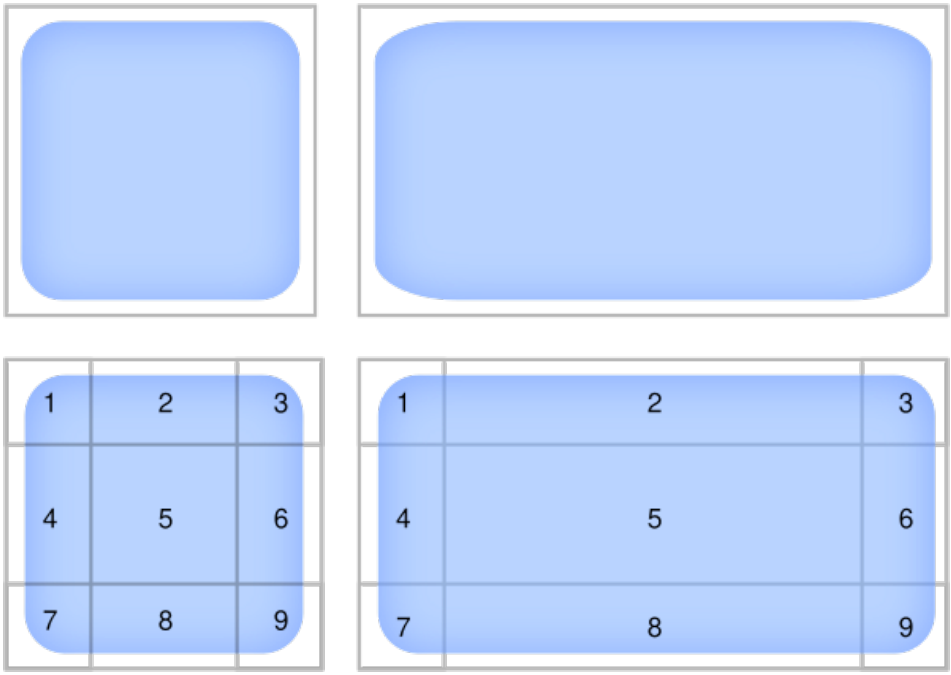
Slika 31, 32: Prikaz različitih postavki čvorova tipa Button i TabContainer

Unutar theme override za svaki Control čvor možemo dodati i StyleBox teksture nad čvorovima. To omogućuje urediti izgled igre na što god želimo, a za to je potreban koncept nine-slice.

5.2.1. Nine-slice

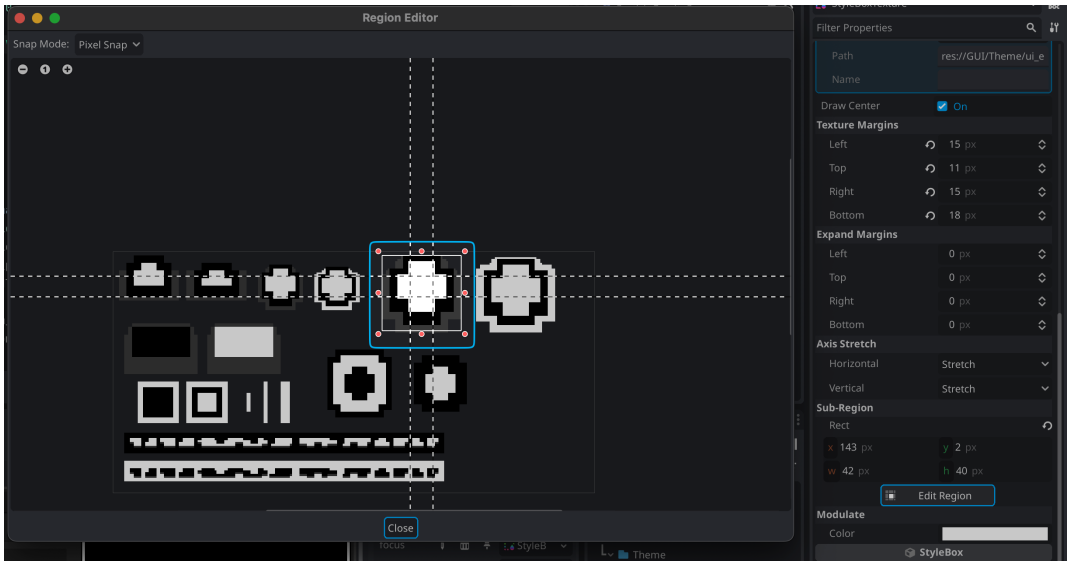
Problem kod video igara, specifično kod dizajniranja korisničkih sučelja je skaliranje grafike i rezolucija ekrana na kojemu se igra prikazuje. Veliki je problem ako nemamo kontrolu nad veličinom prozora ili prozora i elementa koji mogu dinamički promijeniti svoju veličinu. Promjenom njihove veličine njihova tekstura se iskrivi ili uveća disproporcionalno što rezultira lošoj kvaliteti slike. Rješenje za takav problem nam je nine-slice ili takozvani (eng. *9-slice scaling*).

Nine-slice služi za definiranja granica određenog elementa i slike koji se onda isprinta nad elementom. Rezultirajuća slika vanjske dijelove slike pokuša ne iskriviti te zadržati omjer, dok sve što se nalazi unutar granica se duplicira ili skalira. [23]. Zbog toga kod većine video igra, sve što se nalazi unutar dinamičnih elementa korisničkih sučelja ima neki uzorak koji se samostalno po sebi ponavlja ili je vrlo jednostavne naravi (najčešće jednobojni, gradijent ili proziran).



Slika 33: Numerirani prikaz nine-slice (Izvor: Wikipedija) [23]

Vidi se na slici 33. kako je prikazani nine-slice gdje se područje označeno sa brojem 5 rasteže. Ostala područja koja nisu rubovi označeni sa brojevima: 2, 4, 6, 8 nam se onda rastežu ovisno o osi na kojoj se nalaze te se ne rastežu unutar suprotne osi. Na primjer, područja 2 i 8 se samo mogu rastegnuti unutar y osi, dok se područja 4 i 6 mogu samo po x osi. Rubovi, odnosni područja: 1, 3, 7 i 9 se ne rastežu po niti jednoj osi čime zadržavaju svoj omjer i izgled. Nine-slice se koristi unutar gumba gdje se želi da je izgled gumba neovisan o njegovoj veličini.



Slika 34: Prikaz uređivanja nine-slice gumba unutar teme

Vidi se na slici 34. kako je središnje ponavljajuće područje označeno sa pravokutnikom gdje se sve crtkane linije sijeku. Taj dio je dio koji se bude ponavljao unutar elementa. Ostali dijelovi se ponašaju na način na koji je nine-slice definiran. Drugi pravokutnik je takozvani “bounds rectangle” koji definira ukupno područje sličice koja će se prikazati kao nine-slice.

Trenutno odabrani nine-slice je za stanje mirovanja gumba dok nije pritisnuti. Ako bi željeli kreirati stanje gumba kad je pritisnuti onda moramo kreirati novi nine-slice sa drugom sličicom koja je gotovo pa identična, osim jednog dijela kao što je na primjer druga boja obruba ili slično. Stanja gumba dok je pritisnuti, hoover, fokus i slično rješava Godot.

Ostali elementi slike 34. su svi elementi koji se koriste unutar korisničkog sučelja. Prikazana slika je slika ui_elements koja nam sadrži sve slike za elemente koje mislimo koristiti za nine-slice unutar jedne png datoteke. Dobro nam služi za brzo prototipiranje i brzo pronalaženje svih resursa za korisničko sučelje unutar jedne datoteke.

6. Kreiranje korisničkog iskustva

Korisničko iskustvo (eng. *User Experience* - UX) u razvoju video igara je ključni aspekt koji određuje uspjeh i dugovječnost igre. Za razliku od drugih softverskih aplikacija, igre su prvenstveno dizajnirane za zabavu, što korisničko iskustvo čini ključnim faktorom u angažmanu i zabavi igrača.

Temelj korisničkog iskustva u igrama je razumijevanje očekivanja i ponašanja igrača. Moramo uzeti u obzir različite elemente kao što su mehanika igre, estetika i razni sustavi povratnih informacija korisniku. Svaka od navedenih komponenti ima važnu ulogu u oblikovanju iskustva igrača. Na primjer to su: intuitivne kontrole, korektna i intuitivna navigacija osiguravaju da se igrači mogu usredotočiti na igru bez nepotrebnog ometanja, efekti i slično. Također, uvjerljiva priča i vizualno privlačna grafika mogu uroniti igrače, čineći igru boljom i zabavnijom. Visoka privlačna grafika se naravno ne mora samo odnositi na grafiku visoke kvalitete, veći na stiliziranu grafiku koja sama posebni ima određenu dugotrajnu ili nostalgičnu vrijednost.

Još jedan važan dio korisničkog iskustva je povratna informacija (eng. *Feedback*) na interakcije korisnika. Pogotovo kod mobilnih uređaja gdje bi s dodirom prsta trebalo osjetiti određenu povezanost s uređajem odnosno s igrom. Takvo iskustvo pruža se korisniku tako da se dodaju efekti na određene interakcije korisnika. Na primjer, klikom na gumb on malo promjeni svoj izgled ili dodavanjem efekta čestica (eng. *Particle effects*), podrhtavanje kamere koji korisniku sugerira određenu težinu i snagu nekog događaja. Dodatna interaktivnost izvan samog uređaja kao što je pristupanje akcelerometru mobilnog uređaja da kreiramo efekt paralaksa nad korisničkim sučeljem gdje se određeni elementi korisničkog sučelja kreću sporije sa ili u suprotnom smjeru trenutnog uređaja. Takav efekt rezultira osjećajem dubine unutar aplikacije te se pritom brišu zamišljene linije između aplikacije, uređaja i vanjskog svijeta. Takve stvari je zahtjevno za isprogramirati ili čak opravdati jer koštaju relativno puno resursa, a pridodaju gotovo pa ništa igri, ali se baš na takvim malim dodacima kreira izuzetno korisničko iskustvo.

