Pogon otvorenog koda za izradu 2D i 3D igara Godot

Faust, Kristijan

Sveučilište u Rijeci – Odjel za informatiku

Diplomski studij informatike - Informacijsko komunikacijski sustavi

Kristijan Faust

Pogon otvorenog koda za izradu 2D i 3D igara Godot

Diplomski rad

Mentor: dr. sc. Vedran Miletić

Rijeka, 10. studenog 2018.

Sažetak

Godot je pogon otvorenog koda za izradu 2D i 3Digara dostupan pod MIT licencom. Razvojna okolina radi na Linux, macOS, Windows i drugim operacijskim sustavima, a moguće je izrađivati igre za osobna računala, konzole te mobilne i mrežne platforme.

Sam pogon realiziran je kroz C++ programski jezik, dok izrada igara moguća je korištenjem programskog jezika C++, C# ili GDScript. GDScript je programski jezik specifično osmišljen za potrebe pogona te je sintaktički vrlo sličan programskom jeziku Python. Osim širokog izbora programskih jezika za skriptiranje igara, pogon nudi i vlastiti sustav čvorova i scena koji je osmišljen za intuitivno i modularno rastavljanje i slaganje logičkih i apstrakcijskih cjelina prilikom izrade video igara, te kao alternativna solucija za već postojeće uzorke dizajna.

Iz priloženog sadržaja može se uvidjeti širok obujam pogona te njegov naglasak na jednostavnost realizacije funkcionalnosti, kao i svojstven pristup prema samoj izradi igara uz pomoć Godot pogona. I premda u trenutnom stanju Godot nije bolja alternativa ostalim popularnim pogonima na tržištu, kad su u pitanju veće zajednice razvojnih programera koji su navikli raditi u okolinama sa propisanim pravilima, i sam broj funkcionalnosti koje pogon pruža, svakako je najbolji izbor za početnike, rekreativce, brzo prototipiranje ideja, ozbiljnije projekte manje i srednje veličine te sa ne prevelikim brojem razvojnih programera, no i za veće ozbiljnije projekte koje uzimaju u obzir te imaju vremena i interesa za nadogradnju samog pogona sa funkcionalnostima za vlastite potrebe.

Ključne riječi

Godot, pogon, izrada igara, čvor, scena, GDScript

Sadržaj

1. Uvod

Kad su u pitanju moderna razvojna okruženja, većina domena nudi širok broj alata koji se trude korisniku što je više moguće automatizirati i olakšati rad i konfiguraciju sa tehničkim stvarima koje se ne tiču direktno krajnjeg proizvoda, kako bi korisnik mogao posvetiti što više pažnje ostalom sadržaju proizvoda te kako bi se određen posao obavio sa što manje prepreka u što manje vremena.

Za razliku od većine domena, industrija video igara je do nekoliko godina unatrag imala relativno oskudan i stalan izbor alata kad su u pitanju bili pogoni za izradu video igara. Razlozi tome su mnogobrojni te ne pomaže ni činjenica da su pogoni za izradu i izvedbu video igara jedni od najkompleksnijih i najopširnijih alata za razvijanje. Toj situaciji pridonosio je i afinitet tvrtki koje primarno izrađuju video igre, da same izrađuju vlastite pogone prilagođene njihovim potrebama te isključivo za njihovu upotrebu. Neki od tih pogona bi sa određenim godinama dovoljno i sazreli te bi ih tvrtka potom nudila ostatku tržišta i rekreativncima koji imaju afiniteta naučiti ili okusiti zanat izrade video igara. Najpoznatiji primjer takvog pogona je Unreal pogon koji trenutno broji svoje četvrto izdanje. Tek se u nekoliko zadnjih par godina unutar industrije video igara, da primjetiti porast tvrtki koji kao svoj primarni proizvod nude pogon a ne video igru, te uz to rastući afinitet tvrtki koje izrađuju video igre da uopće koriste gotove pogone umjesto da izrađuju svoje. I premda za isprobavanje te za samu izradu video igre, spomenuti pogoni u svojim osnovnim inačicama su u pravilu besplatni, za samo izdavanje igre kao i za otključavanje svih funkcionalnosti tih pogona gotovo uvijek je potrebno platiti određenu novčanu svotu. Osim toga, pogoni za izradu video igara u pravilu gotovo nikada nisu otvorenog koda. Iznimka tom pravilu je i popularni Unreal pogon, no s obzirom na licenciranje koje dolazi uz njegovo korištenje, nemože se smatrati FOSS softverom.

No 2014. godine izlazi Godot pogon. Godot pogon je u potpunosti besplatan, pod MIT licencom, te je otvorenog koda. Radi svega navedenog ne iznenađuje činjenica da do danas pogon broji preko šesto osamdeset kontributora te preko tri tisuće i osamsto forkova na github platformi, te da je u kratko vrijeme u kojem je prisutan na tržištu, zaživio zavidnu zajednicu korisnika.

Cilj ovog rada je dati generalni opis pogona Godot, analizirati njegov dizajn i opisati razvojnu okolinu s osobitim naglaskom na GDScript programski jezik, te opisati proces pakiranja igara za distribuciju na raznim platformama.

2. Općenito o pogonu

Godot je besplatan pogon za izradu 2D i 3D igara, upotpunosti otvorenog koda, izdan pod MIT licencom. Radno okruženje pogona u potpunosti je funkcionalno na Linux, BSD, Haiku, Windows i macOS operativnim sustavima, a aplikacije izrađene pomoću pogona mogu ciljati na okolinu osobnih računala, igračih konzola (neslužbeno), mobitela i mrežnu platformu.

Na službenoj stranici, pogon se reklamira kao veliki skup uobičajenih alata za izgradnju video igara sa kojima se korisnik može posvetiti isključivo njihovoj izgradnji bez da „izmišlja toplu vodu“. Neki od ključnih alata koje pogon nudi su:

- Vizualni uređivač
- Sustav čvorova i scena
- Vlastiti skriptni jezik
- Sustav modula
- Zasebne prikazivače za 2D i 3D okoline
- Pogone za fiziku

Godot se trudi pružiti cijelovitu soluciju za realizaciju svih dijelova produkcije video igre, osim za sam kreativni sadržaj određene igre. Kao i većina modernih pogona svi resursi kojima pogon raspolaže reprezentirani su datotekama na određenom datotečnom sustavu što uvelike olakšava kolaboraciju i verzioniranje samih proizvoda.

Jedan od najprepoznatljivij atributa pogona je njegov sustav čvorova i scena. Pomoću njega korisnik može na intuitivan način djeliti kompleksne sustave i probleme na manje cijeline po vlastitom kriteriju. Sustav čvorova i scena nudi alternativu za mnogobrojne uzorke dizajna koji već godinama pokušavaju standardizirati strukturiranje koda unutar video igara i sličnih aplikacija.

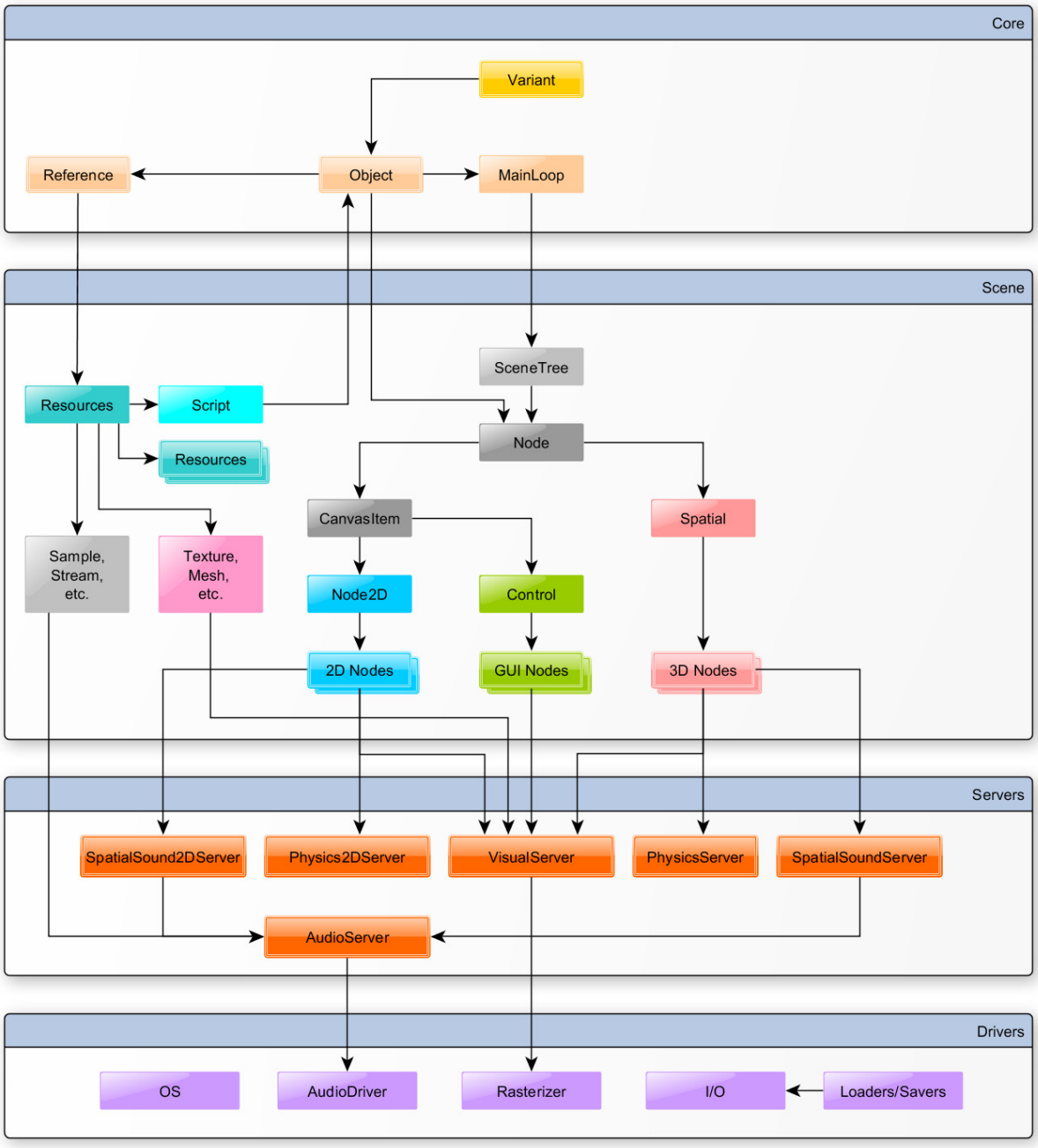
Razvoj Godot pogona započeli su Juan 'reduz' Linietsky i Ariel 'punto' Manzur 2007. godine. Kod pogona postao je javno dostupan tek 2014. godine u veljači na Github platformi. Danas se razvojni tim Godot pogona sastoji od mnoštvo kontributera iz zajednice otvorenog koda, specifično njih preko šesto pedeset prema službenom brojaču na Github platformi u vrijeme pisanja ovog poglavlja.

