Izrada horor igre preživljavanja u programskom alatu Godot

Valentić, Antonio

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Antonio Valentić

IZRADA HOROR IGRE PREŽIVLJAVANJA U PROGRAMSKOM ALATU GODOT

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Antonio Valentić

Matični broj: 0016133923

Studij: Informacijsko i programsko inženjerstvo

IZRADA HOROR IGRE PREŽIVLJAVANJA U PROGRAMSKOM ALATU GODOT

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Mladen Konecki

Varaždin, rujan 2024.

Antonio Valentić

Izjava o izvornosti

Izjavljujem da je moj 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

U ovom pismenom radu proći ćemo kroz proces razvoja prototipa horor igre preživljavanja iz prvog lica u Godot pokretaču igara otvorenog koda. U tu svrhu, u prvom dijelu ovog rada proći ćemo kroz inspiracije za ovu igru, uključujući kratku povijest ovog žanra. Zatim ćemo obrazložiti različite dodatne programske alate koji su bili korišteni tijekom razvoja ove igre, koji su bili korišteni za stvaranje većine resursa igre. Tu ćemo također proći kroz osnovne koncepte rada u Godot pokretaču koji će biti potrebni za razumijevanje procesa razvoja igre.

U drugom dijelu rada ćemo opisati sam proces razvoja igre i u detalje razložiti različite elemente igre, kako su isti planirani i implementirani, te opis poteškoća koje su nastale tijekom razvoja. Na kraju ćemo imati pregled finalnog prototipa igre i potencijalnih poboljšanja u daljnjem razvoju istog.

Ključne riječi: algoritmi, programiranje, godot, razvoj videoigara, horror videoigra

Sadržaj

1. Uvod

Videoigre su jedna od najvećih sektora zabavne industrije, sa milijunima korisnika diljem svijeta, zapošljavajući ljude iz raznovrsnih disciplina. Od dizajnera i programer, sve od menadžera i distributera, objedinjujući umjetniče, programerske, menadžerske i ekonomske discipline kako bi se dobio finalni proizvod.

Jedan od najstarijih žanrova video igara su horor igre, sa svojim velikim rasponom podžanrova, uključujući titularni horor preživljavanja (eng. survival horror) koji implementira mnogo kompleksnije sustave igranja nego većina ostalih podžanrova horor igara. Kompleksne međupovezane razine, upravljanje inventarom stavki, konstantno procjenjivanje rizika i isplativosti konfrontacije s neprijatelja i slični elementi su razlog zašto su me uvijek privlačile videoigre ovog podžanra i koje su me motivirale da krenem rad na ovom projektu. Starost samog žanra je također utjecala na odabir estetike same igre.

Važnost ove tematike također leži i u sve više rastućoj sceni nezavisnih (eng. indie) videoigara na kojima rade vrlo mali timovi, a i nerijetko sami individualci, najčešće sa relativno skromnim budžetima, usprkos kojima uspijevaju stvoriti dojmljiva i nezaboravna iskustva. Posljednjih godina se na scenu besplatnih pokretača videoigara probio Godot, stekavši popularnost kao besplatan, lagan i moćan alat za nezavisne razvijače videoigara.

2. Metode i tehnike rada

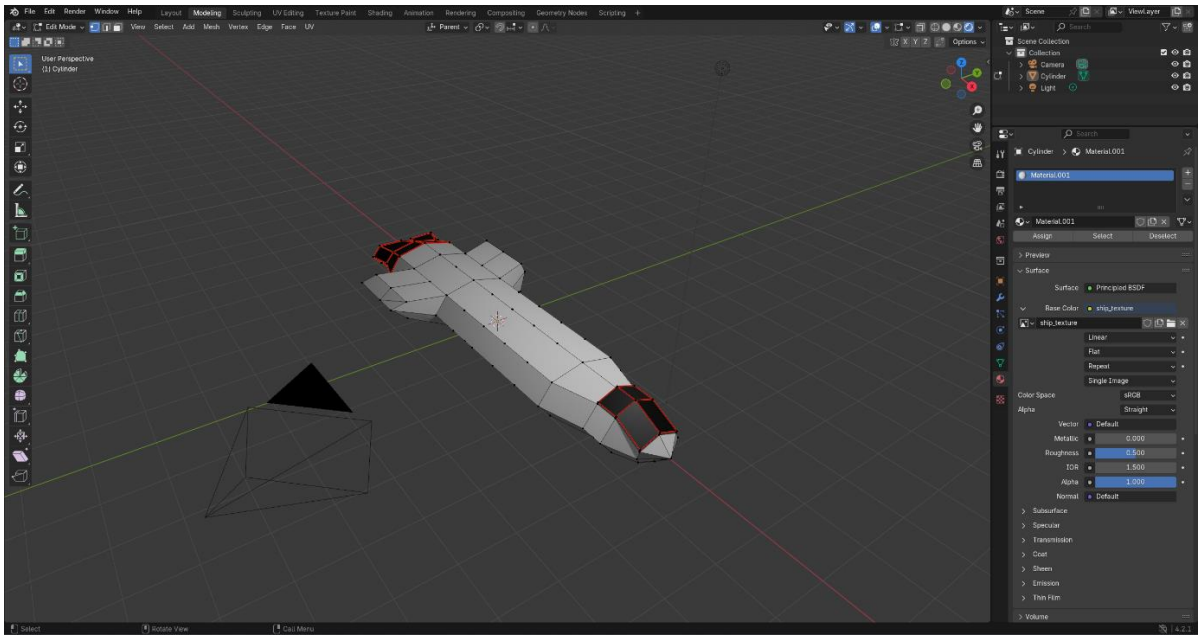
U sljedećim poglavljima ćemo opisati razne programske alate i aplikacije koje su korištene u razvoju ovog projekta. Tu uključujemo i programske alate za kreiranje resursa i razvojno okruženje za programiranje logike same videoigre.

2.1. Blender

Slika 1: Blender logo (Blender Foundation, 2019)

Blender je besplatan programski alat otvorenog koda za izradu 3D sadržaja koji se koristi u raznim industrijama, uključujući razvoj videoigara. On nudi sveobuhvatan skup alata za modeliranje, kreiranje tekstura i animiranje 3D objekata na ekranu, što ga čini svestranim rješenjem za programere koji žele stvoriti resurse za svoje videoigre. Blender podržava različite načine rada na 3D modelima, uključujući poligonalno modeliranje, kiparsko modeliranje (sculpting) i sjenčanje (shading) korištenjem virtualnih materijala, omogućujući i programeriam i umjetnicima da razviju detaljne i optimizirane resurse.

Blenderova uloga u razvoju igara značajna je zbog njegove pristupačnosti i mogućnosti izvoza resursa u različitim formatima kompatibilnim s različitim pokretačima videoigara kao što su Unity, Unreal Engine i naravno Godot. U ovom projektu smo koristili univerzalni glTF (GL Transmission Format) koji je kompatibilan sa većinom današnjih 3D programa i pokretačima videoigara (*glTF 2.0 - Blender 4.2 Manual*, 2024). Format se preporuča zbog svoje optimiziranosti za prikazivanje na ekran u stvarnom vremenu, bez da se gube infomacije o materijalima i plohama koji su potrebni za proces rasterizacije. Sama Godotova dokumentacija preporuča ovaj format, ako što je navedeno u (*Available 3D Formats*, 2024).



Slika 2: Primjer grafičkog sučelja u Blenderu (autorski rad)

2.2. Visual Studio Code

Slika 3: VS Code logo (*VS Code Icons and Names Usage Guidelines*, 2019)

Visual Studio Code je besplatni uređivač izvornog koda razvijen od strane tvrtke Microsoft, koji također služi kao integrirano razvojno okruženje za mnoge programske jezike. Zbog svojeg bogatog repozitorija besplatnih dodataka i nativne podrške za sustav za verzioniranje koda Git, Visual Studio Code je jedan od najkorištenijih uređivača koda, sa navodnih 70% programera koji tvrde da ga preferiraju u svom radu (Thorndyke, 2021).

Ja sam se odlučio za korištenje Visual Studio Code-a u programiranju logike igre, usprkos tome što Godot već ima svoj interni uređivač koda. Razlog tomu je što sam već uvelike upoznat s radom u Visual Studio Code-u, dijelom kroz moju edukaciju u Fakultetu Organizacije i Informatike, a dijelom kroz moju rastuću karijeru kao web developer. Godot dolazi sa svojim LSP (Language Server Protocol) serverom na koji se Visual Studio Code jednostavno povezuje koristeći Godotovo službeno proširenje (*Visual Studio Code*, 2024).

2.3. Aseprite

Slika 4: Aseprite logo (Capello, 2024)

Aseprite je uređivač slika primarno namijenjen za korištenje u svrhu kreiranja takozvanih „pixel art“ slika, što je vizualni stil koji koristi same piksele kao osnovne vizualne blokove. Zbog toga su to uglavnom slike vrlo niske rezolucije koje podsjećaju na grafičke stilove starih konzola za videoigre iz 1980-tih i 90-tih. Danas se on primarno koristi u svrhu stvaranja retro stilova koji imitiraju te stare grafičke stilove. U tu svrhu sam koristio ovaj program u ovom projektu, kako bi dizajnirao grafičko sučelje i teksture za 3D objekte koji su namjerno niske rezolucije kako bi imitirao stil ranih 3D konzola iz 90-tih.

Zbog svoje fokusa na taj stil, Aseprite ima mnogo ograničeniji set alata za crtanje nego što to imaju drugi konkurenti. Još jedna posebnost ovog programa je ta da se preuzimanje pokretačkog koda naplaćuje, ali sam izvorni kod je javno dostupan za korištenje i kompiliranje u privatne svrhe. Tako sam i ja preuzeo izvorni kod i prošao kroz proces kompilacije kao što je opisan u dokumentaciji njihovog Git repozitorija (Aseprite, 2024).

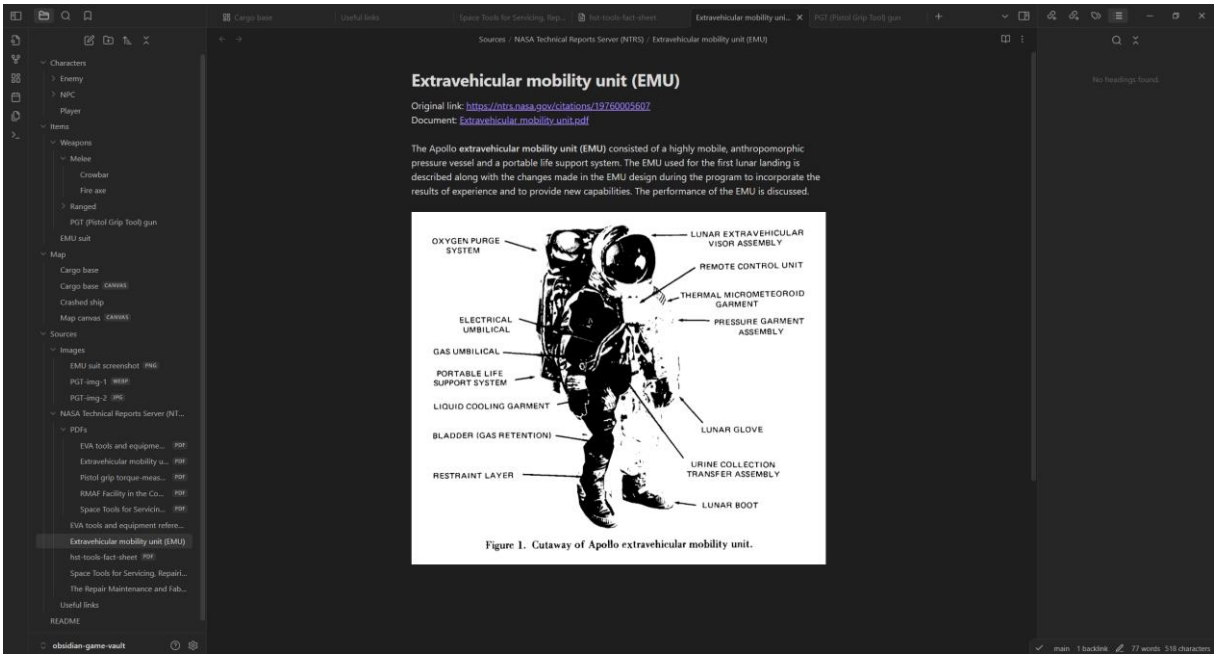
2.4. Obsidian

Slika 5: Obsidian logo (*Obsidian Brand Guidelines*, 2024)

Obsidian je besplatni programski alat za održavanje privatnih baza znanja i vođenje bilješki koji je u ovom projektu korišten za planiranje igre i njenih elemenata, te popisivanje inspiracija i drugih izvora znanja.