6.1. WorldEnvironment

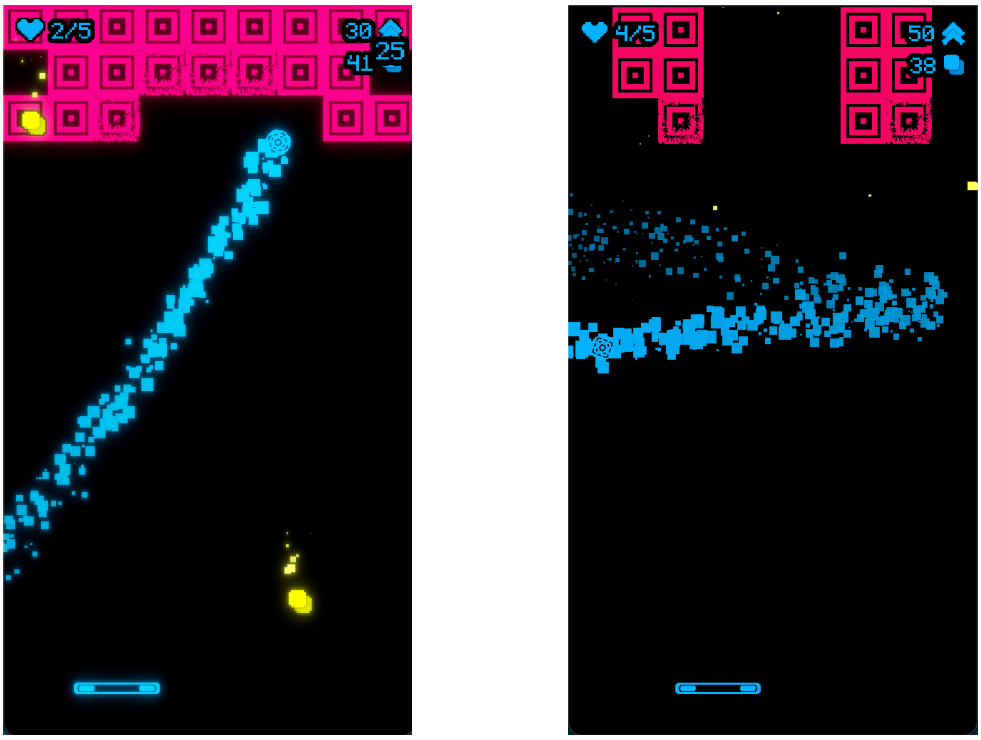
Vizualna aspekt igra ključnu ulogu u stvaranju impresivnih i privlačnih iskustava. Godot nudi veliku količinu alata za postizanje visokokvalitetnih vizualnih prikaza, a jedan od njih je WorldEnvironment. Ovaj čvor je sastavni dio definiranja i kontrole globalnih vizualnih parametara 3D i 2D scena (eng. *Post processing*), obuhvaćajući sve od ambijentalnog osvjetljenja do sofisticiranijih efekata magle. [24]

WorldEnvironment čvor je jedan od rijetkih čvorova koji radi samostalno bez ikakvih direktnih interakcija sa svojom djecom ili roditeljima. On samo pregleda trenutno stanje scene i primjeni efekte na nju. Postoje i više različitih načina rada kod WorldEnvironment čvora unutar kojih on može raditi. Jedan od njih je takozvani “Canvas mode” koji pregleda što je sve na kanvasu i primjeni efekte na to. Djeca CanvasLayer čvora nisu pod utjecajem WorldEnvironment efekata. [25]

Većina efekata unutar WorldEnvironment čvora su primarno zamišljeni za rad sa 3D okruženjima. Malo koje postavke rade na 2D okruženjima. Postavka koja nas zanima, a da rade unutar 2D okruženja je sjaj (eng. *Glow*). Sjaj nam služi za dodavanje postupnog gradijenta svijetlu svakom elementu unutar stabla scena, izuzev CanvasLayer čvora. Moguće je određenim čvorovima maknuti sjaj tako da podesimo njihove HDR (eng. *High Dynamic Range*) threshold. Time se može utjecati na koje čvorove želimo da se sjaj primjeni. [25]

Slika 35: Prikaz sjaja preko WorldEnvironment čvora (Izvor: Godot Docs) [25]

Vidi se na slici 35. kako je sličica na lijevoj strani i sama pozadina bez sjaja. Dok ostale 3 sličice na desnoj strani, sadrže efekt sjaja.



Slika 36, 37: Prikaz sa efektom sjaja (lijevo) i prikaz bez (desno)

Vidimo na slikama 36, 37. Kako je slika 36. sadrži više dubine zbog efekta sjaja u odnosu na sliku 37. Naravno, vrijednosti sjaja se može podesiti kao što je prikazano na slici 38.

Slika 38: Prikaz postavki sjaja na WorldEnvironment čvoru unutar scene

Unutar slike 38. vidi se kako se može podesiti puno postavki što se tiče sjaja. Pod postavkama može se podesiti količinu sjaja unutar razina sjaja gdje veće vrijednosti pod prvom razinom rezultiraju “užim i intenzivnijim pojasom dok veće vrijednosti poput razine 7. rezultiraju opširnijim ili “prigušenim” sjajnom. Vrijednosti unutar razina govore koliko želimo kojeg pojasa iskoristiti. Postavke za normalizaciju odnose se na vrijednosti unutar razina gdje normaliziramo sve vrijednosti da njihov sum bude jednaki 1. Intenzitet se odnosi na jačinu svjetline dok se snaga odnosi na ukupnu količinu sjaja. Bloom i blend mode označuju način na koji se sjaj primjenjuje na viewportu.

6.2. GPUParticles2D

Čvor za kreiranje 2D čestica koji se koristi za stvaranje raznih sustava čestica i efekata. GPUParticles2D sadrži emiter koji generira određeni broj čestica zadanom brzinom. [26]

Unutar ovog rada, sustavi efekta čestica se primarno koriste kao proširenje korisničkog iskustva. Ako sustave čestica ne bi koristili unutar igre, igra bi ostala ista te se interakcija između korisnika i aplikacije ne bi ni na koji način promjenila. Sustavi čestica nam samo služe za poboljšanje korisničkog iskustva. Sustavi čestica koriste se za efekte uništavanje objekata, tragovi lopta, dodatni efekti za indikaciju greške i uspjeha igrač u određenim situacijama i slično.

Već smo prije demonstrirali na slici 36, 37. kako se sustavi čestica mogu koristiti da proširimo trenutnu scenu na još veću razinu zadovoljstva kod korisnika.

6.2.1. Dodatne interakcije sa sustavima čestica

Za dodatne interakcije između korisnika i igre, unutar igre također imamo i kreiranu komponentu “ParticleGravityComponent” koja prima kao @export bilješku čvor tipa GPUParticles2D. Pristupom GUParticles2D čvora mi možemo tijekom izvršavanja igre mijenjati način izvršavanja i prikazivanja sustava čestica.

Unutar komponente ParticleGravityComponent prebrisuje se trenutna gravitacija čestica na neku drugi vrijednost. Vrijednost se generira iz trenutne pozicije platforme koju korisnik koristi za odbijanje i usmjeravanje loptice. Vrijednosti se onda generiraju te mapiraju na jačinu gravitacije tog sustava gdje je gravitacija uvijek 0 u svim smjerovima, osim ako se igračeva platforma ne pomakne u bilo kojem smjerom gdje se onda točka gravitacije pomiče po osi. Vrijednosti dalje od središnje točke gdje se daska prvotno nalazi rezultira jačim mapiranjem vrijednosti gravitacije.

Te interakcije rezultiraju dodatnim efektima gdje se čestice više ne ponašaju strogo linearno, već se ponašaju fluidno i prirodno. Našu komponentu: ParticleGravityComponent

koristi se za postizanje prirodnog osjećaja kod loptice, gdje trag koji loptica ostavlja postiže fluidni osjećaj.

6.3.Shader

Shaderi su posebna vrsta programa koji rade na na grafičkoj kartici. U početku su se koristili za dodavanje dubine 3D scenama, ali danas za mogu mnogo više. Možemo ih koristiti za kontrolu načina na kojim se crta geometrija i pikseli na zaslonu, omogućujući nam postizanje svakakvih vrsta efekta. [27]

Moderni Game Engine za renderiranje poput Godota crtaju sve s shaderima: grafičke kartice mogu izvoditi tisuće instrukcija paralelno, što dovodi do nevjerojatne brzine renderiranja. Međutim, zbog svoje paralelne prirode, shaderi ne obrađuju informacije na način na koji to radi tipičan program. Kod shadera izvodi se na svakom vrhu ili pikselu zasebno. Podatke ne možemo pohranjivati unutar sličica (FPS). Kao rezultat toga, kada radimo s shaderima, moramo programirati i razmišljati drugačije od drugih programskih jezika. [27]

Primjer takvog razmišljanja je dolje navedeni kod koji za određenu visinu i širinu postavi boju piksela na neku neku boju:

```
for x in range(width):
  for y in range(height):
        set_color(x, y, some_color)
```

Gore navedeni kod je tradicionalni i jednostavni način kako bi to napravili bez puno muke. Dok je dolje navedeni način kreirani uz pomoć shadera radi isto, ali uz puno manje koda.

```
void fragment() {
  COLOR = some_color;
}
```

Vidi se da sa znatno manje koda, možemo postići isti rezultat gdje postavimo boju ekrana na neku određenu boju. Shaderi su izuzetno kompleksni i radikalnijeg razmišljanja u odnosu na tradicionalno programiranje. Oni zahtijevaju izuzetno dobro razumijevanje grafičkih kartica i matematike.