Mnogo kontributera i članova zajednice otvorenog koda koji su radili na pogonu, kao i mnoštvo korisnika pogona, dostupno je i otvoreno za upite i rasprave na popularnim komunikacijskim

platformama kao što su IRC i Discord, te je sama zajednica od velike pomoći novo-pridošlicama ne samo u korištenju Godot pogona već u izradi video igara općenito.

3. Pregled osnova i strukture internog koda pogona

Godot pogon pisan je u C++ programskom jeziku. Osnovni temelj pogona sastoji se od 4 apstrakcijskih slojeva koji zajedno omogućuju izvođenje Godot aplikacija.



Slika 1: Temeljna arhitektura Godot pogona

Kod za prikazani temelj pogona izvodiv je na C++98 ISO standardu, dok za dodatne module pogona kao što je sam uređivač scena i ostalo, zahtjeva kompajliranje koda minimalno po C++03

ISO standardu.

Za definicije tipova podataka Godot koristi standardne C98 tipove [3]. Za osnovnu alokaciju memorije Godot koristi gomilu. S obzirom na današnje prosječne količine RAM memorije u uređajima za osnovne alokacije pogona (npr. objekti, reference itd...) gomila kao model je pouzdana. Problemi mogu nastati prilikom alokacije većih količina podataka radi neefikasne segmentacije gomile te s toga Godot pogon nudi kao mogućnost i dinamičnu alokaciju memorije izvan gomile. Kako bi se osigurala kompatibilnost sa samim modelom rada memorije, Godot nudi i vlastite makro naredbe za rad sa memorijom te je preporučeno da programer koristi njih a ne funkcije standardnih biblioteka C/C++ programskih jezika za rad sa memorijom [3]. Naredbe memalloc(), memrealoc() memfree(), zamjenjuju standarde C funkcije malloc(), realloc() i free() a naredbe memnew()/memdelete() i memnew_arr()/memdelete_arr() zamjenjuju C++ naredbe new/delete i new[]/delete[]. Za dinamično alociranje i upravljanje memorije izvan gomile Godot pogon nudi vlastiti vektor predložak DVector<> koji je zapravo standardni vektor iz C++ biblioteke sa dodatnim pomoćnim funkcijama za čitanje i zapis podataka.

Godot pruža i vlastiti skup modificiranih spremnika, koji služe kao zamjenama standardnim C++ vektorima, listama, skupovima i mapama. Osim toga raspolaže vlastitom klasom za znakovne nizove koja posjeduje potpunu podršku za UTF-8 kodiranje i raspolaže ostalim potrebin pomoćnim funkcijama. Uz to pogon dolazi opremljen i sa mnoštvo predefiniranih klasa potrebnim za interni rad pogona no i za izradu samih igara.

3.1. Jezgra pogona

Najbitnija klasa jezgre pogona je Variant [3]. Variant klasa služi kao privremeni spremnik za gotovo sve tipove podataka unutar samog pogona. U većini slučajeva zauzima do 20 bajta po zapisu. Dodatna memorija se alocira isključivo za AABB klasu i klase matrica. Rijetko se koristi kao spremnik za podatke na duže vrijeme, već u generalu služi kao vrsta međuspremnika za komunikaciju, uređivanje, seriajlizaciju i premještanje podataka.

Klasa Object služi kao temeljna klasa za naslijeđivanje većine klasi unutar internog koda pogona, i kao temeljna klasa za naslijeđivanje svih čvorova (interno klasa Node) unutar Godot scena. Na taj način osigurava se da svaki čvor, odnosno objekti od kojih se sastoji određena aplikacija izvedena u Godot pogonu, sadrže određene zajedničke temeljne funkcionalnosti za rad i interakciju sa aplikacijom.

Reference klasa je klasa koja služi za spremanje i brojanje svih tipova referenci na resurse unutar

pogona. Resursi u kontekstu Godot pogona su bilo kakvi zapisi na datotečnom sustavu koji su dio podatkovnog sloja određenog čvora pogona, odnosno instanciranog objekta sa kojima poslužitelji pogona moraju interagirati. To mogu biti korisničke skripte, slike, modeli, muzika i ostalo. Efikasnost dohvaćanja takvih podataka od strane poslužitelja pogona realizirana je preko RID („Resource Identifier“) klase, koja garantira da će svaki resurs unutar određene aplikacije imati jednistveni identifikator te samim time može biti indeksiran i brzo dohvatljiv unutar RAM memorije.

MainLoop klasa služi kao temelj za nasljeđivanje svih glavnih petlji unutar Godot aplikacije. Kao i svaka aplikacija koja korisniku omogućuje interakciju i prikaz podataka u okvirno realnom vremenu, i Godot pogon koristi koncept glavnih petlji. U toj se petlji, za svaku iteraciju odnosno pokušaj prikazivanja okvira korisniku na ekran, odvija logika same aplikacije, od korisničkog koda iz skripti pa do automatizirane komunikacije sa poslužiteljima i upravljačima pogona. U izvdebi, klasa SceneTree iz sloja scene pogona naslijeđuje MainLoop klasu i pokreće glavnu petlju metodom iteration(). Nakon postavljanja svih vrijednosti varijabla okoline u toj metodi i prethodno u samoj konstrukciji klase, odvija se glavna petlja aplikacije koja redom vrši pozive na sve potrebne djelove slojeva pogona, kao što je na primjer poslužitelj pogona za fiziku, čije pozive i njihov redoslijed klasa MainLoop predefinira sama po sebi, dok je ostala specifična logika ugrađena u naslijeđenoj metodi iteration() unutar klase SceneTree:

```
1759        while (time_accum > frame_slice) {
1760
1761                uint64_t physics_begin = OS::get_singleton()->get_ticks_usec();
1762
1763                PhysicsServer::get_singleton()->sync();
1764                PhysicsServer::get_singleton()->flush_queries();
1765
1766                Physics2DServer::get_singleton()->sync();
1767                Physics2DServer::get_singleton()->flush_queries();
1768
1769                if (OS::get_singleton()->get_main_loop()->iteration(frame_slice * time_scale)) {
1770                        exit = true;
1771                        break;
1772                }
1773
1774                message_queue->flush();
1775
1776                PhysicsServer::get_singleton()->step(frame_slice * time_scale);
1777
1778                Physics2DServer::get_singleton()->end_sync();
1779                Physics2DServer::get_singleton()->step(frame_slice * time_scale);
1780
1781                time_accum -= frame_slice;
1782                message_queue->flush();
1783
1784                physics_process_ticks = MAX(physics_process_ticks, OS::get_singleton()->get_ticks_usec() - physics_begin); // keep
1785                physics_process_max = MAX(OS::get_singleton()->get_ticks_usec() - physics_begin, physics_process_max);
1786                iters++;
1787                Engine::get_singleton()->_physics_frames++;
1788        }
1789
```

Slika 2: Kod glavne petlje

Na liniji 1769 prikazanoj na slici 2, vrši se poziv na dodatni korisnički modul glavne petlje u kojem korisnik može izraditi vlastitu logiku i redoslijed odvijanja iste za aplikaciju, bez da mjenja ikakvu već postavljenu logiku pogona. Godot pogon sam po sebi već pruža predefinirani korisnički modul za glavnu petlju u kojemu se odvija poziv na upravljača korisničkih ulaza i izlaza pogona, te u koliko korisnik ne pridonese vlastiti kod, sam korisnički modul je također dio predefiniranog ponašanja glavne petlje pogona.