Ovaj alat mi je omogućio da na vrlo strukturiran način organiziram informacije, podatke i slike koje sam našao na NTRS (NASA Technical Reports Server) arhivi, glavnom repozitoriju javno dostupnih informacija vezanih uz rad NASA-e. U toj arhivi se mogu naći informacije poput patenata za različite van-planetarne alate, opreme u vozilima, odijelima i staništima koje su NASA astronauti koristili na dnevnoj razini kroz povijest.

Tu sam dobio inspiraciju za vizualni stil razina, neprijatelja i oružja koje ću koristiti u projektnoj igri, te vizualne reference koje su mi omogućile da te kreiram potrebne resurse u 3D i 2D aplikacijama koje sam prethodno naveo.



Slika 6: Grafičko sučelje Obsidiana (autorski rad)

3. Inspiracije

U ovoj sekciji rada analiziramo neke od inspiracija koje su bile izvor ideja tijekom razvoja projektne igre. Objasnit ćemo osnovne principe survival horror žanra koji je definirao većinu mehanika igre, te zatim horor žanr koji je inspirirao atmosferu igre i njen vizualni stil. Također je vrijedno napomenuti kako su science fiction elementi igre inspirirani stvarnim NASA-inim dokumentima i patentima koji su sadržavali detaljne skice alata i opreme.

3.1. Horor preživljavanja (Survival Horror)

Horor igre su uvijek imale svoje mjesto u povijesti razvoja video igara. U početku su to bile jednostavne igre gdje igrač izbjegava nepobjedivog neprijatelja. Ali s vremenom, kako su igre mehanički i grafički napredovale, razvile su se ove ideje u sceni horor igara. Svakako ne prvi, ali pionirski poduhvat je postigao Resident Evil 1996. godine koji je svojim financijskim uspjehom popularizirao žanr horora preživljavanja. Igrač više nije bio potpuno bespomoćan pred svojim nepobjedivim neprijateljima, već mu je igra dala alate da se može braniti. Paradoksalno, to je rezultiralo sa efektom koji je igračima dao veći osjećaj anksioznosti i straha nego horor igre gdje je igrač bespomoćan. Razlog tomu je što je sada igrač imao odgovornost da odluči da li želi riskirati borbu ili probati pobjeći od opasnosti, da li se isplati iskoristiti rijetke resurse ili ne. Taj dodatan sloj odlučivanja je glavni razlog zašto je ovaj žanr postao toliko popularan, jer time se obogaćuje horor sa elementom koji nije moguće postići u ni jednom drugom mediju osim u videoigrama (Fahs, 2009).

Slika 7: Resident Evil 1996. (Kasavin, 2007)

3.2. Kozmički horor H. P. Lovecrafta

Horor djela Howarda Phillipsa Lovecrafta poznata su po njegovoj specifičnoj vrsti horora, zvanoj kozmički horor. Za razliku od njegovih vršnjaka, koji su uglavnom pisali gotički horor sa čudovištima poput vampira i vukodlaka, Lovecraft je pisao o stvorenjima i konceptima koje su skoro nemogući za pojmiti. Ideja kozmičkog horora, i kozmicizma kao svjetonazora, je ta da je čovječanstvo potpuno nevažno gledajući enormnost svemira. Da citiramo samog autora: „Živimo na mirnom otoku neznanja usred crnih mora beskonačnosti, i nije nam bilo suđeno da putujemo daleko od obale“. Sva zbivanja i stvorenja u Lovecraftovim djelima su unikatna po tome što ne dijele nikakvu sličnost sa stvarnim bićima ili folklorima. Glavni likovi njegovih priča ne uspijevaju u postizanju svojih ciljeva, te sam čin spoznaje negativno utječe na njih. Upravo zbog ovih preklapajućih tematika i motiva su elementi Lovecraftovog horora često korišteni u kasnijim science fiction pričama (Soloski, 2020). Ovaj horor je bio inspiracija za atmosferu igre, te za izgled i ponašanje neprijatelja u njoj.

4. Godot

U svijetu razvoja videoigara, individualne osobe imaju veliki broj opcija kod odabira alata koje će koristiti za razvoj svojih ideja. Dok su u prošlosti svi napredni pokretači igara koristili plaćene licence, danas su svi popularni pokretači besplatni za korištenje, sa mogućim naplatama tek nakon što se ostvari značajan profit na nekom projektu. Ali tu se također nalazi niz pokretača koji su objavljeni kao projekti otvorenog koda, od kojeg je veliku popularnost stekao Godot. Sa svojom MIT licencom, i stotinama doprinositelja izvornom kodu, Godot je postigao veliki uspjeh u narednih godinu dana, u kojima je po brojevima korisnika postao konkurencija mnogo starijima i popularnijim alternativama Unity i Unreal Engine (Mike, 2024). U tih godinu dana je postigao novu inačicu koda koja je donijela mnoga poboljšanja i alate, što je zajedno sa njegovom kompaktnošću (cijeli pokretač je jedna izvršna datoteka od 120mb) i praktičnom dizajnu privukla mnoge nezavisne programere i developere.

U sljedećim sekcijama ćemo obrazložiti najvažnije elemente rada u Godotovom editoru kako bismo imali potreban kontekst za razumijevanja razvoja naše projektne igre.

4.1. Metode renderiranja

Godot u trenutnoj iteraciji nudi nekoliko metoda renderiranja scene, koje se ponajviše razlikuju po metodama kalkulacije osvjetljenja scena. Ove tri metode također pružaju različitu razinu kompatibilnosti sa starijim grafičkim procesorskim uređajima. Prve dvije imaju podršku za moderna grafička programska sučelja Vulkan i Direct3D 12, dok zadnja ima samo za OpenGL.

Prva ponuđena metoda je *Forward+* metoda, koja koristi grozdasti pristup osvjetljenju, gdje se za osvjetljenje koristi compute shader kako bi grupirao svjetla u 3D mrežu. O shaderima ćemo više kasnije, ali bitno je samo za znati da su shaderi (ili programi za sjenčanje) programi koji se izvršavaju na jezgrama grafičkog procesora, a ne na centralnom procesoru računala. Zatim, u vrijeme renderiranja, pikseli mogu provjeriti koja svjetla utječu na ćeliju mreže u kojoj se nalaze i izvršiti izračune samo za ona svjetla koja mogu utjecati na taj piksel. Ovaj pristup može značajno ubrzati performanse renderiranja na stolnim računalima, ali je znatno manje učinkovit na mobilnim uređajima (*Internal Rendering Architecture*, 2024).

Upravo zato je sljedeća metoda *Forward Mobile* prilagođena specifično za ciljanje mobilnih uređaja. Ona koristi tradicionalniji pristup kalkulaciji svjetla, uz jedinu iznimku da uzima u obzir unikatni način rada mobilnih grafičkih procesora, koji umjesto da kalkulira sjenčanje cijele slike ekrana, ekran se podijeli na manje segmente koji se odvojeno kalkuliraju. Ovaj pristup bolje paše brzoj ali ograničenoj internoj memoriji mobilnih grafičkih procesora (*Internal Rendering Architecture*, 2024).

Zadnja metoda je *Compatibility* koja koristi tradicionalno OpenGL grafičko programsko sučelje, koje kompatibilno sa većinom današnjih grafičkih procesorskih jedinica. Ova metoda je idealna za slučajeve gdje želimo da starija računala mogu pokretati našu igru, uz nedostatak nekoliko ključnih modernih funkcionalnosti koje nisu implementirane u OpenGL-u.

4.2. Skripte

Za definiranje ponašanja objekata u Godotu koristimo njihov interni programski jezik zvan GDScript. On je slabo tipizirani programski jezik visoke razine apstrakcije, sa sintaksom koja jako sliči popularnom Python jeziku. Ali za razliku od Pythona, GDScript nudi poboljšanja u obliku optimizacija brzine tijekom izvršavanja, potpomognuta definiranjem tipova podataka od strane korisnika. Dok se u Pythonu mogu samo dati natuknice za tipove koji pomažu programskom okružju, definiranje tipova varijabli u GDScriptu potpomaže bržem izvršavanju i interpretiranju koda.

Uz sam GDScript jezik, Godot službeno podržava programiranje u C# i C++ jezicima, te daje aplikacijsko sučelje zvano GDExtension koje omogućava interoperabilnost Godota sa bibliotekama napisanim u programskim jezicima poput Rust-a, Go-a i Swift-a (*What Is GDExtension?*, 2024). U našem projektu ćemo koristiti GDScript kako bi minimizirali broj programskih okružja koja trebamo imati postavljena.

4.3. Čvorovi (Nodes)

Čvorovi su najosnovnija jedinica koja izgrađuje bilo koji interaktivni element u Godot igrama. I svi čvorovi su derivirani po principima nasljeđivanja od najosnovnije Node klase, i dijele se na tri generalne kategorije. 2D čvorovi za sve funkcionalnosti potrebne za funkcioniranje 2D igri, 3D čvorovi koji su najčešće ekvivalente svojim 2D rođacima, samo sa proširenim funkcionalnostima i atributima kako bi funkcionirali u 3D prostoru. I zadnji su kontrolni čvorovi koji su namijenjeni za kreiranje grafičkih sučelja, pa ne trebaju imati kompleksne funkcije i parametre za izračunavanje fizičkih interakcija.

Svakom čvoru može biti pridijeljena skripta u odabranom programskom jeziku koja proširuje njegove funkcionalnosti sa funkcijama i atributima koje definira korisnik. Neke od funkcija koje svi čvorovi nasljeđuju su: funckija _ready koja se izvršava svaki put kada je čvor instanciran, funckija _process koja se izvodi u svakom ciklusu izvedbe programa, funckija _physics_process koja je ekvivalenta _process funkciji uz dodatak delta parametra koji označava vrijeme koje je prošlo od zadnjeg ciklusa i koji služi za razdvajanje logike te funkcije od brzine izvođenja ciklusa koji ovise o snazi procesora u računalu (*Nodes and Scenes*, 2024). To je vrlo važno za ne bi došli do situacija gdje igračeva brzina varira ovisno o tome koliko je procesor pod stresom u određenom trenutku.

Osim tih funkcija također imam obitelj _input funkcija koje reagiraju na korisničke unose i imaju hijerarhiju važnosti. Osim ovih funkcija, čvorovi se također mogu dodati u imenovane grupe preko čijih imena se u kodu može od jednom referencirati veliki broj aktivnih čvorova. Svaka skripta je u biti nova klasa koja nasljeđuje metode i atribute od osnovne klase koja tvori dodijeljeni čvor i, ako joj se doda ime, kroz kod se mogu te klase dalje nasljeđivati po tradicionalnim principima polimorfizma. Svaki čvor ima queue_free funkciju koju koristimo kako bismo ga oslobodili iz memorije.

Svaki čvor može imati djecu koja mu se dodjeljuju kroz kod ili kroz Godotov editor, i ta djeca nasljeđuju od njega mnoge atribute. Time se mogu koristiti kombinacije čvorova da bi se ostvarile kompleksne akcije i ponašanja.

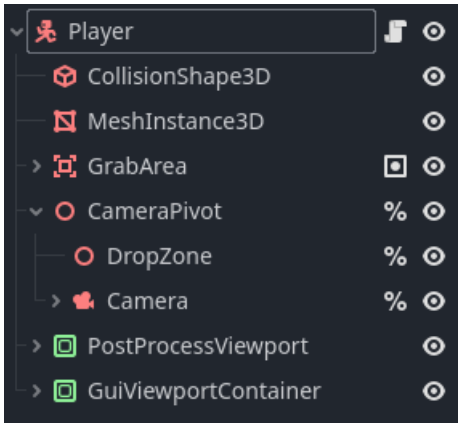
4.3.1. Signali

Signali su način na koji čvorovi djeca mogu komunicirati sa svojim nadređenim roditeljskim čvorovima. Oni su događaji koji se definiraju u čvoru djetetu, te se roditelji čvorovi tijekom svoje inicijalizacije povežu sa deklariranim funkcijama koje će primiti signal kada ga dijete emitira i reagirati na njega (*Using Signals*, 2024).

Svaki čvor nasljeđuje od svih svojih nadređenih klasa unaprijed definirane signale, ali i sam korisnik može definirati svoje vlastite signale kroz skripte. Tako možemo imati čvor koji nasljeđuje klasu Timer koja ima definirati signal *timeout* koji se emitira svaki put kada istekne definirani vremenski interval. Svaki nadređeni čvor se može povezati na taj signal i imati svoju reakciju na njega. Ali mi možemo proširiti ovu klasu sa svojom skriptom i definirati signal *even_timeout* koji će se emitirati na svaki parni po redu timeout događaj. U slučaju unaprijed definiranih signala, na njih se čvorovi mogu povezati kroz sam Godot editor, dok je za vlastite signale pametnije da se povezuju kroz kod.