S obzirom na to da shaderi nisu glavni dio ovog rada, svi korišteni shaderi koji se koriste unutar igre su pribavljeni od posebnog foruma za kreiranje i objavljivanje Godot shadera: www.godotshaders.com, gdje bilo tko može uz par slika zaslona objaviti svoj shader da bi ga ostali mogli koristiti.

Svi shaderi koji su korišteni sadrže CC0 licencu za otvoreno korištenje s ili bez akreditacije vlasnika.

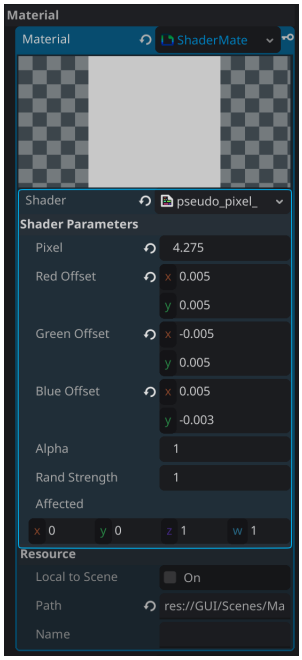
6.3.1. Primjer shadera za pikselizaciju ekrana

Jedan od shadera koji se koristi unutar igre je efekt pikselizacije prilikom tranzicija između scena. Korišteni kod od efekta pikselizacije nije vlastiti, već je od jednog od korisnika Godota koji je objavio svoj shader za ostale da ga mogu koristiti. Kod od shadera spada pod CC0 licencom što znači da je slobodno korištenje i bez citiranja autora shadera.

Shader za pikselizaciju radi tako da pregleda trenutno stanje kanvasa te onda sve piksele koji sadrže boju, odnosno nisu crni razdvoji na crvenu, zelenu i plavu. Pozicija svake zasebno razdvojene boje nam se zove Offset, a količina pikselizacije se zove Pixel kao što je prikazano u dolje navedenom kodu:

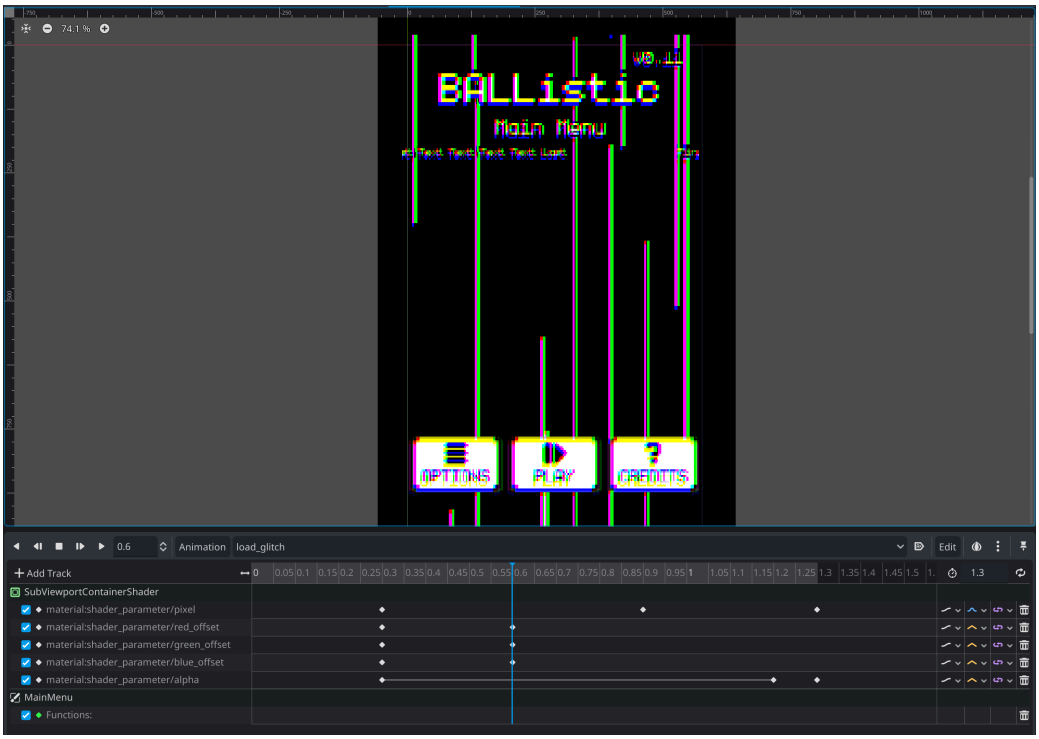
```
// Pribavljeno od GodotShaders:
// Dostupno na: https://godotshaders.com/shader/pixelated-glitch-
// posteffect/
// Kreirano 12.3.2021
// Autor: ombarus [35]

shader_type canvas_item;

uniform float pixel = 1.0;
uniform vec2 red_offset = vec2(0.0, 0.0);
uniform vec2 green_offset = vec2(0.0, 0.0);
uniform vec2 blue_offset = vec2(0.0, 0.0);
uniform float alpha = 1.0;
uniform float rand_strength = 1.0;
```

Ključna riječ uniform iz gore navedenog koda služi nam kao “varijabla” koju možemo manipulirati prije izvođenja programa. Vidimo na slici 39. Kako vrijednosti kod varijabla red_offset i ostalih varijabla je drugačija u odnosu na ono što je prikazano u inspektoru.



Slika 39: Prikaz uniforma iz shadera za pikselizaciju unutar inspektora čvora

Sada se može uz upotrebu Tweena ili AnimationPlayer čvora manipulirati vrijednosti uniform varijabli kroz određeno vrijeme, čime se dobije efekt pikselizacije za tranzicije.



Slika 40: Prikaz AnimationPlayer prozora kako kroz određeno vrijeme pikselizira ekran

Vidi se kako su nam unutar slike 40. prikazane animacijske trake pod čvorom SubViewportContainerShader koji sadrži shader za pikselizaciju. To je poseban čvor koji kao jedino dijete sadrži čvor SubViewport koji cijelo vrijeme kreira teksturu trenutnog prikaza ekrana (eng. *Viewport*) te onda proslijedi shaderu unutar SubViewportContainerShader. Da bi shader radio potrebne su nam te sličice jer inače samo radi efekt pikselizacije, ali ne i efekt razdvajanja određenih boja.

Ne preporučuje se korištenje SubViewport čvora za igre zbog toga što troši relativno puno memorije i resursa za kreiranja sličica trenutnog prikaza, ali s obzirom na to da je ovo početni meni (eng. *Main Menu*) gdje performanse i FPS nisu bitni, onda ga možemo koristiti jer ne utječe na naše performanse unutar igre.

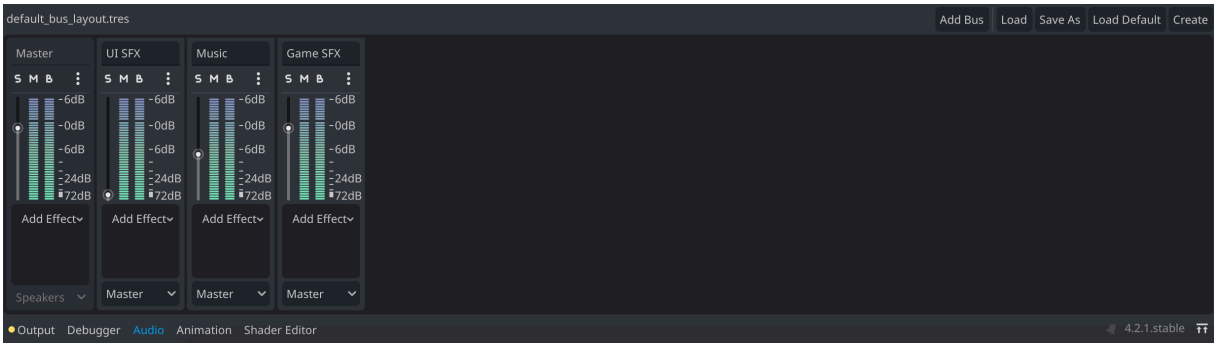
6.4. Zvuk

Za korištenje zvuka unutar igre, potrebni su nam AudioStreamPlayer čvorovi o kojima smo veći prije govorili. Postoje više AudioStreamPlayer čvorova od kojih je istoimeni samo za globalnu reprodukciju mono zvuka, dok AudiStreamPlayer2D čvor svoj zvuk mijenja ovisno o poziciji u odnosu na kameru gdje se reproducirani zvučni zapis stišava s većom udaljenošću od kamere odnosno igrača. Također postoji i AudioStreamPlayer3D za reprodukciju zvuka

unutar 3D okruženja, ali njega ne koristimo unutar ovog rada zbog toga što radimo sa 2D okruženjem, a ne 3D.