3.2. Sloj scene

Sloj scene je prezentacijski sloj pogona te je ujedno najbitniji sloj za korisnike pogona. Pogon je osmišljen da korisnik u idealnom slučaju kroz ovaj sloj gradi kreativni dio aplikacije te vrši interakciju sa ostalim slojevima i automatiziranim internim modulima pogona.

Najbitnija klasa ovog sloja je prethodno spomenuta SceneTree klasa. Osim naslijeđivanja i pokretanja glavne petlje, a samim time i korisničke aplikacije, ova klasa za zadatak ima sadržavanje početnog prikaza na koji se dodaje ostatak scene. Scena u kontekstu Godot pogona je samo određeni skup čvorova. Čvorovi u Godot pogonu su instancirani objekti unutar korisničke aplikacije koji reprezentiraju elementarne cijeline same aplikacije, odnosno korisnički kreativni sadržaj izveden u samom pogonu. Sami koncepti prikaza, scena i čvorova su koncepti viših razina apstrakcije i korisničkih slojeva pogona, te će biti objašnjeni u pripadajućim poglavljima. Osim spomenutog klasa SceneTree posjeduje sve potrebne metode dohvačanja instanciranih čvorova i pripadajućih grupa čvorova kao i izvedbe nekih globalnih funkcionalnosti samih aplikacija kao što je pauziranje aplikacije. Klasa raspoznaje i redoslijed instanciranja čvorova kao i njihove hijerarhije i odnose.

Osim SceneTree klase sloj sadrži i temeljnu klasu čvorova Node, kao i sve njene podklase, te sve resurse korištene u određenoj sceni.

Na sloj scene može se gledati kao na najviši apstrakcijski sloj pogona, odnosno kao na sam pogon, koji uz pomoć komunikacije preko API sučelja sa ostatakom slojeva niže razine izvode svoju zadaću.

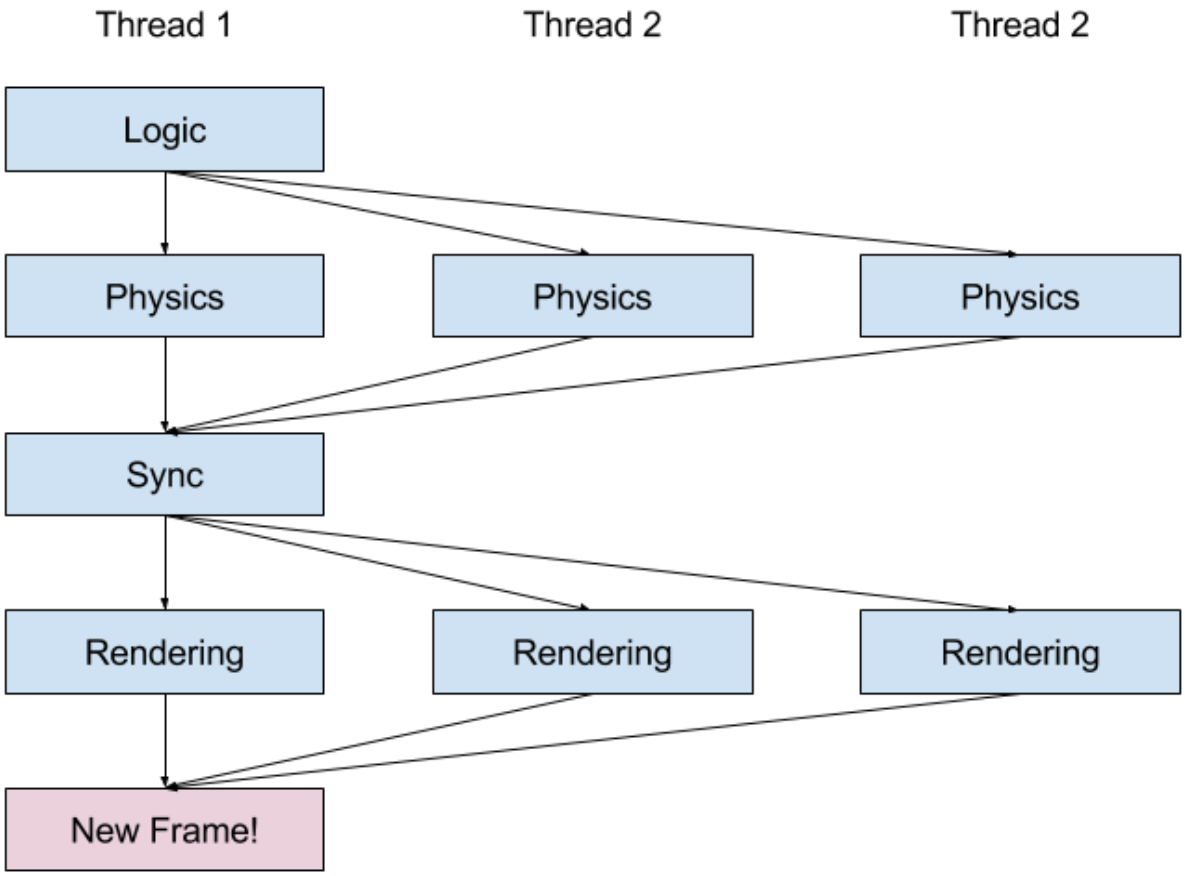
3.3. Sloj poslužitelja

Jedan od prepoznatljivijih atributa Godot pogona su njegovi poslužitelji, odnosno sustav poslužitelja preko kojih pogon vrši svoje zadaće te su oni ujedno i način izvedbe paralelizacije

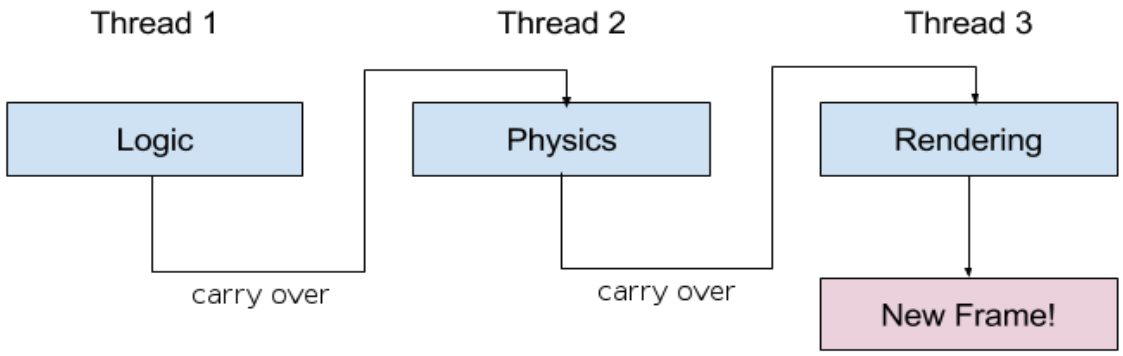
raznih zadaća pogona. Poslužitelji u okruženju pogona su pozadinski procesi, odgovorni za upravljanje podacima i procesima okruženja te na psolijetku i posluživanju rezultata upravljačima i ostalim modulima pogona koji te podatke trebaju. Poput mrežnih poslužitelja, sadrže sve potrebne podatke i stanja odgovarajuće domene te svoje zadaće izvode paralelno[4]. Osim toga, sva komunikacija sa poslužiteljima izvodi se preko istog kanala a sami korisnici nemaju direktan pristup njima. Poslužitelji implementiraju posrednički uzorak (eng. mediator pattern) s kojim interpretiraju RID identifikatore pogona i obrađuju podatke.

Tipičan slijed izvedbe apstrakcijskih cijelina prilikom rada generičnog pogona za video igre bio bi redom izvedba logike igre, potom fizike i ostalih predefiniranih uvjeta okoline te na poslijetku prikazivanje sadržaja. Kako se taj redoslijed logički nemože izbjeći, većina popularnih pogona za video igre paralelizira što je više djelova moguće prilikom računanja fizike i prikazivanja sadržaja uz pomoć upravitelja poslova. U samom procesu simuliranja fizike postoji tek nekoliko koraka kod kojih je moguć visok stupanj paralelizacije izračuna te je najčešće pararelizacija primjenjena na problematike detekcije kolizija i izračuna poslijedica istih. Osim simulacije fizike, paralelizacija je nešto više primjenjiva na prikazivanju i iscrtavanju sadržaja.

Ukoliko bi podaci između prikazivanja sadržaja i ostalih apstrakcijskih cjelina tekli isključivo u jednom smjeru da se zaključiti da se sve cjeline mogu izvoditi gotovo u potpunosti paralelno. Izuzetak tom pravilu je kratak djelić izvedbe u kojem dolazi do komunikacije između logičkog djela aplikacije i poslužitelja fizike.



Slika 3: Tipičan model paralelizacije izvedbe ciklusa popularnih pogona za video igre



Slika 4: Model paralelizacije izvedbe ciklusa Godot pogona

Kako bi se pojednostavio interni kod a time i samo održavanje pogona, te korisnička interakcija sa pogonom, i kako bi se izbjegli mnogobrojni potencijalni problemi na strani korisnika sa zaključavanjem podataka za obradu, muteksima i semaforima, osmišljen je sustav poslužitelja[4].