4.4. Scene

Scene su stablaste strukture sačinjene od čvorova koje čine jednu logičku jedinicu. One nam omogućuju da segmentiramo elemente igre na logične dijelove koji će moći koristiti kao nezavisni moduli. Svaka scena se može sačinjavati od kombinacije čvorova i manjih scena. Svaka scena u Godotu se sprema u .tscn datoteku i takva se scena smatra „pakiranom“ (packed scene) i kao takva se može pozivati u kodu i programski instancirati u bilo kojoj drugoj sceni.



Slika 8: Čvorovi koji čine finalnu scenu Player (autorski rad)

4.5. Resursi (Resources)

Resursi su druge najvažnije programske strukture poslije čvorova u Godotu. Za razliku od čvorova koji predstavljaju najosnovnije funkcionalne jedinice u Godotu, resursi predstavljaju kompleksne i jednostavne strukture podataka koje čvorovi koriste kao atribute, osim jednostavnih varijabli. To mogu biti strukture koje predstavljaju teksture, 3D objekte, pripremljene animacije, fontove itd. (*Resources*, 2024)

Poput čvorova, moguće je također definirati vlastite resurse, koji će predstavljati specifičnu vrstu podataka koju zatim možemo instancirati kao .tres datoteke. Usprkos svojem nazivu, moguće je nadodati i funkcionalnosti na resurse, ali oni neće moći aktivno promatrati scene u kojima se instanciraju, već samo slati signale u jednom smjeru, nikada ih primati.

4.6. Viewport

Viewport je specijalna vrsta čvora koja predstavlja virtualni ekran u svijetu igre i može se koristiti u raznovrsnim slučajevima. Viewport nam u biti daje virtualnu teksturu na koju se projicira pogled sa jedne od virtualnih kamera definiranih u sceni igre. Mi to možemo iskoristiti onda da među ostalim stvarima: projiciramo 3D objekt u 2D igru, projiciramo 2D objekt u 3D igru, kreiramo virtualni ekran s kojim igrač može vršiti interakciju, generirati proceduralne teksture i mnogo više (*Using Viewports*, 2024). Bitno ih je razlikovati od blokova za kontrolne čvorove koji čine grafička sučelja, pošto svaki viewport čvor može imati definirano vlastito nezavisno grafičko sučelje.

Još jedna od koristi ovog čvora je primjena efekata i programa za sjenčanje na već rasteriziranu glavnu sliku igre koja se projicira na korisnikov ekran, i mi ćemo ih u ovom projektu koristiti primarno u ovu svrhu.

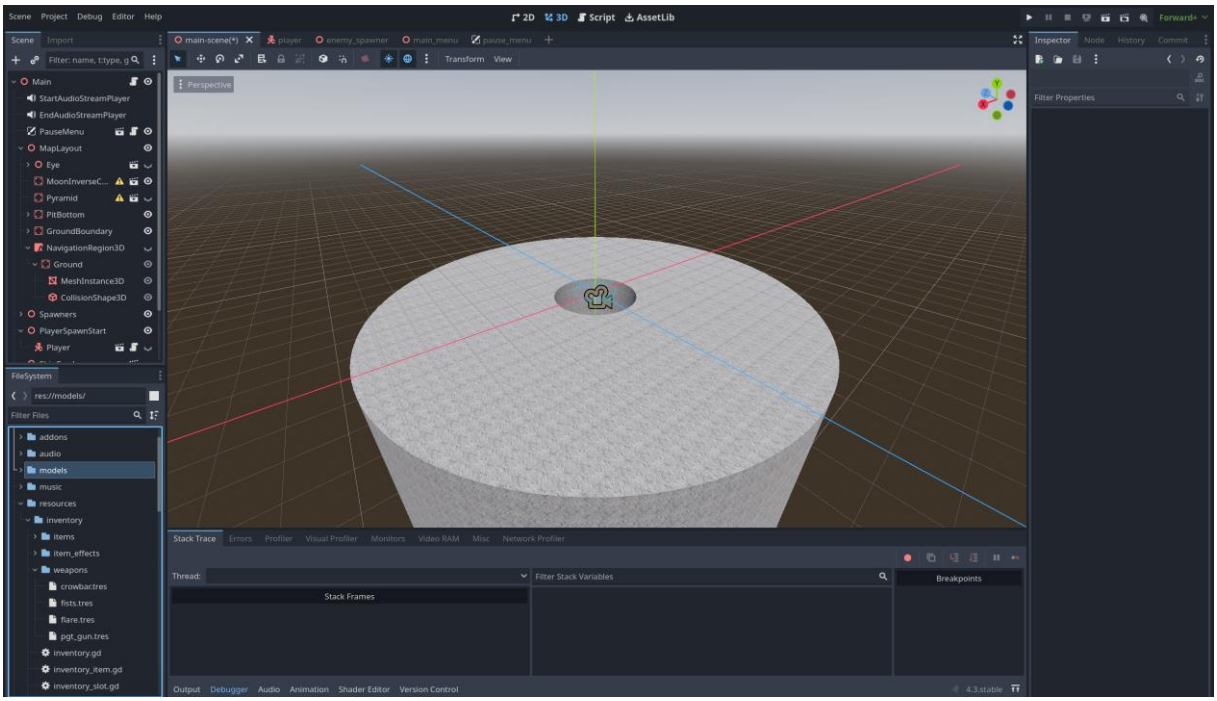
5. Izrada igre

U ovom poglavlju rada ćemo se napokon dotaći primjene alata, inspiracija i koncepata iz prethodnih poglavlja kako bi izradili prototip survival horror igre iz prvog lica. Primarno ćemo opisivati proces izrade projekta u Godot pokretaču, te se samo marginalno dotaći drugih već navedenih alata u odabranim slučajevima gdje se je njihovo korištenje više isticalo.

5.1. Postavljanje početne scene

U samom početku razvoja igre, potrebno je postaviti početnu scenu koja će sadržavati sve postavke potrebne za testiranje elemenata koje ćemo postepeno dodavati i stvarati. To uključuje tlo na kojem će igrač i neprijatelji hodati, te čvor za okoliš (environment) koji definira izgled globalnog osvjetljenja scene.

S obzirom da u ovoj igri igrač neće trebati skakati, nema potrebe stvarati ikakvu vertikalnost u samom terenu razine koristeći stepenice ili uzvisine, a olakšat će nam posao kasnije kada ćemo definirati neprijateljsko kretanje. Kao korijenski čvor koristimo običan Node3D čvor, koji je najosnovniji čvor za kreiranje 3D scena u Godotu.



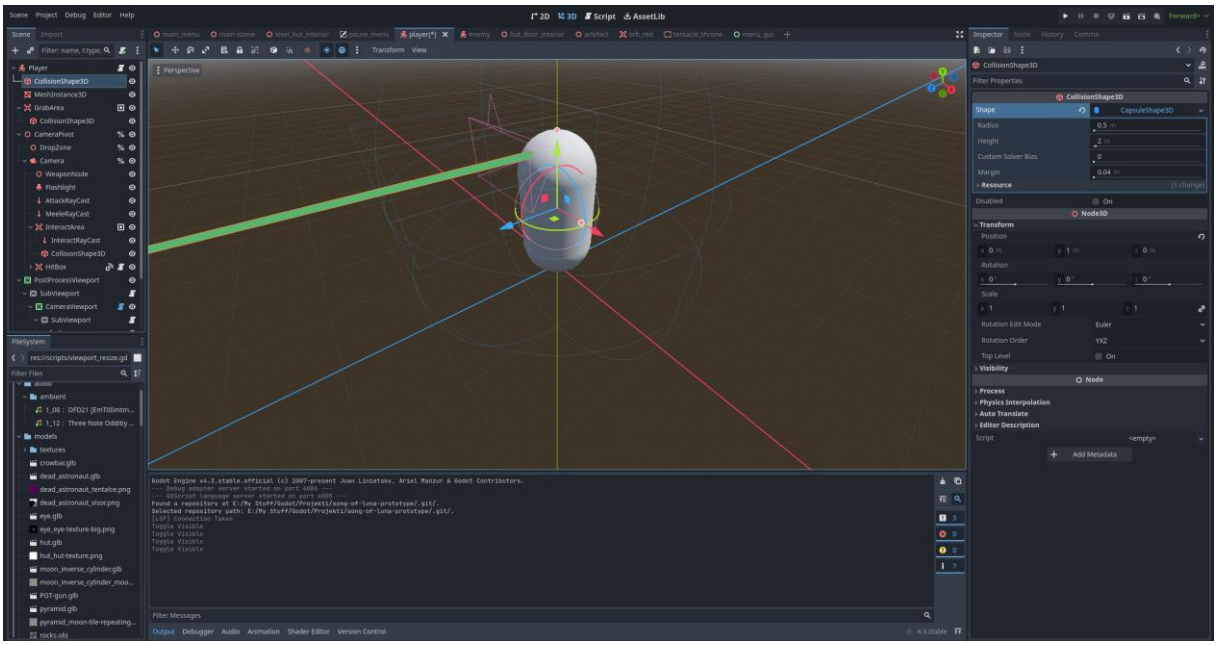
Slika 9: Postavljanje početne scene (autorski rad)

Već u ovom dijelu rada definiramo okvirni izgled početne razine naše igre, gdje će se igrač naći napušten na cilindričnom okolišu na površini mjeseca. Tlo koristi teksturu koja je izrađena procesom pikselizacije službene slike mjesečeve površine preuzeto sa NTRS arhive.

Environment je čvor u Godotu kojim opisujemo kako želimo da naša scena izgleda. Tu se ponajviše referiramo na načine kako je okoliš razine osvijetljen, ali uzima u obzir i kompleksnije efekte poput magle, vizualne efekte zamućivanja i prilagođavanja tonova boje slike (*Environment*, 2024). Nas specifično zanima postavka neba, jer želimo da nebo izgleda poput svemira i zato mora emitirati samo minimalnu količinu svjetla, dovoljnu da igrač može vidjeti razinu izvan dosega njegove svjetiljke. Sam izgled noćnog neba i zvijezda ćemo definirati u kasnijem poglavlju, za sada ćemo staviti samo vrlo tamnu sivu boju neba kao predložak.

5.2. Kreiranje kontrolera za igrača

Kreiranje dobrog čvora za kontrolu igrača će biti jedan od najbitnijih elemenata igre, s obzirom da je to čvor koji će vršiti interakciju sa svim ostalim čvorovima i resursima unutar igre. Radi postizanja što veće modularnosti, ovaj skup čvorova koji će tvoriti našeg igraća ćemo spremiti kao odvojenu scenu koja će se onda kasnije moći instancirati u bilo kojoj drugoj sceni kojoj želimo. Za postavljanje ovog igračevog čvora koristimo znanje iz Godotove baze znanja i video tutorijala od (GDQuest, 2021).



Slika 10: Postavljanje scene igrača (autorski rad)

Korijenski čvor ove scene će biti CharacterBody3D, koji je specijaliziran za kreiranje objekata koji se pomiču kroz korištenje skripti. Ovaj korijen nam daje vrlo jednostavne API pozive za izvršavanje kalkulacija kretnji koje ovise o igračevim unosima (*CharacterBody3D*, 2024). Kalkulacije tih kretnji implementiramo u sljedećem kodu:

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


    var input_dir: Vector2 = Input.get_vector("move_left", "move_right", "move_forward", "move_back")
    var direction: Vector3 = (camera.get_camera_transform().basis * Vector3(input_dir.x, 0, input_dir.y)).normalized()
    velocity.x = direction.x * current_speed
    velocity.z = direction.z * current_speed
    move_and_slide()
```

Ovaj blok koda se izvršava unutar `_physics_process` funkcije, kako bi brzina kretanja igrača bila neovisna o brzini izvršavanja igre. U protivnom bi imali situacije gdje bi se osoba sa mnogo moćnijim procesorom kretala mnogo brže nego osoba sa slabijim računalom. Prvo imamo jednostavnu provjeru da li igrač dotiče tlo, te postavljanje vertikalnog ubrzanja u slučaju ako nije na tlu. Varijabla za gravitaciju povlači vrijednost ubrzanja iz postavki Godot projekata, koje se mogu mijenjati. Ovo je uglavnom za krajnje slučajeve, pošto planirane razine nemaju vertikalno uzdizanje igrača.

Zatim se koristi `Input.get_vector` statična funkcija kako bi se iz korisničkih unosa izračunao normalizirani vektor smjera. Argumenti ove funkcije su nazivi korisničkih akcija, koje smo prethodno definirali u postavka projekta. Npr. „move left“ akcija korespondira na pritisak D tipke na tipkovnici, a „move_forward“ na pritisak tipke W itd. Takvim definiranjem akcija olakšava se posao kreiranja opcijskih menija gdje bi kasnije sami igrači mogli izmijeniti korespondirajuće tipke sa akcijama u igri. Zatim se dobiveni 2D vektor koristi u kalkulaciji normaliziranog 3D vektora koji nam predstavlja konačni smjer pomaka igrača u trenutnoj renderiranoj slici u ciklusu igre. Zatim X i Z komponente dobivenog vektora prosljeđujemo korespondirajućim atributima ovog čvora, pomnoženim sa varijablom za brzinu igrača.