6.4.1. Zvučni kanali

Zvučni kanali (eng. *Audio Bus*) služi za upravljanje svim zvučnim reprodukcijama unutar igre. Zvučni kanali imaju mogućnost klasifikacije određenih zvukova na zasebni kanal. Pritom dobivamo mogućnost manipulacije svake vrste zvuka zasebno. Najčešće zvukovi unutar igara su podijeljeni u par kategorija: Master, UI SFX, Music, Game SFX. Isto kao i kod svih softvera za obradu zvuka (eng. *Digital Audio Workstation* - DAW). Svi zvukovi na kraju prolaze kroz Master kanal. Master kanal kontrolira ukupnu glasnoću igre, svi ostali kanali na kraju prolaze kroz njega. Uobičajeno je da se Master kanal ne dira, jer može poremetiti ostale kanale koje možda ne želimo manipulirati. Jedino što se može manipulirati na Master kanalu je ”Mute” opcija koja stiša sve kanale. [28]



Slika 40: Prikaz zvučnih kanala unutar igre

Vidimo kako glasnoću svakog kanala možemo zasebno namjestiti. Također možemo dodati i efekte na svaki audio kanal kao što su: delay, reverb, phaser i slično da dodatno možemo promijeniti osjećaj igre. Zvučni kanali su definirani unutar default_bus_layout.tres resursne datoteke koja je tipa AudioBusLayout.

Unutar svakog AudioStreamPlayer čvora, uvijek je po zadanom odabrani Master kanal, to nije dobra praksa jer onda nemamo kontrolu nad njim bez da remetimo ostatak zvučnih kanala. Za odabir kanala moramo promijeniti “Bus” svojstvo unutar inspektora čvora kao što je prikazano na slici 41.

Slika 41: Prikaz promjene kanala unutar AudioStreamPlayer čvora

6.5. Podrhtavanje kamere

Podrhtavanje kamere (eng. *Camera Shake*), je snažan vizualni efekt koji igra ključnu ulogu u poboljšanju korisničkog iskustva tijekom igre, simulirajući podrhtavanje kretanje kamere tijekom značajnih događaja u igri. Dodaje se sloj realizma i intenziteta koji statične slike često ne uspijevaju prenijeti. S obzirom na to da je unutar igre kamera stacionarna, podrhtavanje kamere omogućuje da se određenim značajnim trenucima doda osjećaj težine. Kada korisnik osjeti vibraciju ekrana u odnosu na neku eksploziju, koliziju i slično. To kreira osjećaj povezanosti igrača i igre.

Za kreiranje podrhtavanja kamere, mora se kreirati Autoload kojem se može pristupiti svugdje. Za početak treba kreirati enumeraciju unutar koje su sve moguće kombinacije jačine podrhtavanja kamere. Ovu apstrakciju obavezno treba napraviti jer je podrhtavanje kamere bazirano na više vrijednosti koje se lako mogu zaboraviti, tako da se ne mora pamtiti koliko je 12 jače u odnosu na 8 te pritom kreirati nekonzistentna podrhtavanja. Nakon što se kreiraju vrste podrhtavanja, iste se mora mapirati u rječnik koji unutar sebe ima definirane ključeve: strength i fade. Gdje je strength snaga, a fade je brzina izvršavanja.

Sada je samo preostalo unutar _process metode, prilikom pozivanja apply_shake metode izvan skripte. Kreirati linearnu interpolaciju od vrijednosti unesene snage do vrijednosti 0 kroz vrijeme shake_fade (iz shake_settings) pomnoženo sa delta za konzistentnu brzinu bez obzira na FPS. Vrijednost od shake_strength kroz vrijeme od proslijeđene fade vrijednosti se približava vrijednosti 0, ali ju nikad ne postiže. Stoga shake_strength obavezno moramo postaviti na 0 nakon što postigne vrijednost 0.1 jer inače pada u beskonačnost i nikad se ne zaustavi. Camera2D čvor ima offset svojstvo koji mijenja

svoju poziciju unutar trenutne koje se nalazi, odnosno napravi se pomak u odnosu na trenutnu poziciju bez mijenjanja trenutne pozicije (eng. *Superimpose*).

```
# Od: Gwizz [33]
# Dostupno na: www.youtube.com/watch?v=LGt-jjVf-ZU

extends Camera2D

enum SHAKE {
	TOUCH, LIGHT, MEDIUM, STRONG, DEVASTATING
}

var shake_types = {
	SHAKE.TOUCH: {"strength": 5, "fade": 16.0},
	SHAKE.LIGHT: {"strength": 8, "fade": 14.0},
	SHAKE.MEDIUM: {"strength": 10, "fade": 13.0},
	SHAKE.STRONG: {"strength": 12, "fade": 10.0},
	SHAKE.DEVASTATING: {"strength": 22, "fade": 9.0}
}

var rng = RandomNumberGenerator.new()
var shake_strength: float = 0 # Vece je jače
var shake_fade: float = 10.0 # Vece je kraće

func _ready():
	# Inace bi bilo V2(0,0) jer je kamera autoload
	position = Vector2(288, 512)

func apply_shake(shake_type: SHAKE = SHAKE.LIGHT):
	var shake_settings = shake_types[shake_type]
	shake_strength = shake_settings.strength
	shake_fade = shake_settings.fade

func _process(delta):
	if shake_strength > 0:
		shake_strength = lerpf(
			shake_strength, 0, shake_fade * delta
		)
		offset = random_offset()
		if shake_strength < 0.1: shake_strength = 0.0
		# Inace zauvijek lagano titra jer je 0.00...1

func random_offset():
	return Vector2(
		rng.randf_range(-shake_strength, shake_strength),
		rng.randf_range(-shake_strength, shake_strength)
	)
```

Da bi sad koristili podrhtavanje bilo kad nam je potrebno koristimo dolje navedeni kod:

```
CameraShake.apply_shake(CameraShake.SHAKE.DEVASTATING)

CameraShake.apply_shake(CameraShake.SHAKE.TOUCH)
```

Vidi se kroz gore navedeni isječak da jednostavno samo treba pozvati Autoload CameraShake i proslijediti preddefiniranu enumeraciju iz CameraShake sa jačinom koja je apstrahirana iza smislenih riječi, a ne brojeva koji nam ništa ne znače.

7. Primjer praktičnog rada

Kao primjere praktičnog rada, prikazat ćemo kako se sve do sad navedene ključne riječi, izrazi i čvorovi mogu koristiti u zajedništvu za kreiranje odabranih dijelova igre. Ova praktična primjena će pomoći razumjeti kako se svi aspekti Godota međusobno komuniciraju i rade kako bi se postigao određeni rezultat.

7.1. Lopta

Naravno, kao glavni dio ove Breakout (eng. *Brick Breaker*) stila igre, cilj je slomiti objekte uz pomoć usmjerene loptice. Loptica nije tradicionalna loptica, u smislu da ona ne smije na sebe primati impulse, odnosno ne smije mijenjati svoju brzinu, samo smije mijenjati smjer. Razlog tomu je da ako pod određenim kutom unutar određenih uvjeta uhvatimo lopticu, ona se može zaustaviti, što naravno ne želimo. Isto tako ako loptica poprima i primjenjuje impulse ona s dovoljno vremena može izgubiti svoju energiju što će opet rezultirati usporenom ili mirnom lopticom. Zbog toga ne možemo koristiti integrirati čvor RigidBody2D koji unutar sebe ima neisključive impulse koji nam smetaju. Rješenje problema je da se kreira vlastita loptica koja ignorira gotovo pa sve zakone fizike da bi dobili rezultate koje želimo.

Za kreiranje loptice, odabrali smo čvor: CharacterBody2D. Loptica naravno mora imati definiranu brzinu koja je konstantna osim ako ju ne promijenimo unutar igre prilikom nekih dodatnih efekata.

```
extends CharacterBody2D

var speed: float = 650

func _ready() -> void:
	launch()

func launch() -> void:
	velocity = global_transform.y * speed
```

Gore navedeni kod kreira lopticu s jednostavnom konstantnom brzinom. Trenutno fali delta koju ćemo dodati pri prvoj koliziji jer tek onda trebamo računati kuteve za odbijanje čime ćemo moći koristiti i move_and_collide metodu koja je dizajniranja za kretanje koje je nama potrebno.

Unutar _physics_process metode, kreira se logika za koliziju koju ćemo onda proslijediti dalje ovisno o tomu ako se desila kolizija.

Svojstvo velocity koje se koristi za definiranje brzine nije samo brzina, velocity je tip podataka Vector2 koji pokazuje trenutni smjer kretanja. Kao primjer, ako imamo dane vektore: $\vec{a}$ i $\vec{b}$. Gdje je vektor $\vec{b}$ 4 puta veći kao što je prikazano dolje:

$$\vec{a} = \begin{bmatrix} 1 \\ 1 \end{bmatrix},\ \vec{b} = \begin{bmatrix} 4 \\ 4 \end{bmatrix}$$

Ako se onda ti vektori postave kao velocity loptice. Loptica se onda 4 puta brže kreće istom putanjom preko vektora $\vec{b}$ u odnosu na vektor $\vec{a}$. Možemo si zamisliti da se velocity ponaša kao brizna kretanja, ali i kao usmjerenje kretanja.

```
func _physics_process(delta: float) -> void:
	var collision = move_and_collide(velocity * delta)
	if not collision: return

	var normal = collision.get_normal()
	collision_logic(collision, normal)
```

Vidi se kako smo sada dodali deltu pri prvoj koliziji sa nekim objektom koji sadrži CollisionShape2D čvor. Da bi se moglo odrediti što dalje napraviti sa kolizijom, potrebno nam je dohvatiti koliziju od metode move_and_collide te i trenutnu normalu loptice iz move_and_collide za računanje kuta odbijanja od ostalih objekata.

```
# Obradimo koliziju iz _physics_process metode
func collision_logic(collision, normal):
	var collider = collision.get_collider()
	if collider == null: return

	if collider.is_in_group("Paddle"):
		global_position = global_position + Vector2(0, -5)
		handle_paddle_collision(normal)
		CameraShake.apply_shake(CameraShake.SHAKE.MEDIUM)
	elif collider.is_in_group("Wall"):
		handle_wall_collision(normal)
		CameraShake.apply_shake(CameraShake.SHAKE.TOUCH)
	elif collider.is_in_group("Brick"):
		handle_wall_collision(normal)
		CameraShake.apply_shake(CameraShake.SHAKE.TOUCH)
		hit_signal.emit(collider, "Brick", global_position)
```

Vidi kako često znamo postavljati uvijete koji vraćaju samo return bez podataka. To je zbog toga što postoji mogućnost da vrijednost još nemamo te bi počeli raditi operacije nad ili sa podatkom kojega nemamo što bi rezultiralo sa rušenjem aplikacije.