Pogon dolazi sa nekoliko različitih poslužitelja od kojih se većina njih izvodi paralelno. To su:

- Physics2DServer – poslužitelj za simuliranje 2D fizike
- VisualServer – poslužitelj za prikazivanje sadržaja
- PhysicsServer – poslužitelj za 3d simuliranje fizike
- AudioServer – poslužitelj za obradu i rad sa zvučnim zapisima
- ARVRServer – poslužitelj za proširenu i virtualnu stvarnost (najnoviji poslužitelj)

Iako najčešće u praksi poslužitelji za 2D fiziku neće obavljati nikakve zadaće u 3D okolinama i obrnuto, svi navedeni poslužitelji izvršavaju svoje zadaće u potpunosti paralelno. Iznimka je do verzije pogona 3.0 bio AudioServer poslužitelj što se da uvidjeti i na slici 1. Razlog tome je bila podjela poslužitelja zvuka na više njih te je taj poslužitelj morao čekati dobivene izračune od strane SpatialSound2DServer/ SpatialSoundServer te ih uzeti u obzir prilikom ostalih manipulacija zvuka ukoliko je korisnik htio da okolina utječe na zvuk unutar igre. Unatoč tome u praksi su zadaće zvučnih poslužitelja često mnogo manje intenzivnije nego zadaće poslužitelja za fiziku ili poslužitelja za prikazivanje sadržaja te stoga je to sekvencijalno izvođenje bilo neprimjetno u većini slučajeva. Danas je utjecaj okoline na zvuk u potpunosti preuzeo jedan poslužitelj za sve zvučne zapise unutar pogona, što je i očito iz koda za alokaciju poslužitelja u datetoeci register_server_types.cpp:

```
178  void register_server_singletons() {
179          Engine::get_singleton()->add_singleton(Engine::Singleton("VisualServer", VisualServer::get_singleton()));
180          Engine::get_singleton()->add_singleton(Engine::Singleton("AudioServer", AudioServer::get_singleton()));
181          Engine::get_singleton()->add_singleton(Engine::Singleton("PhysicsServer", PhysicsServer::get_singleton()));
182          Engine::get_singleton()->add_singleton(Engine::Singleton("Physics2DServer", Physics2DServer::get_singleton()));
183          Engine::get_singleton()->add_singleton(Engine::Singleton("ARVRServer", ARVRServer::get_singleton()));
184  }
```

Slika 5: Alokacija predefiniranih poslužitelja unutar koda pogona

Osim spomenutih poslužitelja korisnik može naknadno implementirati vlastite poslužitelje po potrebi. Kao minimum poslužitelj mora sadržavati statičnu instancu po uzorku jedinca (singleton pattern), vremenski brojač, zasebnu nitnu petlju, inicijalno stanje i proceduru čišćenja. Korisnički poslužitelj se također treba alocirati kao i predefinirani poslužitelji.

3.4. Upravljački sloj

Komunikaciju između pogona odnosno aplikacije realizirane u pogonu i operativnog sustava vrši upravljački sloj pogona. Iako apstraktne granice upravljača nisu striktno definirane, minimalna

podjela upravljača sastojala bi se od upravljača za ulaz i izlaz, upravljača za prikazivanje, upravljača za zvuk i glavnog upravljača.

Upravljači za zvuk i prikazivanje namjenjeni su za komunikaciju sa više različitih pozadisnkih API sučelja, te su kao takvi rađeni na višoj razini apstrakcije nego što bi to tipičan upravljač jednog pogona za te domene bio, kako bi se održala što veća razina modularnosti komunikacijskih kanala za buduće dodatke. Trenutni upravljač za prikazivanje podržava GLES2, i predefinirano GLES3+ (radi i sa OpenGL 3.3. na osobnim računalima) API sučelje za iscrtavanje 2D i 3D grafike, dok zvučni upravljač podržava nekoliko sučelja za reprodukciju zvuka za razne platforme a to su WASAPI, Xaudio2, RtAudio, Winmidi, Coremidi, Alsa i predefinirani Pulseaudio.

Upravljači za ulaz i izlaz vrše zadaće komuniciranja sa datotečnim sustavima operativnih sustava kao i mrežnom komunikacijom putem UDP i TCP protokola odnosno HTTP protokola.

Glavni upravljač pogona implementiran je u klasi OS. OS klasa je prva instancirana klasa prilikom pokretanja aplikacije u Godot okruženju, te se preko nje instanciraju svi ostali objekti. Ovoj klasi se također proslijeđuje i glavna petlja prilikom inicijalizacije aplikacije, koja je u predefiniranim uvjetima implementirana u klasi SceneTree. Zajedno sa velikim brojem dodatnih pomoćnih klasa odnosno modula pogona, glavni upravljač vrši zadaću upravljanja pogona i komunikacije sa samim operativnim sustavom.

4. Pregled osnova i strukture internog koda pogona

4.1. Čvorovi

Osnovni koncept i gradivna jedinica aplikacija izrađenih pomoću Godot pogona je čvor (izv. Node). U tehničkom smislu čvorom se može smatrati svaki instancirani objekt unutar sloja scene koji pripada klasi Node ili nekoj od klasa koje naslijeđuju klasu Node. U praktičnom smislu na razini korisničkog sloja pogona, čvor je određena vrsta gradivnog elementa aplikacije sa vlastitim skupom funkcionalnosti koje mu omogućuju da na određen način izvršava određene zadatke.

U svojoj suštini čvor je samo naziv za objekt unutar okruženja pogona. Svaki čvor unutar pogona posjeduje vlastito ime. Ime nemora biti jedinstveno u određenom okruženju no zato njegova putanja mora. Putanja se sastoji od imena čvora i hijerarhije čvora unutar stabla scene. Na taj način pogon implementira vlastiti imenski prostor. Svi čvorovi mogu posjedovati atribute i metode analogno klasičnim objektima iz objektno orijentiranih programskih jezika. Čvorovima se mogu proslijeđivati i određeni signali odnosno događaji pa samim time čvorovi imaju mogućnost izvedbe povratnih funkcija na određene događaje unutar pogona. Najbitnije svojstvo povratnih funkcija kod čvorova je da svaki čvor može sadržavati povratnu funkciju na svaki prikazani okvir unutar pogona ali također i na svaki okvir izvedbe fiksnog intervala. Svaki čvor može biti proširen dodatnim funkcionalnostima drugog čvora, analogno naslijeđivanju u ostalim programskim jezicima. Najbitnije svojstvo kod čvorova je mogućnost izgradnje veza roditelj-dijete između više čvorova. Tako dobivena stabla su glavni način strukturiranja i organizacije logike i koda Godot aplikacija, ali i veoma moćan alat prilikom upravljanja i organizacije kompleksnosti aplikacije.

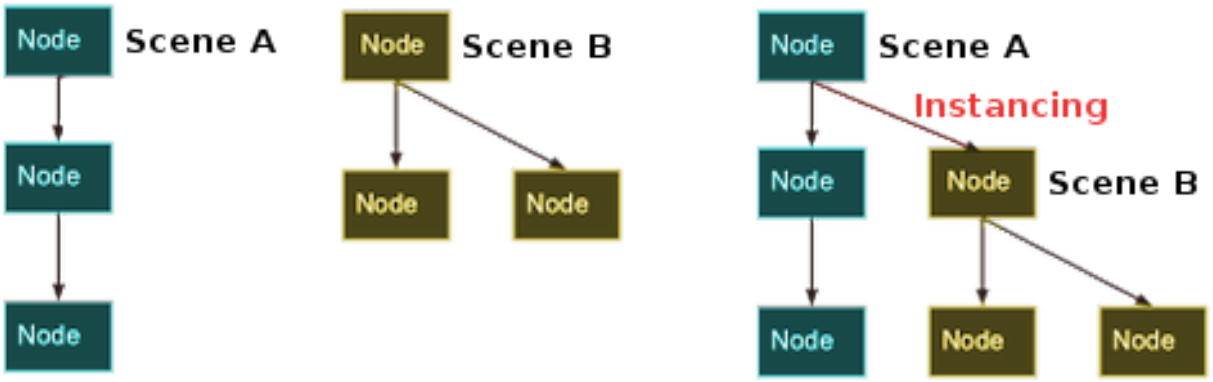
4.2. Scene

Scene u Godot pogonu su skupovi čvorova organizirani u hijerarhiju stabla. Svaka scena ima svoj kroijenski čvor. Glavna razlika između čvora i scene je ta da se zapis scene može spasiti na fizičke medije za pohranu podataka. Zapise scena pogon pohranjuje u obliku .tscn datoteka.

Pokretanje igre ili aplikacije napravljene u pogonu je zapravo pokretanje jedne određene scene. Projekt se može sastojati od mnogo scena ali da bi se projekt pokrenuo, jedna scena mora biti odabrana kao glavna scena.

Najbitnije svojstvo scena je to da scene mogu instancirati druge scene. Prilikom instanciranja scene

unutar scene, relacije roditelj-dijete između čvorova se prenose ovisno o samom čvoru koji instancira određenu scenu unutar scene. Na taj se način veoma kompleksne cijeline mogu apstrahirati na puno manje logičke cijeline te razvijati neovisno o ostatku projekta po asptraktnim mjerama korisnika a ne alata. Isto tako, scene se mogu intuitivno dodavati vlastitim cijelinama bez potrebe za poštivanjem ikakvog nametnutog uzorka dizajna.