```
# Calculate speed
if Input.is_action_pressed("sprint") and is_on_floor() and not
is_recovering_stamina:
    current_stamina = clamp(current_stamina - STAMINA_DECAY, 0.0,
    MAX_STAMINA)
    current_speed = SPRINT_SPEED
else:
    current_stamina = clamp(current_stamina + STAMINA_RECOVERY, 0.0,
    MAX_STAMINA)
    current_speed = MOVE_SPEED

SignalBus.player_speed_updated.emit(current_speed, current_stamina,
MAX_STAMINA)
```

Kasnije ćemo ovaj dio koda nadograditi sa funkcijama koje izračunavaju igračevu brzinu u sprintu, koja se aktivira na igračev pritisak gumba i ovisi o tome koliko je izdržljivosti ostalo igraču. Izdržljivost mu se obnavlja dokle god ne pokušava sprintati, a ako se izdržljivost skroz potroši, igraču je onemogućeno sprintanje sve dok se njegova izdržljivost nije potpuno

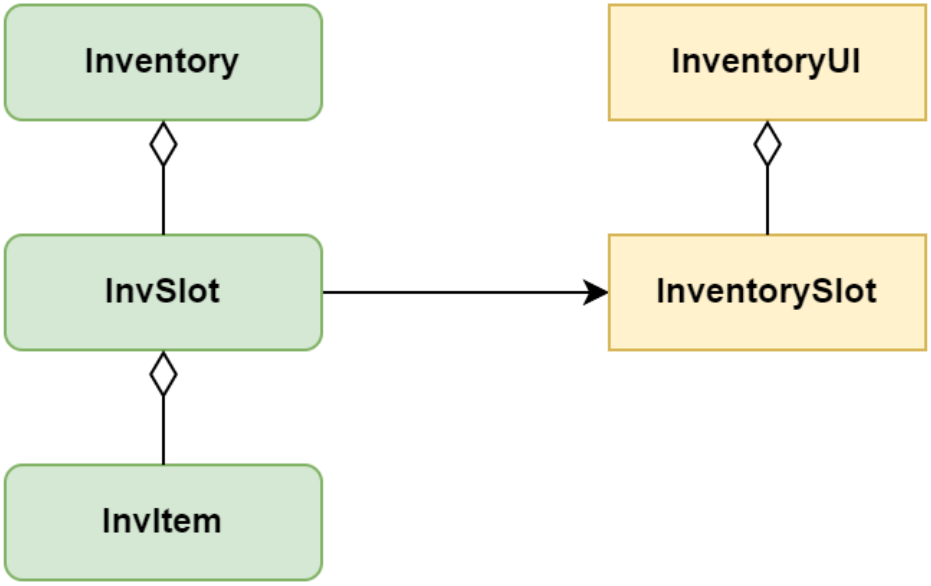
oporavila. S obzirom da će bez sprintanja igrač biti malo sporiji nego neprijatelji, igrač će morati pažljivo procjenjivati rizik vezan uz trošenje svoje izdržljivosti.

```
func _unhandled_input(event: InputEvent):
    if is_dead:
        return

    # Rotate camera
    if event is InputEventMouseMotion:
        pivot.rotate_y(-event.relative.x * LOOK_SENSITIVITY)
        camera.rotate_x(-event.relative.y * LOOK_SENSITIVITY)
        camera.rotation.x = clamp(camera.rotation.x, deg_to_rad(-
X_ROTATION_LIMIT), deg_to_rad(X_ROTATION_LIMIT))
```

Koristimo funkciju _unhandled_input u skripti od kontrolera kako bi detektirali pokrete miša. Ova funkcija se poziva svaki put kada korisnik napravi neki unos u igri koji ni jedna druga funkcija u igri nije konzumirala. U funckiji prvo potvrđujemo da je igrač živ i da je vrsta događaja na koji se reagira pomak miša. Zatim se kalkulira rotiranje kamere i vrata igrača. Razlog zašto su to odvojeni čvorovi je taj što horizontalno rotiranje kamere stvoriti neprirodno pomicanje ekrana koje nije ekvivalentno pomicanju vidnog polja kod čovjeka.

5.3. Sustav inventara stavki (inventory)

Kako bismo stvorili modularan i prilagodljiv sustav inventara (eng. inventory) kojem će igrač imati pristup, moramo ga podijeliti na dva dijela. Nadograđujući predložak od (DevWorm, 2023), definiramo inventar koji se sastoji od prilagođenih resursa koji pohranjuju u sebi sve informacije koje želimo da stavke u inventaru imaju, a zatim definiramo grafičko sučelje inventara koje se sastoji od 2D čvorova i koje ima sličnu strukturu kao i inventar kao resurs.



Slika 11: Dijagram klasa za sustav inventar (autorski rad)

Prvo definiramo Inventory klasu kao roditeljsku klasu koja kao atribut sadrži listu utora (InvSlot) u koje se mogu ubaciti i ukloniti stavke (InvItem). Broj utora je ograničen na 6, kako bi potaknuo igrača da bude štedljiv sa svojim resursima. Broj istovrsnih stavki koje igrač posjeduje definiran je u InvSlot klasi s integer atributom amount. InvItem klasa sadrži naziv stavke, njezin opis, put do njene ikone koja će ju predstavljati u grafičkom sučelju, te put do scene koja će se instancirati kada igrač izbaci stavku iz svog inventara. Također je tu definirana enumeracija ItemType koja definira vrste stavki koje igrač može pokupiti.

```
func insert(item: InvItem, amount: int):
    # Array of slots that contain the item
    var item_slots: Array[InvSlot] = slots.filter(func(slot): return
slot.item == item)
    if !item_slots.is_empty():
        item_slots[0].amount += amount
    else:
        # Get first empty slot
        var empty_slots: Array[InvSlot] = slots.filter(func(slot):
return slot.item == null)
```

```
            if !empty_slots.is_empty():
                empty_slots[0].item = item
                empty_slots[0].amount = amount
        SignalBus.slots_updated.emit()