Također se vidi da se unutar ove metode određuje jačina podrhtavanja kamere ovisno o grupi sa kojom se kolizija desila preko po CameraShake Autoload.

Unutar collision_logic metode vidi se da prvo kreiramo varijablu collider koji dobi podatke iz collision argumenta da dobimo njegova svojstva. Svojstvo koje je nama izuzetno važno su Grupe. Kao što je prije bilo navedeno, grupe su nam izuzetno korisne da mi na jednostavni način saznamo nešto ili javimo prema većoj skupini čvorova. U ovom slučaju,

samo nas zanima s kojim tipom objekta smo se mi trenutno sudarili. Vide se kreirani upiti za grupe: Paddle, Wall i Brick. Ovisno o grupi proslijedimo podatke normale iz argumenta od collision_logic metodi prema određenoj metodi za obradu odbijanja odnosno prema metodi handle_paddle_collision ili handle_wall_collision koje su prikazane u dolje navedenom kodu:

```
func handle_paddle_collision(normal):
	if normal.dot(Vector2.UP) > 0.0:
		if normal.dot(Vector2.UP) == 1.0:
			var random_angle = deg_to_rad(randf_range(-1, 1))
			normal = normal.rotated(random_angle)
	else:
		print("PADDLE SIDE HIT")

	velocity = velocity.bounce(normal)

func handle_wall_collision(normal):
	velocity = velocity.bounce(normal)
```

Kod prije navedenog koda, mi samo postavimo trenutnu brzinu (velocity) na velocity.bounce(normal) koji nam od velocity za upadni kut da kut odbijanja. Ovisno o tomu ako je grupa tipa Paddle ili Brick i Wall koji vode do iste metode. Vidimo kako unutar logike za obradu paddle kolizije da imamo detekciju ako se loptica udarila sastrane po platformi. Isto tako dodamo malu varijaciju preciznosti uz upotrebu laganog random_angle koji varira trenutni kut putanje odnosno velocity svojstva.

Razlog razdvajanju logike obrade na više metoda je da ako se u budućnosti želi dodatno razgraditi kolizija ili se povezati na neke druge metode koje se pokreću unutar istih ili drugih uvjeta. Pritom imamo spremni temelj za nadogradnju.

7.2. Komponenta za detekciju pokreta prstom

Da bi tijekom igranja igre korisničko sučelje bilo čisto, trebali bi implementirati detekciju određenog pokreta prstom na ekranu (eng. *Swipe*). Za to treba Area2D čvor kojeg ćemo pretvoriti u ponovno upotrebljivu komponentu za daljnje korištenje.

Glavni razlog za kreiranje prostora za detekciju određenih pokreta prstima (eng. *Swipe gestures*) je da maknemo gumb za pauziranje i izlaz iz igre izvan korisničkog sučelja tijekom igre. Gumb za pauziranje i izlaz iz igre ne radi ništa tijekom igranja. Stoga bi bilo najbolje da se makne i zamjeni sa “nevidljivim” gumbom koji se preko prirodnog pokreta prstima pokreće te otkriva panelu s više opcija odnosno putanja. Također pri otkrivanju panele imamo priliku predstaviti informacije korisniku koje nismo prije mogli zbog limitiranog prostora.

Za kreiranje komponente za detekciju određenih pokreta prsta, prvo moramo kreirati komponentu s root tipom Area2D. Također moramo kreirati i enumeracije za @export bilješku, te i signal za emitiranje. Bilješka i argumenti koji se prosljeđuju signalu su tipa SwipeDirection, odnosno tipa podataka naše specifične enumeracije.

```
# Gesture Swipe Detection Component
extends Area2D

signal swipe_gesture_signal(swipe_direction: SwipeDirection)

var start_position = Vector2.ZERO

@export var swipe_direction: SwipeDirection = SwipeDirection.LEFT_SWIPE

enum SwipeDirection {
        LEFT_SWIPE, RIGHT_SWIPE, UP_SWIPE, DOWN_SWIPE,ALL
}
```

Također smo dodali i početnu poziciju koju smo postavili na Vector2(0,0). Unutar inspektora naše komponente sada imamo i @export bilješku swipe_direction koja služi za odabir kojeg točno pokreta želimo detektirati za instancu komponente. Naravno možemo i bilješku postaviti na “ALL” da detektiramo sve moguće pokrete prstom.

Za detekciju inputa potrebni nam je signal_on_input_event koji dolazi ugrađeni u Area2D čvor:

```
func _on_input_event(viewport, event, shape_idx):
        if event is InputEventScreenTouch:
                if event.pressed:
                        start_position = event.position
                else:
                        handle_swipe(event.position)
```

Kad se desi događaj event tipa InputEventScreenTouch, te da se pritom ekran drži pritisnutim preko uvjeta if event.pressed == true. Onda se postavi početna pozicija start_position na event.position, pritom ako prestanemo držati ekrana, pretpostavimo da smo završili sa pokretom prsta i pošaljemo metodi handle_swipe krajnju poziciju, odnosno end_position našeg pokreta prstima.

```
func handle_swipe(end_position) -> void:
        var swipe_distance = end_position - start_position

        # Left swipe detection
        if (swipe_distance.x > 60 and
                abs(swipe_distance.y) < swipe_distance.x):
                if (swipe_direction == SwipeDirection.LEFT_SWIPE or
                        swipe_direction == SwipeDirection.ALL):
                        swipe_gesture_signal.emit(SwipeDirection.LEFT_SWIPE)
        # Right swipe detection
        elif (swipe_distance.x < -60 and
                abs(swipe_distance.y) < abs(swipe_distance.x)):
                if (swipe_direction == SwipeDirection.RIGHT_SWIPE or
                        swipe_direction == SwipeDirection.ALL):
                        swipe_gesture_signal.emit(
                                SwipeDirection.RIGHT_SWIPE
                        )
```

```
# Up swipe detection
elif (swipe_distance.y > 60 and
        abs(swipe_distance.x) < swipe_distance.y):
        if (swipe_direction == SwipeDirection.UP_SWIPE or
                swipe_direction == SwipeDirection.ALL):
                swipe_gesture_signal.emit(SwipeDirection.UP_SWIPE)
# Down swipe detection
elif (swipe_distance.y < -60 and
        abs(swipe_distance.x) < abs(swipe_distance.y)):
        if (swipe_direction == SwipeDirection.DOWN_SWIPE or
                swipe_direction == SwipeDirection.ALL):
                swipe_gesture_signal.emit(SwipeDirection.DOWN_SWIPE)
```

Za gore navedeni kod prosljeđuje se krajnja pozicija dodira preko end_position argumenta gdje onda pogledamo sve moguće kombinacije za odabranu postavku @export bilješke koji se točno pokret desio gdje je minimum piksela koji moramo sa prstom proći po x, y osi jednaki 60 piksela. Drugi dio uvjeta gleda u kojem smjeru se dešava pokret prstom. Kao primjer, uzmimo smjera prstom prema lijevoj strani. Uvjet: abs(swipe_distance.y) < swipe_distance.x osigurava da je okomito pomicanje po y osi manje od horizontalnog pomaka x osi. To nam govori da je povlačenje prsta pretežno vodoravno, a ne dijagonalno ili okomito.

Za kreiranje instance ove komponente samo joj se mora kao dijete definirati CollisionShape2D i kreirati područje unutar kojeg se želi da se povlačenje prsta detektira.

8. Kreiranje resursa za igru

Razvoj igre uključuje više od samo programiranja. Razvoj zahtijeva razne resurse, uključujući zvučne i vizualne efekte. Dva ključna alata za programiranje 2D igara su JSFXR za zvučne efekte i Aseprite za stvaranje sličica (eng. *Sprite*), najčešće sličica niske rezolucije da se dobije pikselizirani stil.