Slika 6: Dijagram instanciranja scene unutar scene

Upravo ovaj pristup izgradnji projekata je vjerovatno i najupečatljiviji atribut rada sa samim pogonom no isto tako možda i najmoćniji alat koji ovaj pogon nudi korisniku na raspolaganje. Ovim pristupom izgradnje projekata pogon se uvelike razlikuje od većine popularnih pogona za izgradnju video igara poput Unity ili Unreal pogona, koji svoje pristupe temelje na uzorcima dizajna kao što su MVC ili entitet-veza diagrami. Ti su uzorci dizajna već godinama afirmirani u izgradnjama aplikacija unutar određenih domena, no ne i u industriji video igrara.

4.2.1. „Text SCeNe“ datoteka

Pogon spremljene scene zapisuje u obliku .tscn datoteka (skraćeno od „Text SCeNe“). Svaka .tscn datoteka reprezentira jedno stablo scene unutar aplikacije koju interna klasa pogona SceneTree učitava te upravlja zajedno sa ostalim učitanim scenama.

Najbitnije svojstvo tih dateteka je da su lako čitljive korisnicima, te time se scene mogu graditi i uređivati koristeći .tscn datoteke, iako taj način nije preporučljiv. Prilikom pokretanja scena ili izvoza projekta .tscn datoteke se kompajliraju u binarne datoteke .scn kako bi se smanjila potrebna količina podataka za čuvanje i vrijeme učitavanja scena.

Sam opisni jezik kojim se realiziraju .tscn datoteke je u suštini poprilično jednostavan, no podrazumijeva dobro poznavanje svih korištenih Godot čvorova i funkcionalnosti unutar određene scene kao i njihova predefinirana ponašanja. Primjer minimalnog čvora svijetlosti u 3D okolini

opisanog .tscn datotekom izgledao bi:

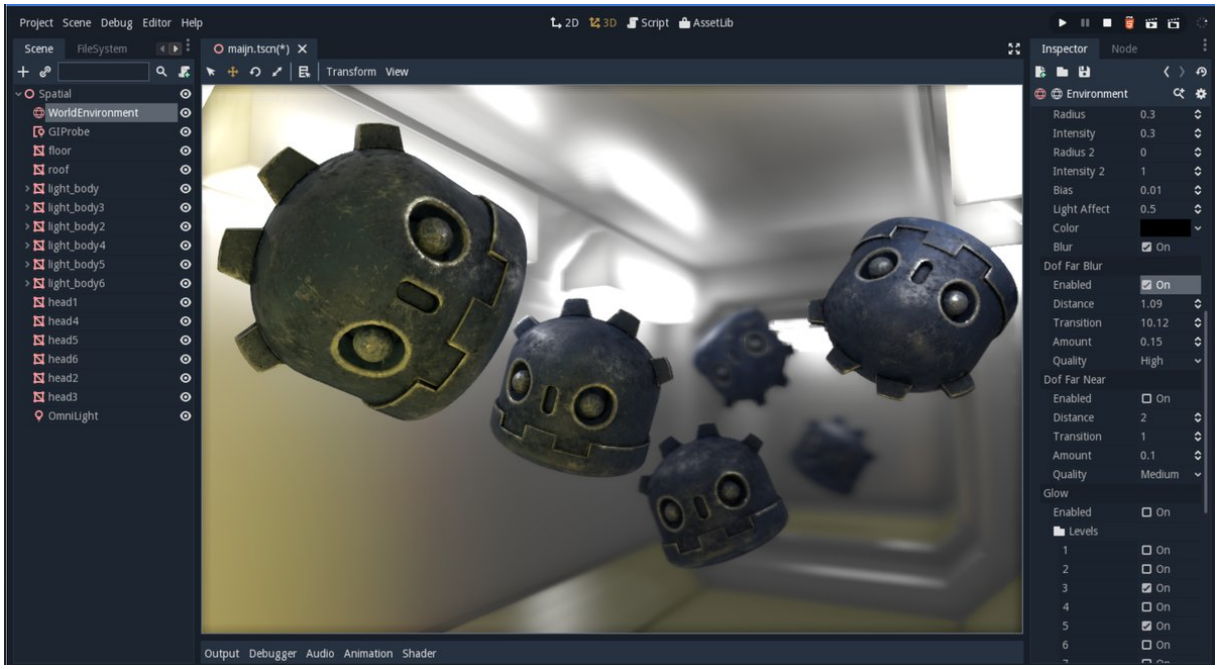
```
[node  type="OmniLight" name="Lamp" parent="."]

light_energy = 1.0
light_specular = 1.0
transform = Transform( -0.29086464643478394 , -0.7711008191108704 , -1.0054539442062378 )
omni_range = 30
shadow_enabled = true
light_negative = false
light_color = Color( 1.0, 1.0, 1.0, 1.0 )
```

4.3. Resursi

Resursi unutar pogona su podatkovni kontenjeri. Sami po sebi ne izvode nikakve akcije ali su intenzivno korišteni od strane čvorova i ostalih modula pogona. Služe za učitavanje i radom sa teksturama, skriptama, mrežama poligona, animacijama, zvukom i ostalim kreativnim sadržajem igre odnosno aplikacije. Bilo kakvo učitavanje od strane pogona nekakvih podataka sa spremnika proizlazi rezultirajućim resursom unutar pogona. Jednom učitani resurs u memoriju neće se ponovno učitati dokle god se taj resurs ne izbriše iz memorije. Bitno je napomenuti da osim čvorova i sami resursi mogu sadržavati druge resurse unutar okoline pogona. Resursi naslijeđuju klasu Reference te samim time i njen brojač referenci kojai osigurava automatsko brisanje iz memorije jednom kad je vrijednost brojača dosegla nulu.

4.4. Godot uređivač

Godot pogon dolazi u paketu sa veoma opremljenim i modularnim uređivačem, po uzoru na većinu modernih i popularnih pogona za video igre. U svojoj suštini, Godotov uređivač je primarno uređivač scena, premda mnogobrojne naprednije funkcionalnosti uređuju i mnogo stvari izvan domene sloja scene. Uz to, uređivač je u potpunosti izveden preko Godot API sučelja, napisan u C++ programskom jeziku, te je s time on zapravo primjer Godot igre, odnosno dokaz da je Godot pogon i njegovo okruženje i više nego kadro za izgradnju i izvedbu aplikacija širih namjena nego što su to video igre.



Slika 7: Godot uređivač

Uređivač je izveden po uzoru na mnogobrojne druge moderne i popularne uređivače te kao i oni broji moge slične predefinirane module kao što su datotečni prikazivač, prikazivač scene, uređivač atributa čvorova, izlazna konzola, razbubnik (izv. debugger), animator i mnoge ostale. No za razliku od ostalih uređivača popularnih pogona, Godot uređivač je u potpunosti modularan od izgleda i rasporeda modula pa do uključenja osnovnih ili vanjskih modula uređivača trećih stranki. Uz to uređivač dolazi u paketu sa vlastitim uređivačem skripti što nije slučaj kod većine popularnih pogona za video igre. Vlastiti uređivač skripti je djelom iz razloga što pogon dolazi u paketu sa vlastitim skriptnim programskim jezikom, no prilagođeni uređivač skritpi za određeno okruženje uvelike pomaže i olakšava rad u samoj okolini, te su funkcionalnosti poput dopune teksta ili pretrage referenci dio samog uređivača. No ukoliko korisnik želi, uređivač se vrlo jednostavno može konfigurirati za rad sa bilo kojim drugim uređivačem teksta, skripti ili čak IDE alatima.

Dok mnogobrojni ostali pogoni se prema 2D okolinama odnose kao prema 3D okolini gdje prikazani elementi imaju svi jednaku dubinu, Godot pogon ima prilagođene interne pristupe za oba tipa okoline, pa tako i sam uređivač scena se djeli na 2D i 3D scene.

5. Skriptiranje

Godot pogon službeno podržava četiri programska jezika za komunikaciju sa Godot API sučeljem, odnosno za takozvano skriptiranje. To su GDScript, VisualScript, C# i C++. Postoje i suradnički projekti koji su uspješno integrirali i neke druge programske jezike u pogon, primjerice Python.

5.1. GDScript

Osnovni i najstariji programski jezik za skriptiranje unutar Godot pogona je GDScript (skraćeno od „Godot Script“). GDScript razvijen je specifično za potrebe skriptiranja unutar pogona nakon nezadovoljavajućih rezultata integracija programskih jezika Lua, Python i Squirrel. S obzirom da je i najstariji skriptni jezik pogona, daleko je najzreliji odabir između četiri ponuđena programska jezika.