func remove(item: InvItem, amount: int):
    # Check if item equiped
    if equiped_item.item == item:
        unequip(item)
    # Array of slots that contain the item
    var item_slots: Array[InvSlot] = slots.filter(func(slot): return
slot.item == item)
    if !item_slots.is_empty():
        item_slots[0].amount -= amount
        if item_slots[0].amount <= 0:
            item_slots[0].item = null
            item_slots[0].amount = 0
    SignalBus.slots_updated.emit()
```

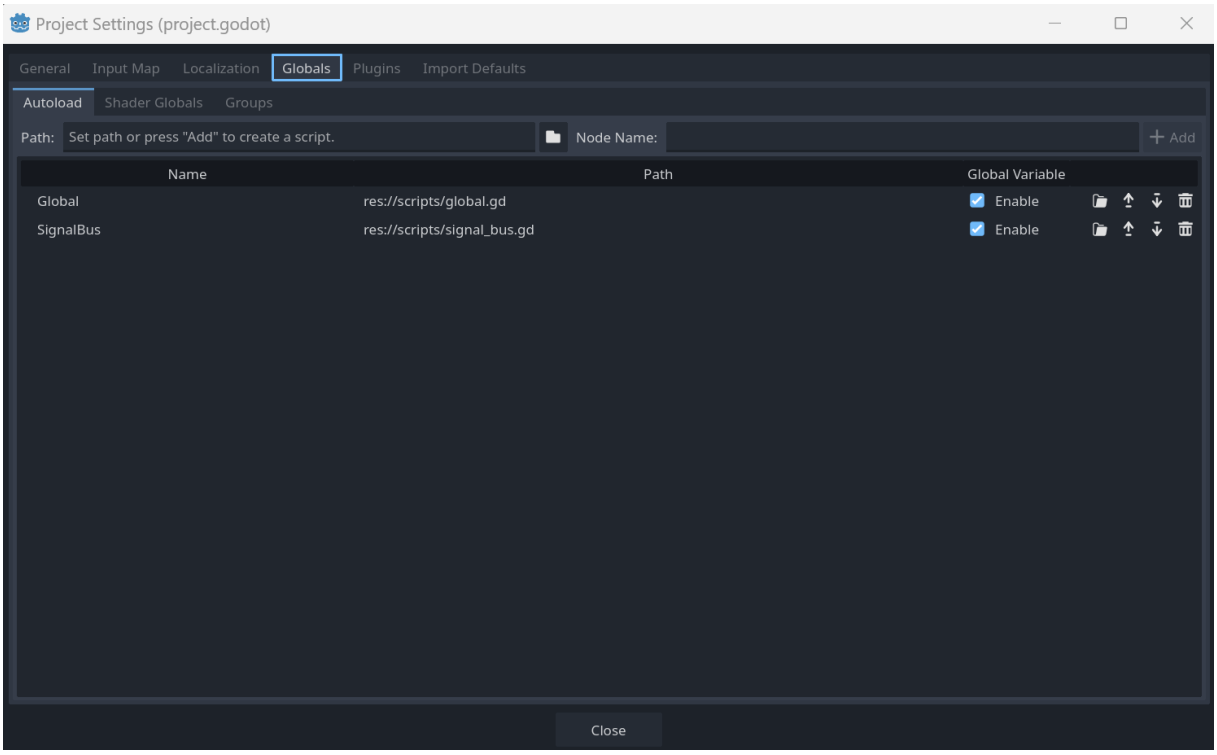
U početku u klasi Inventory definiramo osnovne insert i remove funkcije koje ćemo koristiti za manipuliranjem sadržaja inventara. Obije funkcije kao argument primaju referencu na objekt koji se želi ukloniti ili dodati, te količinu istog. Svaka funkcija zatim obavi provjeru količina slobodnog mjesta i postojećih stavki, te izvršava svoju akciju ako su uvjeti zadovoljeni. Na kraju svake funkcije se obavlja poziv na SignalBus globalnu klasu, koju ćemo objasniti u sljedećim sekcijama.

InventoryUI klasa je definirana u skripti koja je postavljena na korijenskom Control čvoru u kojem je definiran izgled i raspored elemenata u grafičkom sučelju za inventar. Ova klasa primarno sprema referencu na resurs u kojem je pohranjen sadržaj igračevog inventara, referenca na utore koji će biti čvorovi djeca od ove skritpe, te definira u sebi update_slots funkciju na koju se radi poziv svaki put kada se izvrši bilo kakva operacija ili promjena nad stavkama koje su u igračevom posjedu. Kako bi se postigla uspješna komunikacija između

ovih grafičkih elemenata i samih resursa inventara, potrebno je definirati nekoliko globalnih funkcija u ovom projektu.

5.3.1. Globalna skripta

Prvo ćemo definirati globalnu klasu Global, koja će u sebi pohranjivati varijable i statičke funkcije kojima ćemo imati pristup u svakom dijelu koda igre pozivanjem naziva klase Global. Za početak će nam ova skripta služiti za pohranjivanje reference na Player objekt, koja će se postavljati svaki put kada se igračev čvor instancira u nekoj sceni. Time će svi drugi čvorovi i resursi moći pristupiti objektu i atributima od Player objekta, a u ovom slučaju to uključuje atribut u kojem je pohranjen igračev inventar. Kasnije ćemo u ovoj klasi pohranjivati reference na sve razine koje ćemo moći učsitati u igri, te implementirati praćenje instanciranih stavki i neprijatelja u tijeku igre.

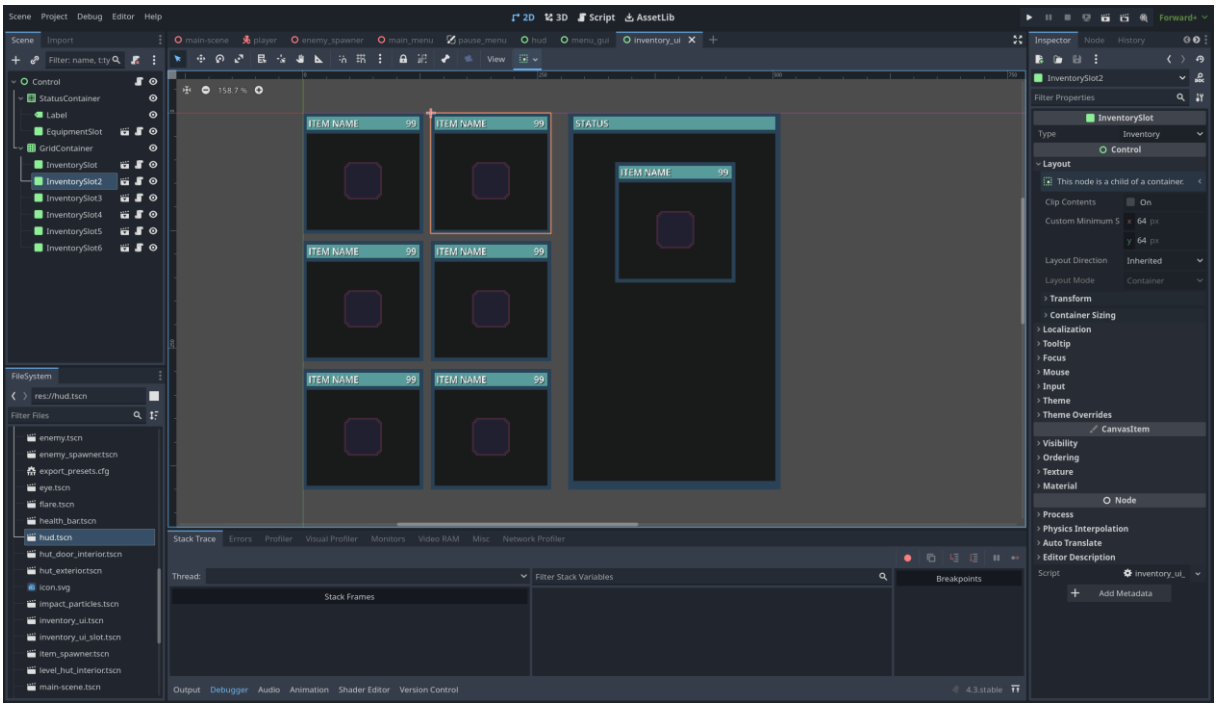


Slika 12: Postavljanje globalnih skripti (autorski rad)

Godot nam pruža jednostavno sučelje u kojem se mogu definirati autoload funkcije koje će biti inicijalizirane tijekom pokretanja igre i time odmah biti dostupne svim drugim čvorovima i skriptama koje će im trebati pristupati iz svojih _ready funkcija.

5.3.2.SignalBus

Sljedeća i zadnja skripta koju ćemo definirati kao globalni autoload je SignalBus skripta koja će na jednostavan način implementirati Event Bus uzorak dizajna. U toj skripti ćemo definirati veliki dio signala koje će čvorovi u našoj igri koristiti. Tako će se čvorovi moći tijekom svoje inicijalizacije povezati na signale definirane u SignalBus skripti, kao što bi radili sa svojim podređenim čvorovima, bez brige da će se čvor inicijalizirati prije davatelja signala.



Slika 13: Scena grafičkog sučelja inventara (autorski rad)

Nakon što smo definirali i postavili SignalBus kao globalnu skriptu, možemo u Godotovom editoru pripremiti novu scenu koja će u sebi sadržavati sve čvorove koji čine grafičko sučelje inventara, te u skripti vezanoj za korijenski čvor povezati se na signale koje smo definirali u SignalBus skripti. Tako ćemo na svaku stavku koji je igrač uzeo ili odbacio, emitirati u Player klasi signal koji obavještava da se taj događaj dogodio, i u skripti od grafičkog sučelja primiti taj signal i osvježiti grafičko sučelje sa novim podacima iz Inventory resursa. I kasnije će se također bilo koji drugi elementi igre, koji trebaju reagirati na promjene u inventaru igrača, moći spojiti na te signale.

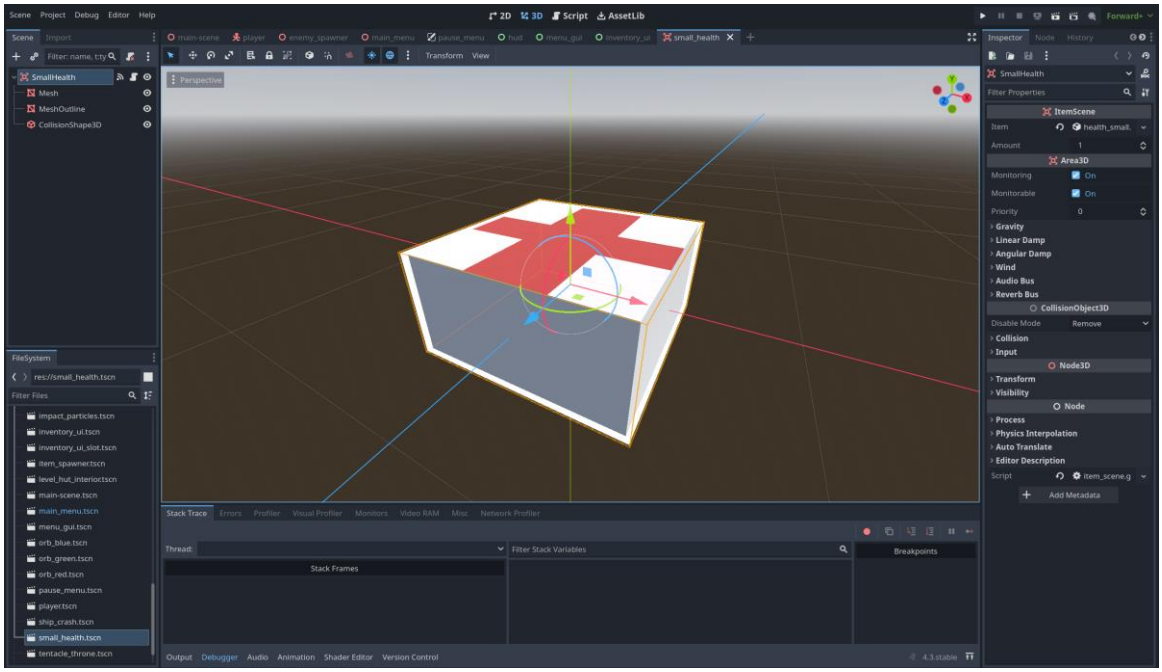
```
# GUI signals
signal gui_hidden()
signal gui_shown()
signal game_paused()
signal game_unpaused()

# Player signals
signal player_speed_updated(current_speed: float, current_stamina:
float, max_stamina: float)
signal enemy_killed(id: String)
signal player_health_updated(value: float, max_value: float)
signal player_death()