8.1. Kreiranje zvučnih efekata pomoću alata JSFXR

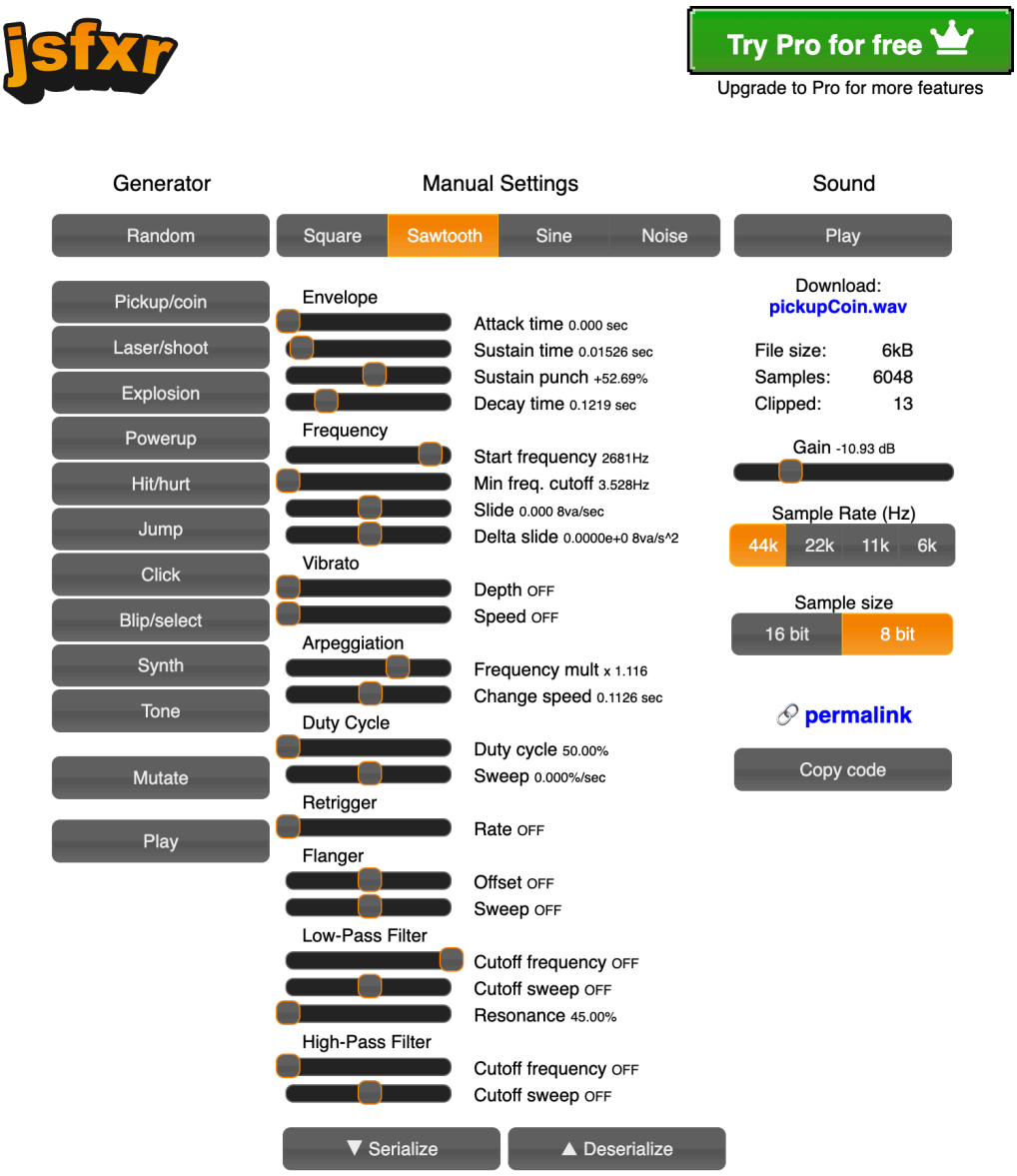
JSFXR je JavaScript port popularnog alata SFXR. DRPetter je 2007 godine kreirao SFXR kao jednostavni i intuitivni alat za generiranje zvučnih efekata za jednostavne igre [29]

Alat je primarno bio dizajnirani za takozvane Game Jams koji su globalni događaji koji traju između 24 sata i tjedan dana, ovisno o događaju. Gdje je cilj da se unutar izuzetno kratkog vremena napravi igra unutar određenih pravila. Pravila mogu biti svakakva, na primjer događaj igre mora obavezno sadržavati mačke ili pravilo može biti da se radnja mora obavezno događati na vlaku. Takva stroga pravila potiču maštu i kreativnost.

Njegova izvorna svrha bila je pružiti jednostavno sredstvo za ubacivanje osnovnih zvučnih efekata u igru za one ljude koji su naporno radili kako bi svoje unose obavili unutar 48 sati i nisu imali vremena trošiti na traženje prikladnih načina za to. Ideja je da ljudi mogu samo pritisnuti nekoliko gumba u aplikaciji i dobiti neke uglavnom nasumične efekte koji su prilagođeni u smislu da korisnik može prihvatiti/odbiti svaki predloženi zvuk. [29]

U najjednostavnijem smislu, aplikacija radi tako da pritisnemopar slučajnih gumba te se ih stišće tako dugo dok se ne čuje zvuk koji nam se sviđa ili tražimo. Ako pronađemo zvučni efekt koji nam se sviđa, ali još nije točno ono što tražimo, možemo ga ručno namještati da paše našemu slučaju korištenja. To rješava više problema. Prvi problem je da se ne očekuje od korisnika nikakvo tehničko znanje o zvuku i obradi zvuka, pritom vrlo jednostavno pronađemo i kreiramo zvuk. Drugi problem koji se rješava je da ako nađemo zvuk koji nam se sviđa, a imamo nekakvo tehničko znanje, možemo ga manipulirati unutar aplikacije bez problema. Zadnji problem koji se rješava je brzina kojom možemo pronaći zvuk. Naime, licence za zvučne efekte su relativno skupe i najčešće ih se mora kupiti kao skup više zvučnih efekta, a ne kao individualne zvučne zapise, pritom je i dosta veliki problem pronaći zvučne efekte koji nam se sviđaju.

JSFXR radi na istome principu kao i SFXR, jedina razlika je što je open source web aplikacije, a ne desktop aplikacija.



Slika 42: JSFXR web aplikacija [30]

Vidi se kako je na prikazanoj slici 42. Prikazano puno klizača i gumba za kreiranja zvučnih efekata, također vidi se i (eng. Sample rate) koji označuje kvalitetu zvučnog efekta odnosno broj ciklusa u sekundi. Te i količina bitova koji se koriste za preciznost definiranja audio zapisa. Veće vrijednosti stope uzorkovanja i veličine uzorka (eng. Sample size) rezultiraju većom datotekom.

Također, vidi se kako i sa lijeve strane ima više gumba kao što su: Pickup/coin, Laser/shoot... Oni odabiru već preddefinirane pojase za klizače da kreiraju zvuk unutar preddefiniranog područja za tu riječ kojoj smo mi kroz više godina korištenjem interneta, igranja video igara i slično, asocirali zvuk sa tom riječju. Zbog toga nam je svima dobro poznati zvuk skupljanja novčića ili postizanja novog nivoa unutar video igre. Tu iskorištavamo taj koncept da brzo i jednostavno dobijemo zvuk koji svi asociramo sa nekom riječju ili

radnjom. Odabirom zvuka on se reproducira, ako smo zadovoljni sa zvukom možemo ga spremiti ili modificirati. Naravno možemo i povećati ili smanjiti stopu uzorkovanja i veličinu uzorka. Svi zvučni efekti koji su korišteni unutar rada su kreirani preko JSFXR.

8.2. Kreiranje vizualnih resursa pomoću alata Aseprite

Izdan 2014. godine, Aseprite je program koji programerima video igara i animatorima omogućuje stvaranje sličica i umjetnosti za svoje projekte koji su bazirani na takozvanom pixel art stilu. Sadrži sučelje grafičkog dizajna za stvaranje pikselne umjetnosti s nekoliko kistova, prilagodljivom paletom boja i načinima stapanja. Također sadrži skup standardnih alata za animaciju, uključujući vremensku traku, označavanje ključnih sličica i ostalo. [31]

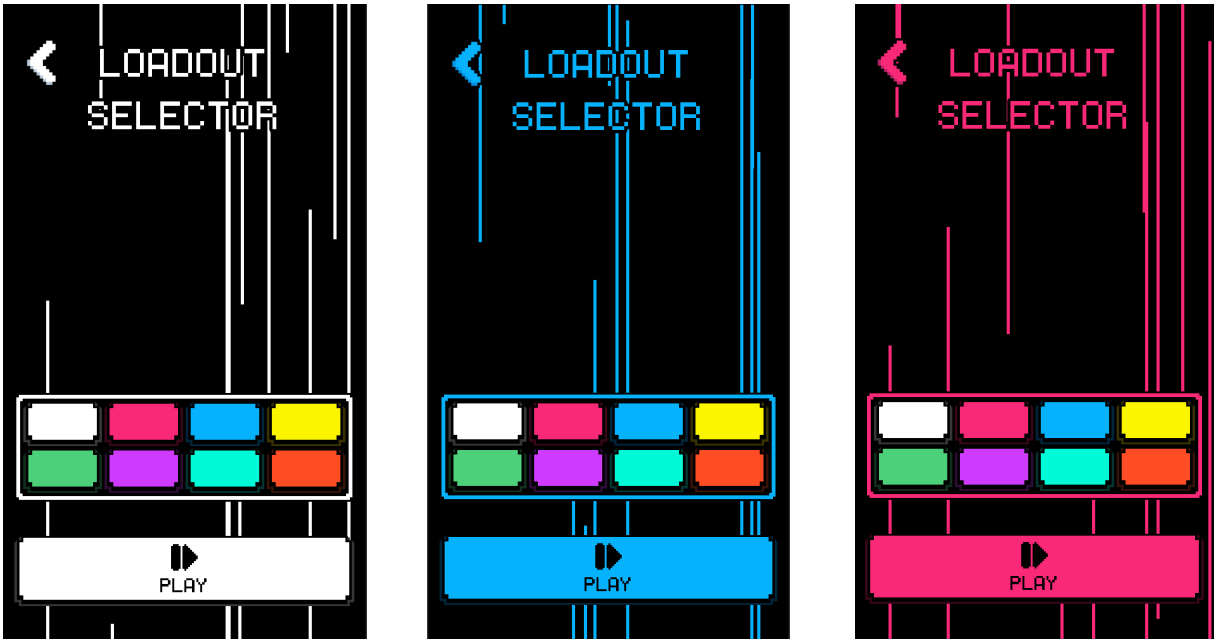
8.2.1. Kreiranje našeg stila

Izuzetno je važno kreirati jedinstveni stil igre. Za ovaj rad odabrao sam kreirati retro (eng. *Pixel art*) stil. Pixel art je fascinantan spoj umjetnosti, tehnologije i nostalgije. Pixel art, sa svojom jednostavnošću, ima moć vratiti igrače na same početke video igara sa modernom tehnologijom, te istovremeno nudi prostor za modernu kreativnost i inovaciju.