GDScript je progamski jezik sa dinamičnim tipovima podataka, te je veoma sličan popularnim skriptnim programskim jezicima kao što su Python i Lua. Integriranost u uređivač pogona je na visokoj razini te je preko GDScripta tijek rada u pogonu vjerovatno i optimalniji, posebice zato jer nije potreban nikakav dodatan IDE za automatsku dopunu koda i slične funkcionalnosti. Unatoč njegovoj dinamičnoj prirodi, GDScript je relativno brz jezik kad su u pitanju vremena učitavanja i vrijeme interpretiranja koda, te se uvelike oslanja na niti za učitavanje podataka, nešto za što većina skriptnih programskih jezika dinamičnog tipa nije optimalno rješenje. GDScript okruženje ne implementira vlastiti sakupljač smeća (izv. Garbage collector), te s toga koristi direktno pogonsko upravljanje memorije. U sam jezik ugrađeni su i Godotovi tipovi podataka za 2D i 3D matematiku poput raznih vektora, matrica i ostalih struktura koji se koriste za programiranje video igara, te je stoga GDScript odličan odabir programskog jezika i za elegantno izvođenje operacija linearne algebre.

5.2. GDNative (C++)

GDNative je pogonovo sučelje koje dozvoljava skriptiranje pogona u C++ programskom jeziku. Iako je kroz GDNative sučelje pogona moguće integrirati podršku za velik broj programskih jezika, trenutna službena podrška je isključivo za C++ programski jezik. S obzirom da je GDNative sučelje implementirano preko mosta za Godot C++ API sučelje napisanog u C programskom jeziku, mješanje različitih implementacija i verzija kompajlera za C++ skripte je također moguće. Ova

mogućnost pogona je veoma korisna za djelove aplikacije, odnosno igre koje su zahtjevne za resurse računala.

5.3. C#

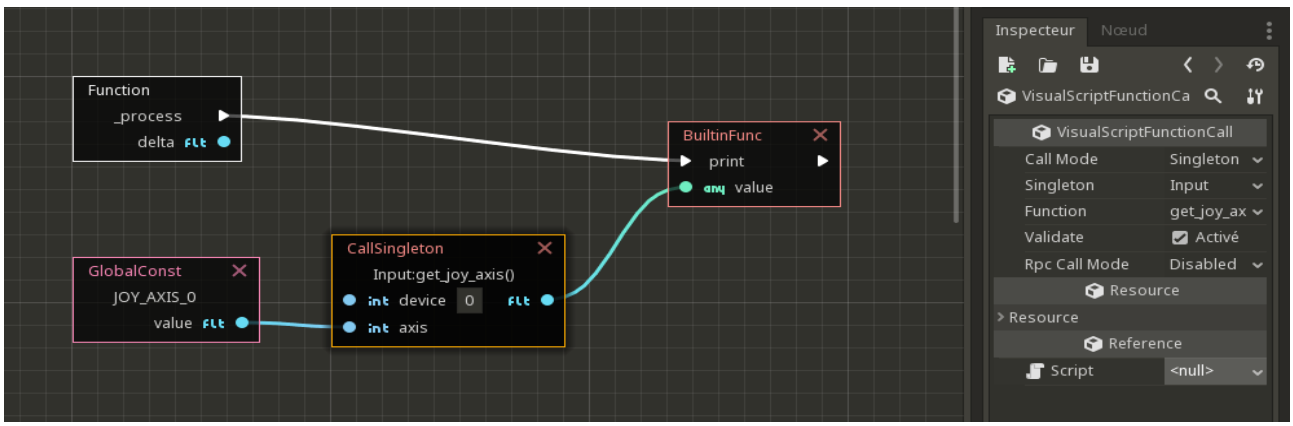
Kao solucije za veće timove ili tvrtke koje se odluče za razvojno okruženje Godot pogona, Godot nudi C# kao alternativu koja za određenu zamjenu u performansama nudi nešto poznatije i standardiziranije okruženje poznato većini sudionika industrije.

Inicijativu je pokrenuo Microsoft vlastitim donacijama projektu Godot pogona. Time pogon prati trend u industriji video igara od zadnjih nekoliko godina u kojem uporaba C# programskog jezika raste, kako kao odabrani programski jezik popularnih pogona tako i kao programski jezik internih alata popularnih timova.

No bitno je napomenuti kako uz C# dolazi i njegov sakupljač smeća, te su s time performanse izvedbe aplikacija nešto sporije, posebice zato jer okruženje u tom slučaju radi sa dupliciranim zapisom referenci na dva različita mjesta. Kao dio samog jezika u paketu dolazi i podrška za Mono .NET okruženje sa svojim korisnim funkcionalnostima.

5.4. VisualScript

Prema uzoru na mnogobrojne ostale pogone za video igre, Godot pogon nudi svoju inačicu vizualnog skriptiranja VisualScript. Kao i kod ostalih pogona, ovaj pristup skriptiranju je prvobitno namjenjen za početnike u programiranju ili korisnike pogona koji nisu programeri, već rade na drugim domenama igre (umjetnici ili dizajneri) a htjeli bi uvid u logiku igre ili mogućnost eksperimentiranja sa istom.



Slika 8: Primjer izgradnje logike sa VisulScript jezikom

VisualScript je tipičan blok-veza vizualni jezik koji je osmišljen specifično za rad sa internom logikom Godot pogona. Iako jezik ima sva svojstva Turingove mašine (izv. Turing complete), radi svoje naravi ne skalira dobro kad su u pitanju veće količine kako samog koda tako i njegove kompleksnosti. Osim za spomenute primjene, jezik mogu koristiti i programeri za dijagrame stanja i automate određenih logičkih cijelina.

5.5. Skripta

Svaki čvor unutar pogona dolazi sa vlastitim funkcionalnostima. Skripta je skup proširenih funkcionalnosti čvora od strane korisnika. U Godot okruženju svaki čvor može imati samo jednu skriptu koja mu pripada. Svaka skripta unutar okruženja mora imati pripadajući čvor. Čvor kojemu skripta pripada je djelokrug (izv. scope) skripte. Za postizanje globalnog djelokruga pogon nudi AutoLoad funkcionalnost, odnosno mogućnost instanciranja klase jedinca (izv. Singleton) na vrhu hijerarhije svih stabala scena.

Skriptiranje unutar Godot pogona osmišljeno je kroz nadjačavanje (izv. overriding) virtualnih funkcija ugrađenih u samu klasu Node. Na taj način te funkcije služe kao povratne funkcije (izv. callback) na određene događaje unutar aplikacije. Svaki čvor nasljeđuje osam osnovnih void povratnih funkcija od klase Node. To su:

- _enter_tree() - povratna funkcija na dodavanje čvora unutar stabla scene
- _exit_tree() - povratna funkcija na brisanje čvora unutar stabla scene
- _input(InputEvent event) - povratna funkcija koja obrađuje događaje korisničkog unosa
- _process(float delta) - povratna funkcija na svaki prikazani okvir
- _physics_process(float delta) - povratna funkcija na svaki fiksni okvir za obradu fizike
- _uhnadled_input(InputEvent event) - povratna funkcija na neobrađene unose
- _uhnadled_key_input(InputEventKey event) - povratna funkcija na neobrađene unose tipkovnice

Svaki tip čvora nadalje može definirati vlastite povratne funkcije na određene događaje unutar pogona te mnogi drugi tipovi posjeduju vlastite predefinirane povratne funkcije.

6. GDScript

Godot pogon osmišljen je primarno sa GDScriptom kao njegovim skriptnim jezikom. Radi toga, sam je jezik veoma usko povezan sa pogonom i njeovim API sučeljem, te su funkcionalnosti jezika direktno proširenje C++ baze pogona.

GDScript je programski jezik sa dinamičnim tipovima podataka, veoma sličan Python programskom jeziku po sintaksi. S toga relativno je jednostavan za naučiti. Osmišljen je sa minimalnošću potrebnog koda na umu, te s toga odličan je za brzo prototipiranje ideja sa relativno malo koda. Uz to osmišljen je po principima „duck typing“ paradigme i uvelike podržava i potiče polimorfizam. GDScript koristi vlastiti interpreter koji je dio pogona te preko kojeg se vrše potrebni pozivi na pogonovo C++ API sučelje.

Kao i u većini programskih jezika, svaki niz znakova sastavljen od znakova engleske abecede brojeva i znaka _, s uvjetom da niz ne započnje brojem, predstavlja kadnidata za identifikator unutar jezika, izuzev ključnih riječi koje se koriste za naredbe i izvedbu samog programskog jezika. Jezik je osjetljiv na velika i mala slova. Varijabla se deklarira sa ključnom riječi var prije identifikatora. Prilikom deklaracije varijable bez dodijele inicjalne vrijednosti, predefinirana vrijednost je tip null. Unarni i binarni operatori su po uzoru na popularnu većinu ostalih programskih jezika. Konstante se definiraju sa ključnom riječi const umjesto var a enumeratori sa ključnom riječju enum.

Komentarske linije započinju se znakom #. I dok za više linijske komentare trenutno jezik ne nudi direktno sintaktičko rješenje, unutar uređivača koda pogona CTRL + k stavlja komentarski znak ispred svih označenih linija.

```
#Ovo je komentar
var identifikator = 0
```

6.1. Tipovi podataka

GDScript dolazi u paketu sa nekoliko ugrađenih tipova podataka. Od primitivnih tipova podataka to su: null, bool, int, float i String. Osim osnovnih tipova GDScript nudi zavidan broj vektorskih tipova podataka. To su:

- Vector2 - Reprezentacija dvodimenzionalnog vektora koji sadrži x i y atribute.
- Rect2 - Reprezentacija četverokuta koji je definiran sa dva vektorska atributa naziva

position i size za poziciju i veličinu četverokuta. Sadrži i treći vektorski zapis naziva end čija je vrijednost uvijek jednaka zbroju pozicijskog i skalarnog vektora veličine.