# Inventory signals
signal equiped_item(slot: InvSlot)
signal unequiped_item(slot: InvSlot)
signal slots_updated
signal drop_item(slot: InvSlot, amount: int)
signal item_picked(id: String)
signal remove_item(item: InvItem)
```

Kasnije ćemo nadodati u SignalBus skriptu deklaracije za mnogo drugih signala, poput signala koji ciljaju promjene u grafičkom sučelju, npr. kada korisnik pauzira igru i treba se prikazati pauzni meni. Zatim imamo signale koji obavještavaju grafičko sučelje da je igrač ubrzao ili usporio, signal da je neprijatelj ubijen ili igračevo zdravlje reducirano. Za sam inventar imamo signale za slučajeve kada igrač pokupi stavku s poda, izbaci ga iz svog inventara, postavi ga kao opremu ili makne iz utora za opremu.

5.3.3. Stavke (items)

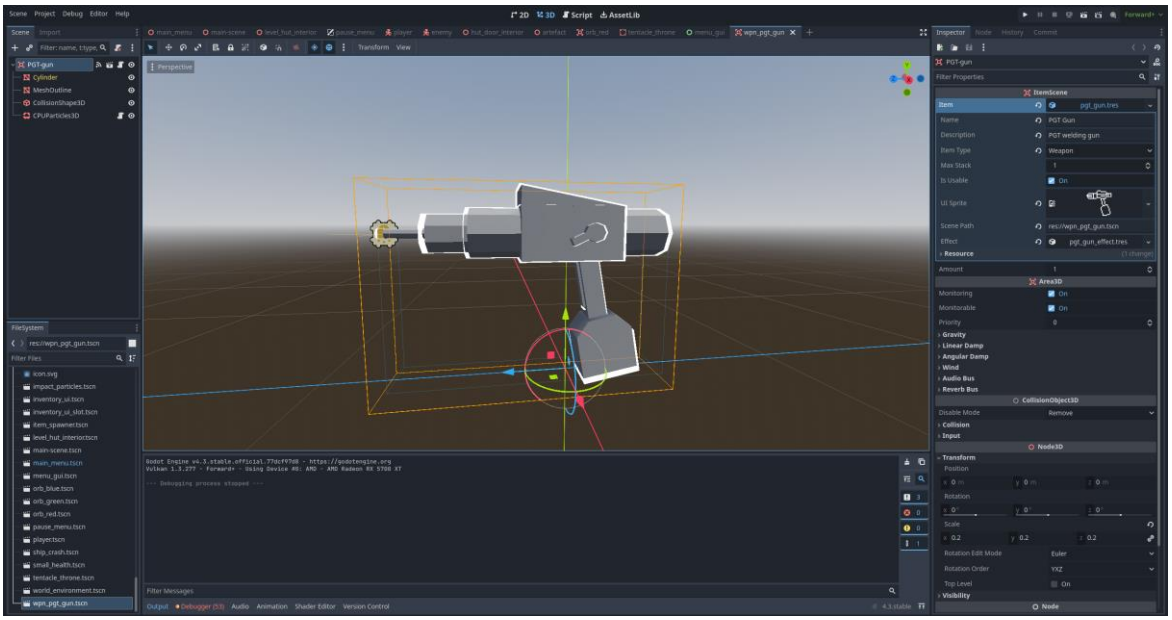
Stavke koje ćemo definirati spadat će u nekoliko vrsta koje smo već definirali u klasama resursa. Prva vrsta je Consumable (potrošna) koja predstavlja stavke koje igrač jednokratno

iskorištava radi nekih beneficija. To će u ovom slučaju biti stavke zdravlja, koje obnavljaju igračevo zdravlje za određenu količinu.



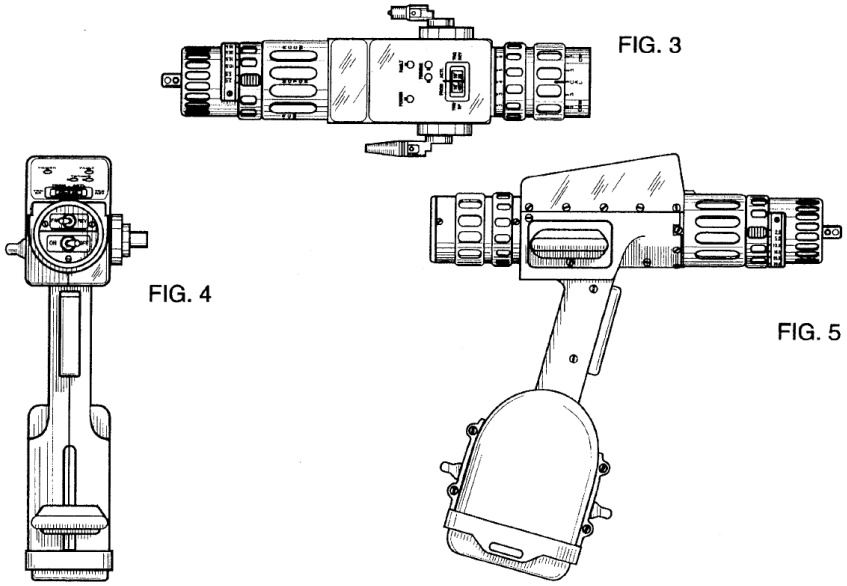
Slika 14: Scena manjeg paketa zdravlja (autorski rad)

U prikazanoj sceni imamo jednostavni set čvorova koji tvore jednostavnu stavku za zdravlje. Jedini bitni atributi izvezeni iz univerzalne skripte za scene stavki su kolicina (amount) stavki koje ova scena sadrži, i referenca na resurs od te same stavke. Osim 3D modela i teksture objekta, ova scena također definira 3D obrub oko objekta koji se programski prikazuje samo kada igrač cilja na taj predmet. Za to koristi svoj čvor za sudare kako bi detektirao kada mu igrač uđe u polje interakcije.



Slika 15: Scena PGT pištolja (autorski rad)

Sljedeća važna vrsta stavke je oružje. U ovom slučaju, na slici je prikazano dalekometno oružje „PGT gun“, koji je inspiriran pravim „Pistol Grip Torque“ alatom koji je razvila NASA za izvršavanje radova u okolišu bez atmosfere i/ili gravitacije. U igri se ovo ponaša kao standardni pištolj, koji u spremniku ima 6 metaka, i koji konzumira resurs koji mora biti u inventaru igrača kako bi se mogao ponovno napuniti. Sva oružja imaju atribut kojim se određuje koliku će štetu nanositi neprijateljima. U sljedećoj slici se može vidjeti tehnički dizajn iz službenog patenta koji sam koristio kao referencu za izradu modela u Blenderu.

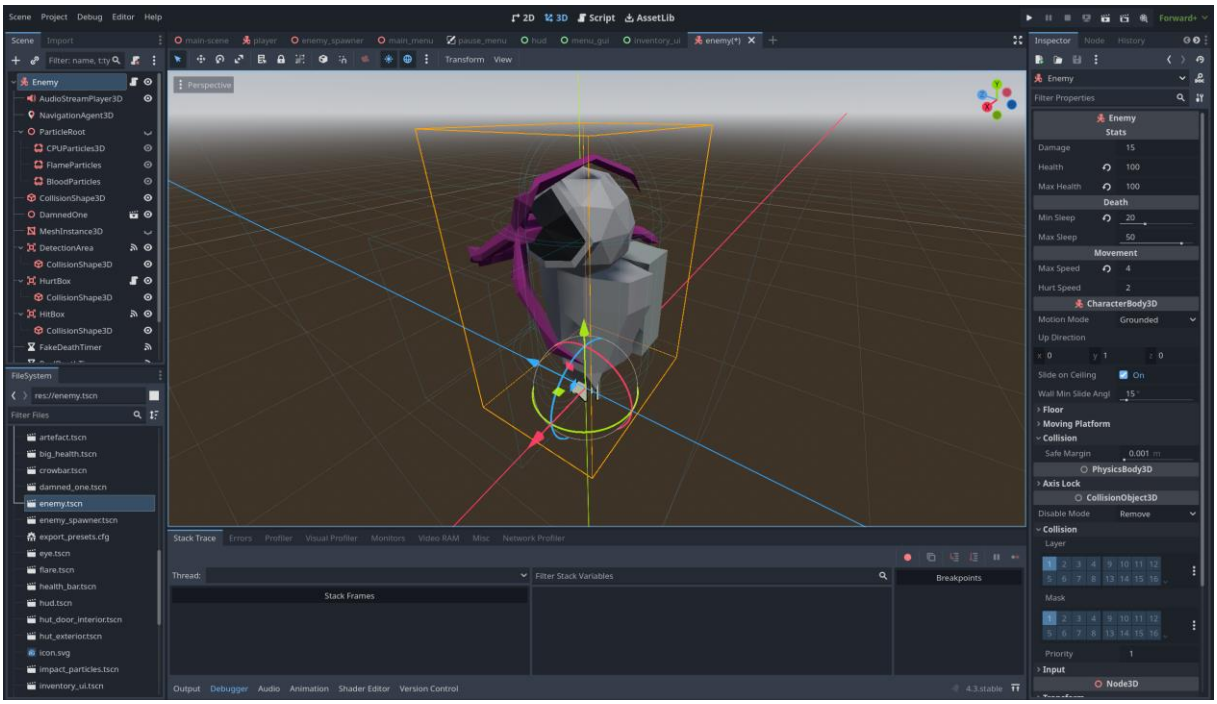


Slika 16: PGT dizajn iz službenog patenta (Richards, 1997)

Osim dalekometnog pištolja, u igri smo također implementirali kratkometno oružje u biti jednostavne željezne poluge, koja radi veću štetu neprijateljima nego prethodni pištolj, ali zahtjeva od igrača da preuzme veći rizik sa približavanjem neprijatelju.

5.4. Neprijatelji

Glavni neprijatelj u ovoj igri će biti u konstantnom stanju mirovanja i nadziranja, sve do trenutka kada igrač ili uđe u njegov vidokrug, definiran sa kružnim područjem detekcije 10 metara polumjera, ili ako sam igrač napadne neprijatelja prije tog trenutka. Neprijatelj će zatim naganjati igrača sve dok mu se zdravlje ne iscrpi.



Slika 17: Neprijateljev čvor (autorski rad)

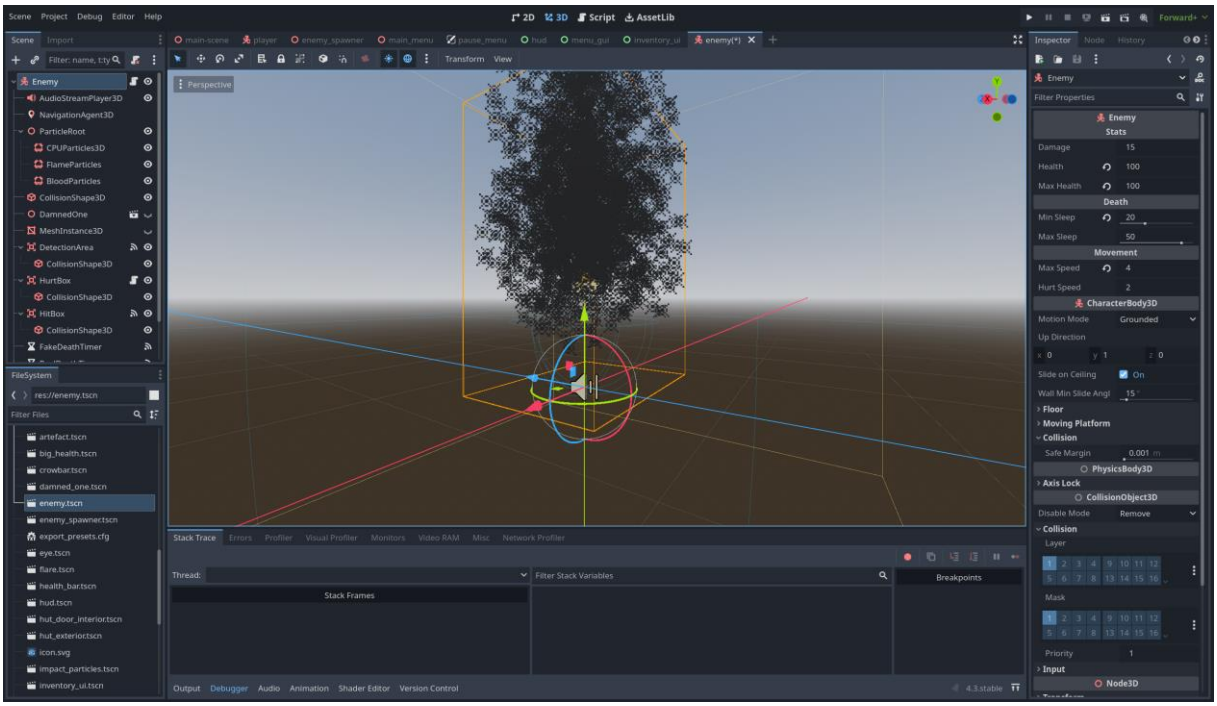
Sam model neprijatelja je modeliran u Blenderu i podijeljen je na dva dijela, glavno tijelo koje izgleda kao astronaut bez udova, i odvojenog modela koji predstavlja ticala koja izlaze iz njegove glave. Dizajn je inspiriran službenom dokumentacijom EVU (Extravehicular mobility unit) odijela kako je opisano u službenom NASA dokumentu (Carson i sur., 1975). Modeli su odvojeni u paketu kako bi im se moglo unutar Godota odvojeno primjenjivati programe za sjenčanje.

Slika 18: Neprijateljevi izvezeni atributi (autorski rad)

Skripta koja kontrolira neprijatelja izvozi mnoge varijable kako bi se unutar Godotovog editora moglo prilagoditi njihovo ponašanje u testnoj sceni. Jedna od posebnih ponašanja ovih neprijatelja je način na koji se moraju poraziti. Nije dovoljno samo svesti neprijateljevo zdravlje na nulu, u tom trenutku samo padaju u stanje sna koje traje u nasumičnom intervalu između određenih minimuma i maksimuma u sekundama. Nakon što taj period istekne, neprijatelj će se probuditi i obnoviti zdravlje na pola. Ovaj proces će se ponavljati sve dok igrač ne zapali neprijatelja korištenjem rijetkog resursa baklje (Flare) kojih uvijek ima u manjoj količini nego neprijatelja, što tjera igrača da bude štedljiv sa njima i izbjegava borbu s neprijateljima kada god je moguće.

5.4.1. Sustav čestica (particle system)

Različiti elementi ove igre koriste sustave čestica kako bi se rekreirali vizualni efekti na dinamičan način umjesto korištenja unaprijed pripremljenih animacija. Jedan od primjera je udarac projektila o tlo kojeg ispucava PGT pištolj. Ili sam pucanj iz istoga. Oba ova primjera koriste pripremljenu scenu koja u sebi sadrži CPUParticles3D čvor koji je pripremljen za lansiranje čestica u smjeru i sa jačinom koja je unaprijed određena. Sam odašiljač tih čestica je za prvi primjer instanciran u trenutku pucnja iz pištolja, koristeći RayCast čvor kako bi izračunao točku sudara između projektila i tla, te iskoristio koordinate istoga da instancira pripremljenu scenu sa česticama.



Slika 19: Sustav čestica oko neprijateljskog čvora (autorski rad)

U gornjoj slici se pak vide čestice koje prekrivaju siluetu neprijatelja, koristeći umjesto jednostavnih pikseliziranih poligona, pripremljene teksture kreirane u Aseprite alatu namijenjene da imitiraju konzistentnost dima u starinskim igrama. Ovaj dim konstantno prati neprijatelja, te je zato optimiziran sa manjim brojem instanciranih čestica po sekundi, sa većim volumenom istih kako bi nadoknadili za sam broj čestica.

5.5. Programi sjenčanja (shaderi)

Shaderi su mali programi koji se izvršavaju na jezgrama grafičkog procesora, koji sa svojih tisućama procesorskih jedinica izvršava te programe u paraleli, najčešće za svaki piksel rasterizirane slike u jednom ciklusu (*Introduction to Shaders*, 2024). U našem projektu koristit ćemo dvije vrste ovih programa za sjenčanje. Koristit ćemo kao primjer ticala koja se nalaze na modelu neprijatelja. Godot nam daje pojednostavljenu verziju vrlo popularnog OpenGL Shading Language (GLSL) jezika, zvanog gdshader. Jezik naliči po sintaksi klasičnom C jeziku.

```
shader_type spatial;

uniform vec4 color : source_color;

uniform float morph_amount : hint_range(0, 1) = 1.0;

uniform float speed_factor : hint_range(0, 1) = 1.0;
```

U ovom primjeru gdshader koda prvo deklariramo vrstu shadera da je spatial (prostorni) što daje do znanja Godotu da se on primjenjuje na 3D model, a ne na 2D teksturu. Nakon toga definiramo uniform varijable, što su jedine varijable koje će shader čitati i primati od procesora računala. Tu definiramo koju boju želimo da ticala imaju i koliko jako i brzo želimo da se migolje.