Pixel art je stil digitalne umjetnosti gdje se slike ručno i precizno kreiraju piksel po piksel. Pixel art stil potječe iz ranih dana video igara, gdje su hardverska ograničenja zahtijevala upotrebu malih razlučivosti ekrana, ograničenih paleta boja i memorije. Unatoč tim ograničenjima, ili možda upravo zbog tih ograničenja, pikselna umjetnost je opstala i razvila se kao retro nostalgični stil, zadržavajući značajno mjesto na području vizualnog dizajna igara. Zbog svoje jednostavne naravi, bilo tko može dizajnirati pixel art igru. Kao neka osnova, pixel art sebe ograničava određenim dimenzijama koje su izuzetno niske rezolucije. Najčešće dimenzije su 16x16, 32x32 ili 64x64. Postoje naravno i veće rezolucije, ali i one su izuzetno niske “kvalitete” u odnosu na modernu rasteriziranu ili vektorsku kvalitetu slika koje su višestruke ili pa beskonačne kvalitete što se tiče grafike bazirane na vektorima. Još jedno ograničavanje pixel art stila je paleta boja. Paleta boja kod modernih slika najčešće sadržava preko 1 000, ako ne i 100 000 boja unutar jedne slike. Pixel art sebe namjerno ograničuje na samo par boja, najčešće do 32 boje. Takvo ograničenje forsira umjetnika da kreira slike koje imaju određenu “harmoniju” i stil. Sve slike koje se unutar video igre onda koriste, sadrže identični stil i paletu, što doprinosi povezanom vizualnom stilu.

Za postizanje izuzetno jednostavnog stila igre, unutar igre koriste se samo 2 boje: bijela sa hex oznakom #FFFFFF i crna sa hex oznakom #000000. Naravno s transparentnošću. Moja igra uz upotrebu samo bijele i crne boje ne bi bila baš interesantna jer se gubi mogućnost sjena i osvjetljavanja. Boja postoji ili ne postoji, nema između. Da bi se riješili monotonosti, a opet zadržali izuzetno jednostavni možemo koristiti modulaciju boja. Unutar Godota, svaki čvor koji je naslijeđeni od Node čvora sadrži svojstvo “modulate”.

Svojstvo modulate modulira boje koje se nalaze unutar tog čvora na neku drugu boju. Ako se koristi čista bijela boja, odnosno hex kod #FFFFFF, onda pomoću modulacije, tu boju pretvorimo u bilo koju drugu boju želimo. Naravno s obzirom na to da je crna boja nedostatak boje, ona se ne modulira na neku drugu boju jer ostaje crna. Ako su nam sve boje unutar igre bijele, onda mi imamo mogućnost modulacije te boje na bilo koje druge želimo tijekom izvršavanja aplikacije. Ovo rješava problem monotonosti. Korištenjem samo 2 boje smo izuzetno jako ograničeni u našoj kreativnosti. Problem monotonosti rješava tako da korisniku pružimo mogućnost mijenjanja boje na set predodređenih boja. Pritom se kreira jedinstveno korisničko iskustvo, jer gotovo niti jedna video igra ne dopušta mijenjanje izgleda korisničkog sučelja. Čim se korisniku dopusti da mijenja nešto što inače ne mogu, oni osjećaju određenu količinu zadovoljstva i vlasništva, iako je samo mijenjanje izgleda korisničkog sučelja na par preddefiniranih boja.



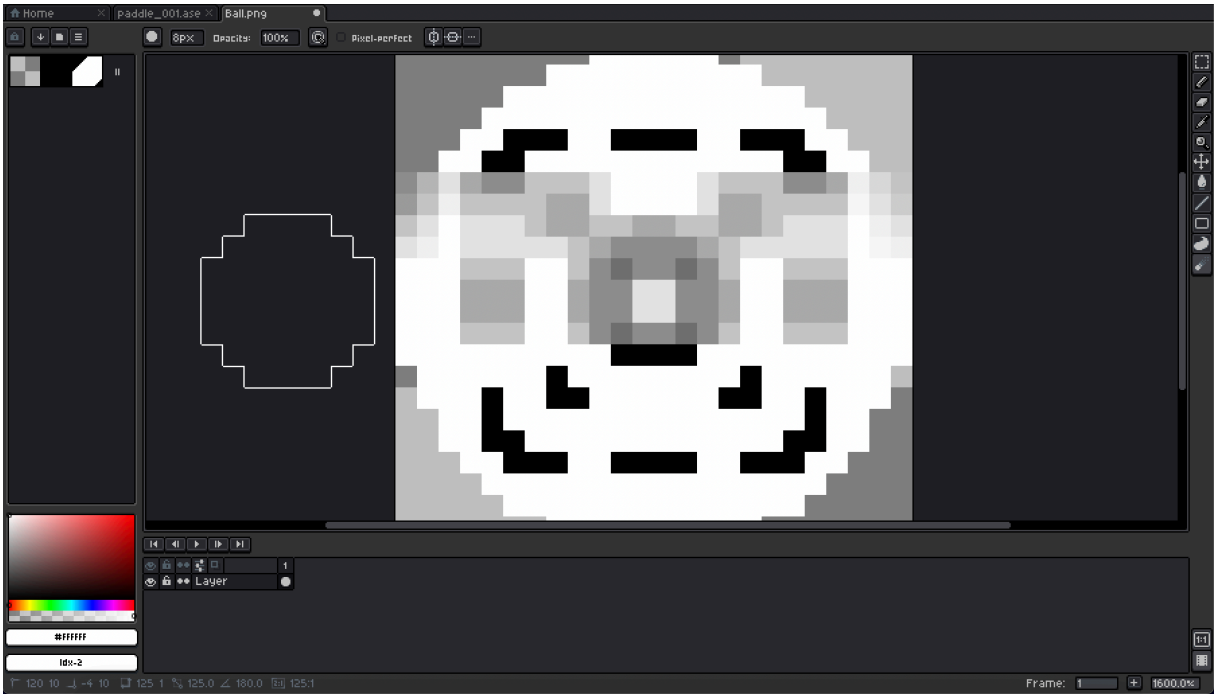
Slika 43, 44, 45: Prikaz mijenjanja boja korisničkog sučelja igre

Mijenjanje boja ne samo da se odnosi na korisničko sučelje, već se prenosi i na samo igru. Time se postiže određena kohezija unutar igre i znatno pridonosimo ukupnom korisničkom iskustvu. Sve boje se prilikom promjene se spremaju lokalno na uređaju tako da ostaju i nakon gašenja igre.

GDScript nudi apstrakciju za pristupa datotečnom sustavu koji je neovisni o operacijskom sustavu na kojemu se nalazi. Prilikom ulaska u igru, pročita se vrijednosti unutar settings.cfg da dobijemo vrijednost boje koje je korisnik zadnje odabrao i postavimo igru i korisničko sučelje na odabranu boju. Promjenom boje igre prebrisujemo spremljenu boju na uređaju za daljnje korištenje.

8.2.2. Aseprite

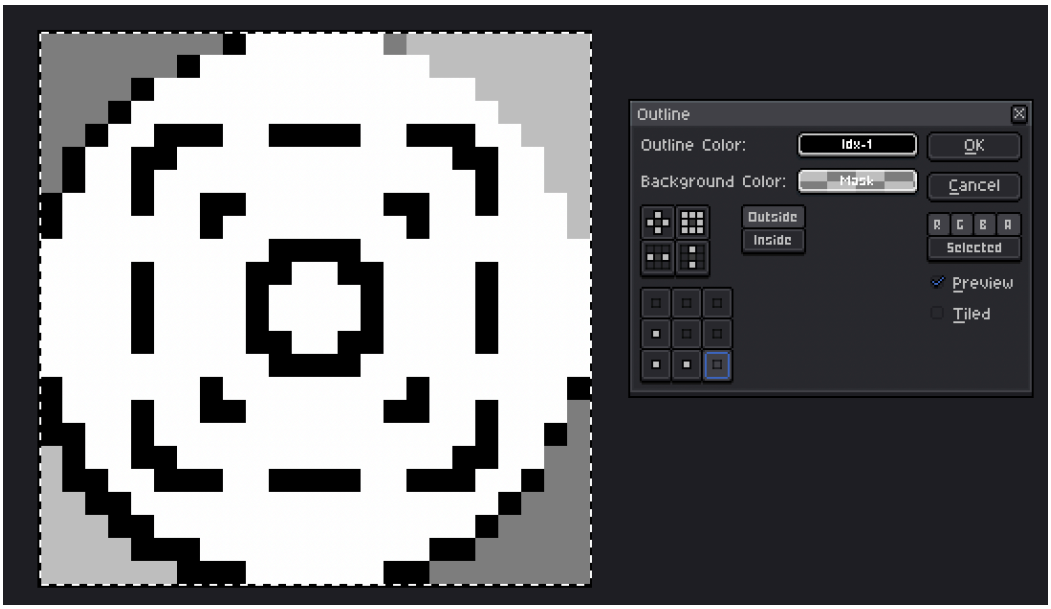
Izuzetno moćan i namjenski kreirani pixel art alat je Aseprite (Allegro Sprite Editor). Kreirani 2014. godine. Aseprite je jedan od najpopulanijih alata za kreiranje pixel art digitalne umjetnosti. [31]



Slika 46: Prikaz korisničkog sučelja Aseprite [32]

Aseprite sadrži više različitih alata za kreiranje izgleda kojeg želimo. Osnovni alat kojega koristimo je kist. Također imamo i alate za brisanje, fill, odabir boja, kreiranje selekcije, blur, jumble i ostale alate. Jumble alat nam samo premješta trenutne piksele iznad kojih se jumble kist nalazi, pritom kreira prirodni nered. Jumble je alat koji se generalno ne koriste kod tradicionalne rasterizirane grafike visokih rezolucija jer nisu potrebni. No oni su izuzetno korisni kod pixel art stila digitalne grafike jer nam znatno mogu ubrzati rad u kreiranju slika.