- Vector3 - Reprezentacija trodimenzionalnog vektora sa x, y i z atributima.
- Transform2D - Reprezentacija matrice sa tri redka i dva stupca koja se koristi prilikom operacija transformacija u dvodimenzionalnog prostoru.
- Plane - Reprezentacija trodimenzionalne ravnine u normaliziranoj formi koja sadrži vektorski zapis naziva normal i skalar udaljenosti naziva d.
- Quat - Reprezentacija kvaternionske matematičke strukture koja se sastoji od četiri vrijednosti naziva w, x, y i z koji zajedno predstavljaju četverodimenzionalan kompleksan broj i kod koje množenje članova strukture nije komunitativno. Množenjem tih struktura mogu se reproducirati rotacijske sekvence objekata u trodimenzionalnom prostoru. Kao takvi uvelike su korisni prilikom izvedbe sferičkih linearnih interpolacija, odnosno takozvanih SLERP izračuna između dvije rotacije.
- AABB - Reprezentacija kvadra analogna Rect2 tipu podataka za trodimenzionalni prostor. Najčešće se koristi za brze provjere preklapanja određenih objekata.
- Basis - Reprezentacija matrice sa tri redka i tri stupca. Sastoji se od tri trodimenzionalnih vektora naziva x, y, i z. Gotovo uvijek služi kao ortogonalna baza za Transform tip podataka.
- Transform - Reprezentira matricu od tri redka i četiri stupca koja se koristi prilikom transformacija u trodimenzionalnom prostoru. Sastoji se od atributa s nazivom basis i tipa Basis i Vector3 tipa naziva origin.

Svi opisani vektorski tipovi podataka dolaze sa prilagođenim funkcijama za vlastite potrebe te s očekivanim funkcionalnostima kao primjerice zbrajanje dva vektora. Vector2, Vector3 i Basis mogu pristupati svojim atributima poput polja.

```
var vektor = Vector3(4.0, 3.0, 2.0) * Vector3(2, 3, 4)
print(vektor)
```

GDScript dolazi sa nekoliko ugrađenih tipova podataka specifičnih za Godot pogon. To su:

- Color - Reprezentacija boje unutar okoline pogona. Osnovni zapis boje sastoji se od četiri decimalne vrijednosti u intervalu [0, 1], naziva r, g, b i a za crvenu, zelenu i plavu vrijednost boje, te alfa faktor boje odnosno njena transparentnost. Klasa ima mogućnost zapisa

osnovnih vrijednosti i u osam-bitnom formatu u intervalu [0, 255]. Osim toga u klasu je ugrađeno nekoliko korisnih metoda za rad sa bojama, kao primjerice contrasted(), koja vraća najveći kontrast određene boje, kao i konstruktore koji primaju html vrijednosti boja ali i imena nekih tipičnih boja. Osim spomenutih osnovnih vrijednosti klasa posjeduje i atribute h, s i v za nijansu, saturaciju i vrijednost svijetlosti.

- Path - Reprezentacija putanje čvora unutar određenog stabla scene. Osim čvorova može referencirati i njihove atribute i resurse. Konstruktor prima isključivo znakovni niz odnosno String tip podataka, a putanja se tretira kao relativna ili apsolutna ovisno o obliku samog znakovnog niza, prema UNIX načelima putanji.
- RID - Reprezentacija vrijednosti identifikatora resursa. RID su neprozirni (izv. opaque) tipovi podataka, prvobitno namjenjeni korištenju od stranje pogonskih poslužitelja za dohvaćanje resursa. Klasa sadrži isključivo konstruktor koji prima klasu Object i metodu get_id() koja vraća cijelobrojni jedinstveni identifikator resursa. Ukoliko se konstruktoru proslijedi nevaljani resurs, dodjeljuje se nekorišteni identifikator sa vrijednošću 0.
- Object - Klasa koju naslijeđuju svi neugrađeni tipovi unutar pogona. Podrazumijeva od korisnika da vodi brigu upravljanja memorijom, osim u pojedinim slučajevima, primjerice kod korištenja derivata Reference, ili kod brisanja čvora odnosno bilo koje instance klase Node ili njenog derivata u čijem će se slučaju automatski izbrisati i sva njena djeca.

```
extends Node

func _ready():

    var boja = Color("red")
    var tekst = get_node("/root/putanja/do/cvora/cvorZaTekts")
    tekst.add_color_override("font_color", boja)

    queue_free(tekst) #Brisanje tekst labele iz memorije
```

Prikazani primjer koda instancira objekt klase Color. Potom u varijablu tekst sprema referencu na čvor koji je tekstualni niz na ekranu, te pomoću metode naslijeđene od klase Control mijenja boju slova tekstualnog niza. U kodu vidimo i primjer naslijeđivanja unutar skripte kao i nadjačavanja povratne funkcije čvora, što bi značilo da će se ovaj kod izvesti prilikom instanciranja čvora. Osim toga uvlačenje linija koda unutar povratne funkcije nije isključivo iz estetskih razloga već je i za određivanje djelokruga funkcije, po uzoru na Python programski jezik. Isto pravilo uvlačenja vrijedi i za ostale djelove koda, primjrice petlje ili kondicionali, gjde se očekuje definicija bloka koda odnosno određenog djelokruaga.

Na poslijetku funkcijom queue_free() demonstrativno oslobađamo memoriju od čvora za prikaz

znakovnog niza zajedno sa svom djecom čvorovima koje bi takav čvor posjedovao. Godot API sučelje a samim time i GDScript nudi dvije funkcije za brisanje čvorova iz memorije. Funkcija free() instantno briše čvor i svu njegovu djecu, bez obzira dali taj ili neki drugi čvor izvodi određeno procesiranje na čvoru za brisanje u danom okviru ili ne. Iz tog razloga free() funkcija se ne bi trebala olako koristiti, te je prvobitno namjenjena za manipulaciju memorije niže razine prilikom izmjena Godot API sučelja. Iz tog razloga pogon nudi funkciju queue_free() koja će pričekati sa izvođenjem svih procesiranja unutar određenog okvira te obrisati čvorove u redu za brisanje tek na samom kraju izvođenja okvira dok su čvorovi u besposlenom stanju. S obzirom na njihovu međusobnu asinkronost, queue_free() funkcija ne osigurava sigurno brisanje čvorova kad je u pitanju komunikacija čvorova unutar petlje fiksnog intervala i petlje prikaza.

Svi spomenuti ugrađeni tipovi podataka do sada, osim klase Object, proslijeđuju se po vrijednosti. S obzirom na njihovu veličinu i činjenicu da svi ti tipovi imaju automatsko upravljanje memorijom temeljem internog brojača referenci unutar pogona, njihovo gomilanje na stogu ne predstavlja realan problem osim u specifičnim slučajevima. Osim instance klase Object i klasa koje naslijeđuju klasu Object, ugrađeni tipovi spremnika podataka Array i Dictionary se također proslijeđuju po referenci.

Tip podataka Array Godotova je zamjena za Python liste odnosno za dinamična polja u drugim popularnim programskim jezicima. Godot polja su skupovi niza podataka koji mogu biti različitog tipa, te imaju funkcionalnost dinamičke alokacije. Indeksiraju se brojevima počevši od indeksa broj 0. Indeks može biti i negativan što označava obrnuti redoslijed. Sama polja alocirana su linearno unutar memorije radi brzine obrade i indeksiranja. Relativno velike alokacije polja, sa nekoliko desetaka tisuća elemenata u određenim slučajevima mogu radi toga prouzročiti fragmentacije memorije, odnosno pogon može alocirati neki drugi podatak između indeksiranih zapisa istog polja. Za takve slučajeve pogon nudi PoolArray tipove polja koja primaju isključivo jedan tip podataka te su atomični, no radi toga i nešto sporiji od uobičajenog polja unutar pogona.