```
void vertex() {
    float speed = TIME * speed_factor;
    float x = VERTEX.x;
    float y = VERTEX.y;
    float z = VERTEX.z;

    VERTEX.x += cos(y * speed) * sin(z * speed) * morph_amount;
    VERTEX.y += cos(x * speed) * sin(z * speed) * morph_amount;
    VERTEX.z += cos(x * speed) * sin(y * speed) * morph_amount;
}

void fragment() {
     ALBEDO.rgb = color.rgb;
}
```

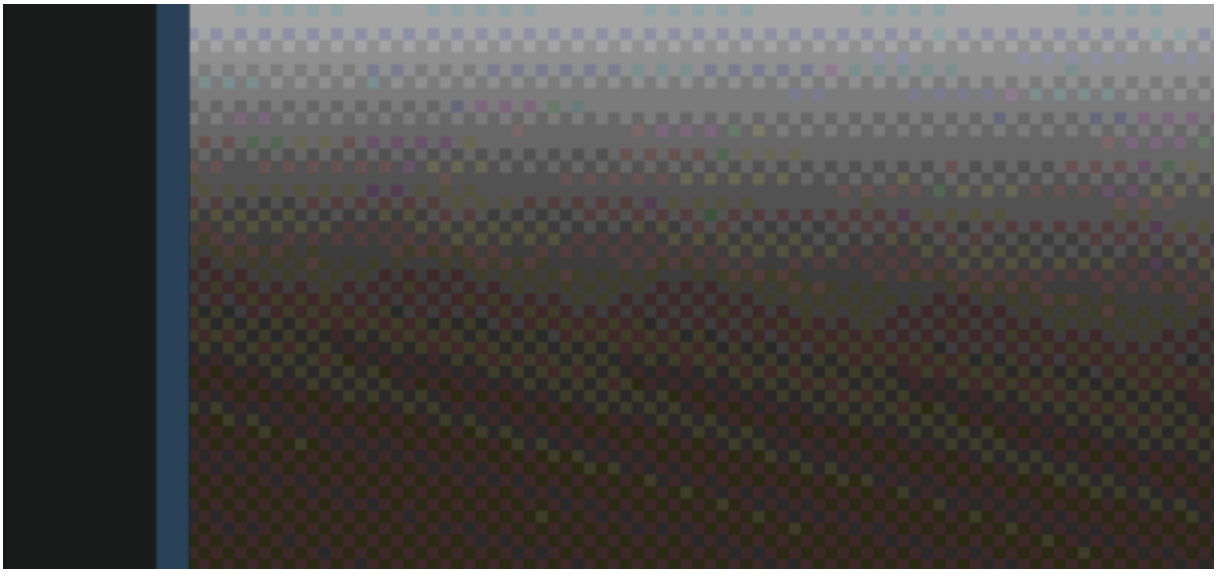
Sljedeće otvaramo dvije najvažnije funkcije koje izvršavaju shaderi. Prva je vertex funkcija koja se izvršava nad svakom točkom koja povezuje plohe u 3D modelu. Sljedeća je fragment funkcija koja se izvršava za svaki rasterizirani piksel na našem modelu, i tu u ovom primjeru prosljeđujemo boju koju smo definirali u uniform varijabli color. U vertex funkciji pak u početku definiramo varijablu za brzinu i raščlanjujemo početne koordinate točke nad kojom će se izvršavati ovaj shader. Na kraju se za svaku os 3D koordinata izvršava kalkulacija koja množi rezultate sinus i kosinus funkcija, te još pomnoženo sa varijablama za brzinu i intenzivnost. Time dobivamo valovitu kretnju svake točke modela ticala koja im daje privid živosti bez da smo ih morali profesionalno modelirati u Blenderu ili nekom drugom 3D alatu.

Osim sjenčanja modela, imamo i shadere koji su primjenjeni na sam viewport igre kako bi se primijenili na već rasteriziranu sliku. S tim shaderima kreiramo iluziju da igramo staru igru iz devedesetih.

```
shader_type canvas_item;
uniform float colors : hint_range(1.0, 32.0);
uniform float dither : hint_range(0.0, 0.5);

void fragment()
{
	vec4 color = texture(TEXTURE, UV);

	float a = floor(mod(UV.x / TEXTURE_PIXEL_SIZE.x, 2.0));
	float b = floor(mod(UV.y / TEXTURE_PIXEL_SIZE.y, 2.0));
	float c = mod(a + b, 2.0);

	COLOR.r = (round(color.r * colors + dither) / colors) * c;
	COLOR.g = (round(color.g * colors + dither) / colors) * c;
	COLOR.b = (round(color.b * colors + dither) / colors) * c;
	c = 1.0 - c;
	COLOR.r += (round(color.r * colors - dither) / colors) * c;
	COLOR.g += (round(color.g * colors - dither) / colors) * c;
	COLOR.b += (round(color.b * colors - dither) / colors) * c;
}
```

Ovaj fragment shader koristimo kako bi reducirali količinu boja koje se mogu pokazati na ekranu i kako bi dodali efekt podrhtavanja (dithering) u bojama u obliku uzorka šahovnice, prilagođavajući kod od (whiteshampoo, 2022). U ovom kodu prvo u color varijablu učitavamo boju teksture koja je već primjenjena na ekran, što će biti zadnja slika ekrana koju je grafička kartica rasterizirala. Zatim koristimo a, b i c varijable za podjelu piksela ekrana na 2 puta 2

rešetku, ovisno da li je piksel paran ili neparan. Zatim kvantiziramo već postavljene boje ekrana ovisno o colors uniform varijabli, te primjenjujemo dither varijablu kako bi dobili efekt šahovnice u dijelovima gdje dolazi do miješanja dviju boja. Povrh ovog shadera za reduciranje boja naslojavamo još jedan kompleksniji shader od (zuwiwano, 2023), prilagođen za noviju verziju Godota, kako bi dobili privid gledanja u stari CRT monitor.



Slika 20: Izgled finalne redukcije boja ekrana (autorski rad)

5.6. Učitavanje drugih razina

Kako bismo mogli učitavati različite razine u našoj igri, moramo implementirati funkcije u Global klasi koje će manipulirati stablom scena tijekom izvršavanja igre. U sam vrh skripte pripremit ćemo varijable koje će tijekom pokretanja aplikacije već pripremiti listu razina koje želimo da su uvijek dostupne za učitavanje kroz kod.

```
# Level scenes
@onready var main_menu_scene: PackedScene = load("res://main_menu.tscn")
@onready var main_scene: PackedScene = load("res://main-scene.tscn")

var current_scene: Node3D = null
func set_current_scene(scene: Node3D):
    current_scene = scene
```

```
func get_current_scene() -> Node3D:
    return current_scene
```

Sljedeće želimo definirati globalnu varijablu current_scene, s svojim setter i getter funkcijama koje će se pozivati u _ready funckijama u svakom čvoru koji je korijen od scena za razine u igri. Tako uvijek možemo znati točno koja je razina trenutno učitana u igri.