Aseprite omogućuje i korištenje jednostavnih efekata. Najkorisniji efekti unutar našeg rada je efekt outline. Koji kreira obrub za cijelu trenutnu sliku ili selekciju unutar slike. Obrub nam služi da dodamo dubinu, osvjetljenje i sjene slici. Unutar igre primarno služe za diferencijaciju između raznih slika jer su sve sličice iste boje, stoga je potreban crni obrub da se vidi rub između raznih preklapajućih slika. Outline efekt izuzetno brzo i jednostavno kreira prirodni obrub koji može kreirati efekt osvjetljenja unutar igre, bez potrebe za korištenjem resursno intenzivnog osvjetljenja.



Slika 47: Prikaz kreiranja obruba na slici

9. Zaključak

Unutar ovog rada, cilj je bio kreirati kompletnu video igru s pomoću Godot Engine alata, s posebnim naglaskom na kreiranja dobrog korisničkog sučelja i korisničkog iskustva. Uz Godot, također su se koristili alati JSFXR za kreiranje zvučnih efekata za reprodukciju, te i Aseprite za kreiranje vizualnih resursa za video igru.

Godot je razvojno okruženje koji se temelji na konceptu čvorova i scena. Kroz korištenjem različitih čvorova i scena u zajedništvu mogu se kreirati komponente koje služe određenoj svrsi ili rješavaju određeni problem. Zbog naravi scena i čvorova, unutar Godot razvojnog okruženja jako se potiče praksa kompozicije u odnosu na nasljeđivanje.

Godot sam po sebi ima dosta problema. Glavni razlog problema je što je Godot projekt otvorenog koda te je koordinacija za rješavanje problema relativno problematična. Često se znaju desiti greške gdje poruka pogreške sugerira da se greška desila unutar Godota, a ne našeg koda. Također, aplikacija se kod kompleksnijih zahtjeva često zna rušiti. Bez obzira na to, Godot je izuzetno kvalitetno okruženje za kreiranje video igara bez puno muka. Sa svojim intuitivnim načinom razmišljanja o strukturi scena.

Također se pokazala i otvorena strana Godot zajednice uz upotrebu kompleksnih shader koda. Gotovo pa da je nemoguće pronaći resurse i informacije koje nisu besplatne i otvorene za korištenje, modifikaciju i daljnju distribuciju. Zajednica koju je Godot stvorio je nešto što se rijetko viđa

Unutar aplikacije koja je priložena uz rad, posebno sam se fokusirao na kreiranju inovativnog korisničkog sučelja i korisničkog iskustva uz upotrebom boja i intuitivnih pokreta, gdje svaki korisnik može video igru dodatno prilagoditi sebi. Iako je to relativno mala stvar, ona u ukupnom zadovoljstvu korisnika izuzetno puno znači.

Popis literature

[1] Mike, “The Evolution/History of the Godot Game Engine,” GameFromScratch.com. Pristupljeno: 26. lipanj., 2024. [Online]. Dostupno na: https://gamefromscratch.com/the-evolution-history-of-the-godot-game-engine/

[2] “History of the computer game.” Pristupljeno: 27. lipanja 2024. [Online]. Dostupno na: https://www.jesperjuul.net/thesis/2-historyofthecomputergame.html

[3] “I, Robot (1984),” TV Tropes. Pristupljeno: 28. lipnja 2024. [Online]. Dostupno na: https://tvtropes.org/pmwiki/pmwiki.php/VideoGame/IRobot1984

[4] T. W. published, “The PC games market grew a lot more than the console games market last year, says research firm,” *PC Gamer*, May 20, 2024. Pristupljeno: 28. lipnja 2024. [Online]. Dostupno na: https://www.pcgamer.com/gaming-industry/2023-pc-games-revenue-increase-newzoo/

[5] “iPhone vs. Android User & Revenue Statistics (2024),” Backlinko. Pristupljeno: 28. lipnja 2024. [Online]. Dostupno na: https://backlinko.com/iphone-vs-android-statistics

[6] “Discover the Best Phone Aspect Ratio & Screen Resolution [2024 Guide].” Pristupljeno: 28. lipnja 2024. [Online]. Dostupno na: https://www.aiseesoft.com/resource/phone-aspect-ratio-screen-resolution.html

[7] E. Belinski, “Resolution by iOS device — iOS Ref.” Pristupljeno: 28. lipnja, 2024. [Online]. Dostupno na: https://iosref.com/

[8] “What is GDExtension?,” Godot Engine documentation. Pristupljeno: 6. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html

[9] “Understanding tree order :: Godot 3 Recipes.” Pristupljeno: 28. lipnja 2024. [Online]. Dostupno na: https://kidscancode.org/godot_recipes/3.x/basics/tree_ready_order/index.html

[10] “GDScript reference,” Godot Engine documentation. Pristupljeno: 28. lipnja, 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html

[11] “Using signals,” Godot Engine documentation. Pristupljeno: 29. lipnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/getting_started/step_by_step/getting_started/step_by_step/signals.html

[12] “Singletons (Autoload),” Godot Engine documentation. Pristupljeno: 29. lipnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/scripting/tutorials/scripting/singletons_autoload.html

[13] “how can i abreaviate large numbers? - Godot Forums.” Pristupljeno: 23. lipnja 2024. [Online]. Dostupno na: https://godotforums.org/d/35589-how-can-i-abreaviate-large-numbers

[14] “Tween,” Godot Engine documentation. Pristupljeno: 20. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/classes/classes/class_tween.html

[15] “Godot tweening cheat sheet : r/godot.” Pristupljeno: 20. srpnja 2024. [Online]. Dostupno na: https://www.reddit.com/r/godot/comments/frqzup/godot_tweening_cheat_sheet/

[16] “Resources,” Godot Engine documentation. Pristupljeno: 15. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/scripting/resources.html

[17] “Groups,” Godot Engine documentation. Pristupljeno: 15 . srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/scripting/tutorials/scripting/groups.html

[18] “Dictionary,” Godot Engine documentation. Pristupljeno: 16. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/classes/classes/class_dictionary.html

[19] “Canvas layers,” Godot Engine documentation. Pristupljeno: 17. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/2d/tutorials/2d/canvas_layers.html

[20] “Control,” Godot Engine documentation. Pristupljeno: 17. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/classes/classes/class_control.html

[21] “Container,” Godot Engine documentation. Pristupljeno: 17. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/classes/classes/class_container.html

[22] “Using the theme editor,” Godot Engine documentation. Pristupljeno: 18. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/ui/tutorials/ui/gui_using_theme_editor.html

[23] “9-slice scaling,” *Wikipedia*. Jul. 21, 2024. Pristupljeno: 19. srpnja 2024. [Online]. Dostupno na: https://en.wikipedia.org/w/index.php?title=9-slice_scaling&oldid=1235912011

[24] “WorldEnvironment,” Godot Engine documentation. Pristupljeno: 3. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/classes/classes/class_worldenvironment.html

[25] “Environment and post-processing — Godot Engine (stable) documentation in English.” Pristupljeno: 3. srpnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/3d/environment_and_post_processing.html

[26] “GPUParticles2D,” Godot Engine documentation. Pristupljeno: 20. lipnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/classes/classes/class_gpuparticles2d.html

[27] “Introduction to shaders,” Godot Engine documentation. Pristupljeno: 20. svibnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/shaders/tutorials/shaders/introduction_to_shaders.html

[28] “Audio buses,” Godot Engine documentation. Pristupljeno: 25. svibnja 2024. [Online]. Dostupno na: https://docs.godotengine.org/en/stable/tutorials/audio/tutorials/audio/audio_buses.html

[29] “DrPetter’s homepage - sfxr.” Pristupljeno: 1. lipnja, 2024. [Online]. Dostupno na: https://www.drpetter.se/project_sfxr.html

[30] jsfxr. Pristupljeno: Jul. 27, 2024. [Online]. Dostupno na: https://sfxr.me/

[31] “What is Aseprite?” Pristupljeno: 20. ožujka 2024. [Online]. Dostupno na: https://www.computerhope.com/jargon/a/aseprite.htm

[32] D. Capello, “Aseprite.” Pristupljeno: 20. ožujka, 2024. Dostupno: www.aseprite.org/

[33] Gwizz, *Godot 4 Camera Shake Tutorial*, (2023). Pristupljeno: 25. travnja, 2024. [Online Video]. Dostupno na: https://www.youtube.com/watch?v=LGt-jjVf-ZU

[34] Firebelly, “Create a Complete 2D Survivors Style Game in Godot 4,” Udemy. Pristupljeno: 5. Ožujka 2024 [Online]. Dostupno na: https://www.udemy.com/course/create-a-complete-2d-arena-survival-roguelike-game-in-godot-4/

[35] ombarus, “Pixelated Glitch PostEffect - Godot Shaders.” Pristupljeno: 12. ožujka, 2024. [Online]. Dostupno na: https://godotshaders.com/shader/pixelated-glitch-posteffect/

Popis slika