```
func load_scene(caller: Node, scene: PackedScene):
    # Load new scene instance
    var scene_instance: Node3D = scene.instantiate()
    scene_instance.request_ready()
    get_tree().root.add_child(scene_instance)
    caller.queue_free()
    # Proper window sizing
    get_tree().root.size_changed.emit()
```

Zatim definiramo load_scene statičnu funkciju koja će primati dva argumenta, prvi je čvor koji predstavlja trenutnu razinu koja je učitana i u kojoj se igrač nalazi, a drugi element predstavlja komprimiranu scenu koju će trebati instancirati i postaviti u stablo scena. Nakon što instanciramo tu scenu, na njoj pozivamo request_ready funckiju kako bi ju obavijestili da će uskoro biti dodana u stablo scena. Nakon što je scena nadodata u stablo, originalna scena koja je obavila poziv se čisti iz memorije. Na kraju emitiramo signal koji obavještava izmjenu prozora aplikacije, kako bi se očuvale dimenzije grafičkog sučelja igre.

5.6.1. Persistentnost predmeta i neprijatelja

Problem na koji nailazimo kada želimo da igrač može nesmetano prelaziti iz jedne u drugu razinu, te se vraćati u prethodne, je taj da tu dolazimo do problema persistencije neprijatelja i stavki na koje igrač nailazi. Ne želimo da igrač ubije sve neprijatelje i pokupi sve stavke u jednoj razini, prebaci se na drugu razinu, te po povratku na originalnu razinu naiđe na iste neprijatelje i stavke koje više ne bi smjeli postojati.

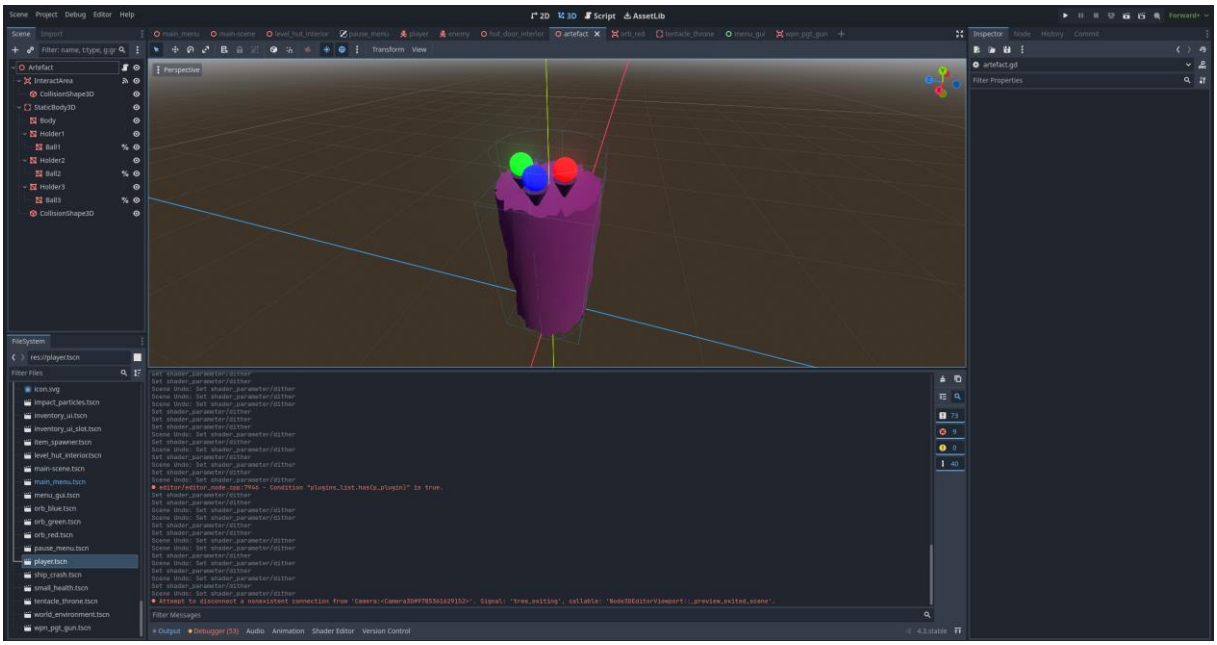
Ovaj problem rješavamo tako da umjesto da unaprijed postavimo sve neprijatelje i stavke u scenu razine, pripremimo posebne čvorove Spawnere koji će dinamički instancirati stavke i neprijatelje ako su uvjeti zadovoljeni.

Spawner čvorovi za neprijatelje i stavke imaju jednostavne skripte, koje emitiraju signal da se zna koji su čvor instancirali, te koji se definira u zajedničkoj „Spawner“ grupi unutar Godot projekta kako bi se tijekom izvršavanja koda moglo dohvatiti sve čvorove koji pripadaju toj grupi i da su trenutno prisutni u stablu scena. Zatim svaka scena razine u svom _ready kodu iterira kroz sve dostupne Spawnere i instancira im predodređenu scenu neprijatelja ili stavke. Ali uz to, također im se pridjeljuje unikatni ID u obliku stringa. Ti ID-ovi su bitni, jer svaki Spawner prije nego instancira neprijatelja ili stavku kao djetinjski čvor, provjerava check_enemy_id i check_item_id statičke funckije od Global klase da li se smije izvršiti to instanciranje.

Kada se neka stavka konzumira, ili se neki neprijatelj dokrajči (znači da nije samo u stanju spavanja), onda se njihov unikatni ID nadoda u listu koja je atribut Global klase, i ta lista se nikada ne briše osim u slučaju kada se igra resetira ili igrač izađe van u glavni meni igre. Tada se instanciranje neprijatelja i stavki prati ispočetka.

5.7. Ciljevi prototipa

Za kraj ovog prototipa moramo definirati neke ciljeve koje će igrač morati ispuniti kako bi završio igru. U ovom slučaju sam se odlučio na jednostavan zadatak sakupljanja ključnih predmeta koji otključavaju finalnu scenu igre. U početku igre igrač se nalazi nasukan na mjesečevoj površini, samo sa metalnom polugom kao oružjem da se brani.



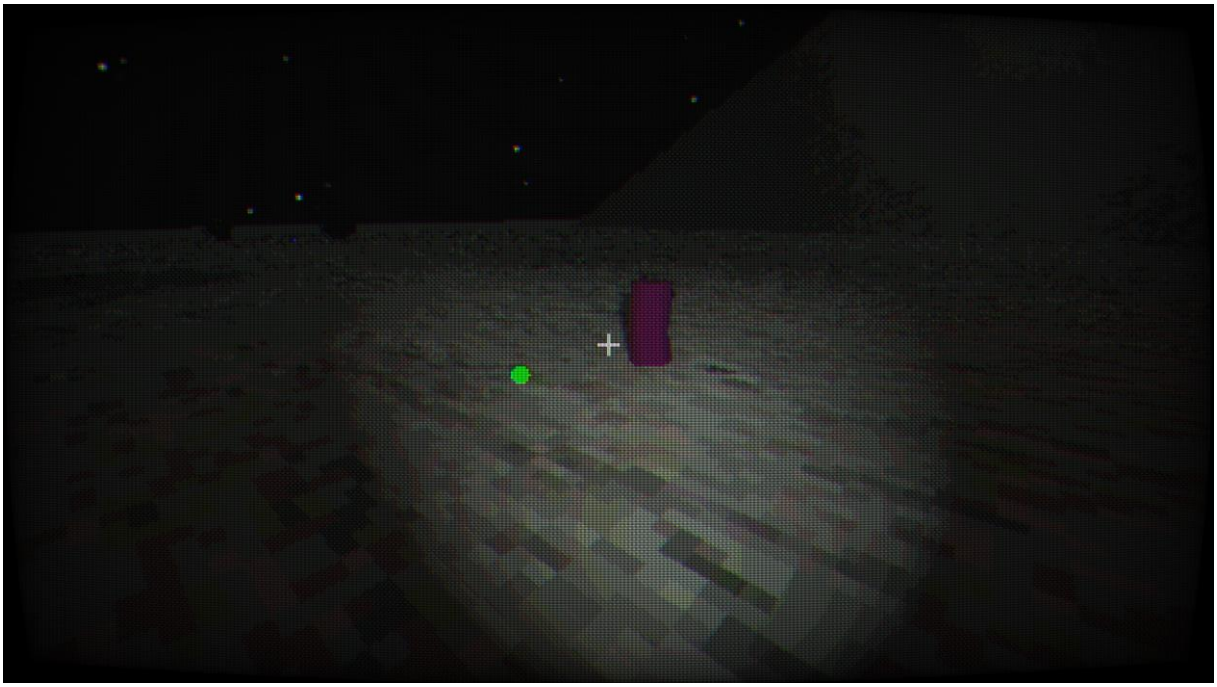
Slika 21: Postolje za ključne predmete igre (autorski rad)

Ključni predmeti u ovom slučaju su tri sfere koje emitiraju različite boje, koje moraju biti sakupljene na pulsirajućem postolju koje je postavljenu na sredini početne razine, koja će služiti kao početna razina na koju se igrač vraća po pronalasku ključeva.

Scena sa postoljem za obojane sfere u svojoj skripti prati stanje igre koje je definirano u Global klasi sa jednostavnim boolean varijablama koje definiraju koje je ciljeve igrač ispunio i koje nije. Kada igrač uspješno sakupi sve obojane sfere i položi ih na postolje, emitira se signal za pokretanje finalne scene u igri. Po završetku te scene, igrač je vraćen na početni meni igre, i sav napredak se kroz Global klasu resetira kako bi igra mogla početi ispočetka.

6. Finalni rezultat

U početku igre igrač se nalazi nasukan na mjesečevoj površini, samo sa metalnom polugom kao oružjem da se brani. U svojoj okolini vidi jedino pustoš mjeseca i mračnu piramidu u udaljenosti. Na sredini kružnog terena nailazi na prvog neprijatelja te na glavni cilj igre.



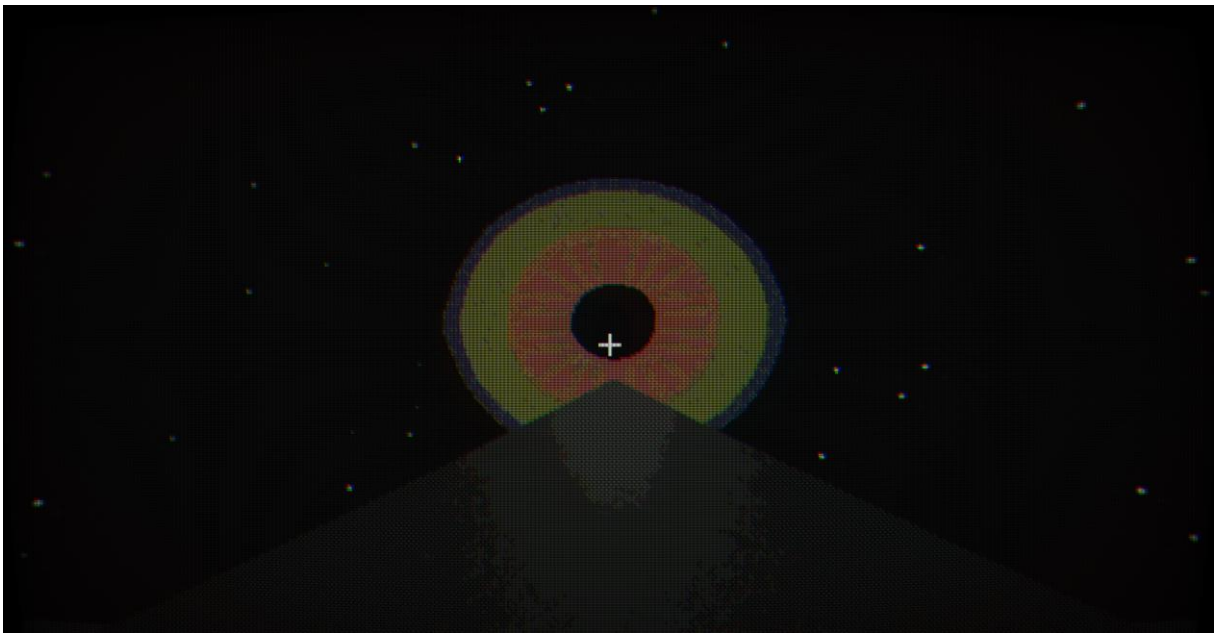
Slika 22: Prva sfera i ključno postolje (autorski rad)

Prva obojana sfera i njihovo postolje nalaze se jedno pored drugoga kako bi igrač odmah mogao dokučiti koja je svrha njegovog zadatka. Sljedeća se plava sfera nalazi u daljini, opkoljena grupom neprijatelja koji će u početku biti previše teški za poraziti zbog nedostatka resursa na strani igraća. Zato se potiče igrač na potraži sljedeću sferu u skloništu čiji je ulaz na drugom kraju početne razine. Pored ulaza igrač nailazi na PGT pištolj, ali ne na metke.

Slika 23: Unutrašnjost astronautskog skloništa (autorski rad)

Istražujući unutrašnjost skloništa, igrač postepeno nailazi na neprijatelje i resurse poput paketa zdravlja, municije i baklji. Od triju soba do kojih vode dugački hodnici, jedna od njih sadrži živuće postolje od ticala koja se kreću po podu, usred kojeg je crvena sfera položena. Igrač po povratku može pronaći skriveno pakovanje velikog zdravlja, koje mu može obnoviti polovicu maksimalnog zdravlja.

Po izlasku iz skloništa, igrač bi trebao imati iskustva i resursa potrebnih da se izbori sa grupicom neprijatelja i pokupi zadnju plavu sferu sa poda. Po povratku do postolja na sredini početne razine i postavljanja svih sfera na njega, pokreće se finalna scena, u kojem se iza udaljene piramide otkriva gigantsko oko u nebu koje promatra igraća, popraćeno uznemirujućom glazbom.



Slika 24: Finalna scena igre (autorski rad)

7. Zaključak

Kroz ovaj rad prošli smo kroz cjelokupni proces razvijanja prototipa horor videoigre preživljavanja u prvom licu koristeći pokretač za igre Godot. U početnom dijelu rada smo ustanovili alate koje smo koristili kroz proces razvoja, uključujući i programerske i grafičke alate. Zatim smo se upoznali sa osnovnim konceptima Godota koji su nam dali kontekst potreban da bi razumjeli daljnje teme i sekcije gdje se razvija sama igra.

Razvoj igre smo počeli sa početnom testnom razinom koja je imala rudimentarne postavke nužne za testiranje daljnjih elemenata igre. Posvetili smo zatim veći dio vremena na razvoj klase kontrolera za igrača kako bi mogao izvršavati interakciju s igrom, te na temelje sustava inventara za stavke koje će igrač upotrebljavati tijekom igre. Nakon toga smo dizajnirali neprijatelje i definirali njihova ponašanja, razvili niz oružja kako bi se igrač mogao braniti od njih, te definirali vizualni stil igre koristeći programe za sjenčanje. Na kraju smo definirali ciljeve prototipa igre koji se trebaj ispuniti kako bi igrač postigao pobjedu u igri. Finalni rezultat je atmosferična horor igra preživljavanja sa retro vizualnim stilom koji podsjeća na rane 3D horor igre iz 90-tih.

Postoji niz poboljšanja koja bi se mogla napraviti na ovom projektu. Za početak, sustav učitavanja razina bi se mogao refaktorirati kako bi razine i njihovi prijelazi koji ih povezuju mogli biti više modularni i da se ne moraju pismeno definirati u Global klasi. Također bi se moglo bolje ostvariti sjenčanje koje ostvaruje dobiveni retro stil, jer trenutni pristup koji je korišten je iz starije verzije Godota i koristi viewportove za svaki sloj sjenčanja, što troši više resursa i procesorske snage računala nego što bi trebalo sa novijim metodama koje su implementirane. Nakon toga bi bio zadovoljan ako mogu nadodati još jednu vrstu neprijatelja, koja se ne mora spaljivati i koju bi mogao instancirati u većem broju, te još barem dvije razine koje bi igrač mogao istraživati. Još jedan dio igre kojem bio osobito trebalo dati pozornosti je zvučni dio, jer uz iznimku kod atmosferske muzike, ne postoje nikakvi drugi audio efekti u trenutnoj igri. Trebalo bi pronaći dobar izvor audio efekata s otvorenim licencama, ili bi trebao sam kreirati efekte i montirati ih u Audacity alatu.

Popis literature

Aseprite. (2024, kolovoz 9). *Aseprite/INSTALL.md at main · aseprite/aseprite*. GitHub. https://github.com/aseprite/aseprite/blob/main/INSTALL.md

Available 3D formats. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/tutorials/assets_pipeline/importing_3d_scenes/tutorials/assets_pipeline/importing_3d_scenes/available_formats.html

Blender Foundation. (2019). Logo. *Blender.Org*. https://www.blender.org/about/logo/

Capello, D. (2024). *Aseprite Documentation*. https://www.aseprite.org/

Carson, M. A., Rouen, M. N., Lutz, C. C., & James W. McBarron, I. I. (1975). *Extravehicular mobility unit*. https://ntrs.nasa.gov/citations/19760005607

CharacterBody3D. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/classes/classes/class_characterbody3d.html

DevWorm (Voditelj). (2023, listopad 26). *How to Create a INVENTORY in Godot 4 (step by step)* [Video recording]. https://www.youtube.com/watch?v=X3J0fSodKgs

Environment. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/classes/classes/class_environment.html

Fahs, T. (2009, listopad 30). IGN Presents the History of Survival Horror. *IGN*. https://www.ign.com/articles/2009/10/30/ign-presents-the-history-of-survival-horror

GDQuest (Voditelj). (2021, listopad 30). *3D Movement in Godot in Only 6 Minutes* [Video recording]. https://www.youtube.com/watch?v=UpF7wm0186Q

glTF 2.0—Blender 4.2 Manual. (2024, rujan 8). [2024]. https://docs.blender.org/manual/en/4.2/addons/import_export/scene_gltf2.html

Internal rendering architecture. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/contributing/development/core_and_modules/contributing/development/core_and_modules/internal_rendering_architecture.html

Introduction to shaders. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/tutorials/shaders/introduction_to_shaders.html

Kasavin, G. (2007, lipanj 25). Resident Evil: Deadly Silence Review. *GameSpot*. https://www.gamespot.com/reviews/resident-evil-deadly-silence-review/1900-6143721/

Mike. (2024, siječanj 29). Game Engine Popularity in 2024. *GameFromScratch.Com*. https://gamefromscratch.com/game-engine-popularity-in-2024/

Nodes and Scenes. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/getting_started/step_by_step/getting_started/step_by_step/nodes_and_scenes.html

Obsidian Brand Guidelines. (2024). https://obsidian.md/brand

Resources. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/tutorials/scripting/tutorials/scripting/resources.html

Richards, P. W. (1997). *Pistol grip torque-measuring power tool*. https://ntrs.nasa.gov/citations/20080004626

Soloski, A. (2020, kolovoz 7). Gods, Monsters and H.P. Lovecraft's Uncanny Legacy. *The New York Times*. https://www.nytimes.com/2020/08/07/arts/television/hp-lovecraft.html

Thorndyke, K. (2021, rujan 16). 8 Popular IDEs and Code Editors. *Codecademy Blog*. https://www.codecademy.com/resources/blog/popular-ides-and-code-editors/

Using signals. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/getting_started/step_by_step/signals.html

Using Viewports. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/tutorials/rendering/tutorials/rendering/viewports.html

Visual Studio Code. (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/contributing/development/configuring_an_ide/contributing/development/configuring_an_ide/visual_studio_code.html

VS Code icons and names usage guidelines. (2019). https://code.visualstudio.com/brand

What is GDExtension? (2024). Godot Engine Documentation. https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/tutorials/scripting/gdextension/what_is_gdextension.html

whiteshampoo. (2022, lipanj 9). Color reduction and dither. *Godot Shaders*. https://godotshaders.com/shader/color-reduction-and-dither/

zuwiwano. (2023, travanj 24). CRT Shader. *Godot Shaders*. https://godotshaders.com/shader/crt-shader-2/

Popis slika

Popis slika treba biti izrađen po uzoru na indeksirani sadržaj, te upućivati na broj stranice na kojoj se slika može pronaći.