Alternativni tip podataka koji služi kao kontenjer ugrađen unutar pogona je tip Dictionary. Dictionary je Godotova implementacija asocijativnih polja drugih popularnih programskih jezika. Ovaj tip je skup ključ-vrijednost parova. Postoje dvije dostupne sintakse za definiranje asocijativnog polja unutar GDScript programskog jezika. Standardna sintaksta ekvivalentna je Python sintaksi za rad sa riječnicima i dozvoljava da ključ bude bilo koja vrijednost definirana unutar GDScript programskog jezika, dok je druga ekvivalentna tabličniom zapisu unutar LUA programskog jezika i kao takva zahtjeva da ključ slijedi ograničenja za identifikatore unutar GDScript programskog jezika.

```
extends Node

func _ready():

    var polje = [] #inicijalizacija polja
    polje = [1, 2, 'tri']
    var polje2 = polje
    polje2[1] = 'dva'
    print(polje) #Ispisuje [1, dva, tri] (proslijeđivanje po referenci)
    assert(polje[1] == polje[-2]) #Primjer negativnog indeksiranja
    print(polje.size()) #3
    polje.append(4) #Primjer dinamičke alokacije
    print(polje.size()) #4

    #Standardan način definiranja asocijativnog polja
    var asocijativnoPolje1 = {
        null: 'nula',
        1: 'jedan',
        'dva': 'dva',
        0.3: 'tri'
    }

    asocijativnoPolje1.cetiri = 'četiri' # Alternativna sintaksa za
dohvačanje/definiranje ključa ukoliko ključ zadovoljava svojstva identifikatora
    print(asocijativnoPolje1['cetiri']) #Standardni način za dohvačanje/definiranje
ključa unutar asocijativnog polja

    #LUA način definiranja asocijativnog polja
    var asocijativnoPolje2 = {
        jedan = 'jedan',
        dva = 'dva',
        tri3 = 'tri'
    }
```

6.2. Petlje

GDScript nudi dvije opcije kad su petlje u pitanju. Naredba while označava početak djelokruga koda petlje. Ukoliko se djelokrug sastoji od jedne linije koda, onda djelokrug može biti u istoj liniji kao i naredba while, inače vrijedi pravilo uvlačenja djelokruga. Ovo pravilo vrijedi i za sve ostale ključne riječi ili naredbe koje označuju određene djelokrugove.

```
extends Node

func _ready():

    var i = 0
    var znakovniNiz = 'Znakovni niz'
    while i < znakovniNiz.size():
        print(znakovniNiz[i])

    while true: print('Infinite loop')
```

Osim while petlje, jezik nudi i for petlju za iteriranje kroz intervale ili kontenjere uz pomoć iteratora. Funckija range() vraća polje opisano argumentima funkcije po uzoru na istoimenu Python

funkciju.

```
extends Node

func _ready():

    for x in range(5):
      print(x) # 0, 1, 2, 3, 4

    for x in range(1, 5):
      print(x) # 1, 2, 3, 4

    for x in range(1, 5, 2):
      print(x) # 1, 3

    for slovo in "Riječ":
      print(slovo) # R, i, j, e, č

    for x in [2, 4, 6]:
      print(x) # 2, 4, 6

    var asocijativnoPolje = {'jedan': 1, 'dva': 2}
    for kljuc in asocijativnoPolje:
        print(kljuc, " -> ", asocijativnoPolje[kljuc]) # jedan -> 1, dva -> 2
```

Ukoliko predefinirani iteratori nisu dovoljni za korisničke potrebe, GDScript nudi mogućnost definiranja vlastitih iteratora nadjačavanjem metoda _iter_init, iter_next i _iter_get spomenute temeljne klase pogona Variant.

```
extends Node

#Primjer iteratora koji će uzeti u obzir i krajnju vrijednost prilikom iteriranja
class KorisnikovIterator:

    var pocetni
    var trenutni
    var zadnji
    var korak

    func _init(pocetak, kraj, korak = 1):

        self.pocetni = pocetak
        self.trenutni = pocetak
        self.zadnji = kraj
        self.korak = korak

    func is_done(): return (trenutni <= zadnji)

    func do_step():
        trenutni += korak
        return is_done()

    func _iter_init(arg):
        trenutni = pocetni
        return is_done()

    func _iter_next(arg): return do_step()

    func _iter_get(arg): return trenutni
```

```
    func _ready():

        var iterator = KorisnikovIterator.new(1, 5)
        for i in iterator:
            print(i) #1, 2, 3, 4, 5
```

Unutar oba tipa petlje mogu se koristiti naredbe continue i break za preskok određene iteracije, odnosno za prekid izvođenja petlje.

6.3. Kontrola toka

Kao i u ostalim programskim jezicima bazična kontrola toka programa vrši se sa tri naredbe if, else i elif.

```
extends Node

func _ready():

    randomize()

    var a = randi() % 2
    var b = randi() % 2

    if a:
        print('Izvedba kondicionala A')

    elif b:
        print('Izvedba kondicionala B')

    else:
        print('Izvedba ostalih kondicionala')
```

GDScript ne posjeduje ternarni operator no dozvoljava sintaktički slično izražavanje if else naredbama.

```
extends Node

func _ready():

    randomize()
    var nasumicniBroj = randi() % 10

    nasumicniBroj += 1 if nasumicniBroj < 5 else -1
```

GDScript nudi i naredbu match koja je ekvivalent switch naredbe iz ostalih programskih jezika sa proširenom funkcionalnošću.

Naredba match provjerava uzorak dane vrijednosti te na temelju njega izvodi određen djelokrug koda. GDScript match naredba podržava i kontinuiranu padajuću provjeru uzoraka naredbom continue koja će prekinuti izvedbu koda unutar djelokruga i krenuti na sljedeću provjeru uzorka. Postoji nekoliko različitih tipova uzoraka u kontekstu match naredbe. To mogu biti primitivne

vrijednosti poput brojeva ili znakovnih nizeva u obliku konstanti ili varijabli. Predefinirani slučaj i ključnu riječ default iz ostalih programskih jezika zamjenjuje vrijednost _ ili deklaracija nove varijable kao uzorka, koja će poprimiti vrijednost uzorka ukoliko se njen djelokrug koda krene izvoditi.

```
extends Node

func _ready():

    var x = 1

    match x:
        1:
            print('Broj 1')
            continue # Provjere uzoraka se neće prekinuti

        2, 3, 4:
            print('Broj 2 3 ili 4')

        "pet":
            print('Broj pet')

        _:
            print('Uzorak koji odgovara svim uzorcima')
            continue

        var noviUzorak:
            print('Uzorak koji odgovara svim uzorcima te sprema vrijednost uzorka u
varijablu: ', noviUzorak )
```

Osim navedenih postoje još i uzorci polja i uzorci asocijativnih polja. U svom uobičajenom obliku dani uzorak mora upotpunosti odgovarati definiranom polju ili asocijativnom polju po veličini i po ključevima. Ukoliko se kao zadnji element uzorka unutar polja odnosno asocijativnog polja da vrijednost .., uzorak može biti podskup dane vrijednosti match naredbe.

```
extends Node

func _ready():

    var polje = [1, 2, 3, 4]
    match polje:
        [1, 2, 3, 4, 5]: print('Uzorak se ne poklapa')

        [1, 2, 3]: print('Uzorak se ne poklapa')

        [1, 2, 3, 4]:
            print('Uzorak se u potpunosti poklapa')
            continue

         [1, 2, 3, ..]: print('Uzorak je podskup')

    var asocijativnoPolje = {"jedan": 1, "dva": 2, "tri": 3}
    match asocijativnoPolje:
        {"jedan": 1}: print('Uzorak se ne poklapa')

        {"jedan": 1, "dva": 2, "tri": 3}:
            print('Uzorak se u potpunosti poklapa')
            continue
```

```
        {"jedan", "dva", "tri"}:
            print('Uzorak se poklapa po ključevima')
            continue

        {"jedan": 1, ..}: print('Uzorak je podskup')
```

Ukoliko uzorku asocijativnog polja damo samo vrijednosti ključeva, vrijednosti spremljene unutar ključeva neće se gledati.

6.4. Funkcije

Funkcije unutar GDScripta uvijek pripadaju određenoj klasi. Hijerahija redoslijeda dijelokruga varijabli funkcije kreće od lokalnog djelokruga funkcije, zatim prelazi na atribute klase i na poslijetku provjerava globalnu razinu. Za specifikaciju atributa klase koji je u koliziji sa lokalnom ili globalnom razinom postoji ključna riječ self koja sadrži referencu na trenutni objekt. Sve funkcije unutar GDScripta vraćaju određenu vrijednost i u koliko korisnik nije specificirao drugačije naredbom return, ta vrijednost je tipa null. Za deklaraciju funkcije koristi se naredba func.

```
extends Node

var x = 'X'

func _ready():

    var y = 'Y'
    var xy = funkcijaSpoji(y)
    print(xy) #Ispisati će se null vrijednost

func funkcijaSpoji(y):
    print(x + y)
```

Radi performansi okoline funkcije nisu prvorazredni objekti unutar jezika te se njihov djelokrug nemože direktno spremati unutar varijabli i time proslijeđivati ili vraćati iz drugih funkcija. Te se funkcijonalnosti mogu postići pomoću pomoćnih funkcija call() i funcref().

Funkcije mogu biti definirane i kao statične funkcije naredbom static ispred njene definicije. Time funkcija nije vezana za klasu, te se može pozivati u drugim instancama bez da se instancira klasa u kojoj je definirana, u koliko se prije učita skripta kao resurs u scenu stabla.

6.5. Klase

Svako tijelo skripte unutar GDScripta je predefinirano kao bezimena klasa koja se može referencirati isključivo kao resurs odnosno datoteka. Svaka klasa može sadržavati isključivo vlastite atribute i/ili funkcije. Statične funkcije su dozvoljene no ne i statične varijable. Kao primjer

minimalne klase u GDScript jeziku može se uzeti prazna skripta. Takva skripta automatski će naslijeđivati Reference tip pogona no iz tog razloga se nemože pridodati čvoru već evenutalno učitati kao resurs unutar već postojećeg čvora. Da bi se skripta mogla pridodati čvoru i tako postati opis klase zadužene za taj čvor, skripta mora minimalno naslijediti od interne klase Object naredbom extends.

Klase mogu naslijeđivati globalne klase, ostale klase spremljene kao datoteke ili unutarnje klase drugih klasa. Višestruko naslijeđivanje nije direktno moguće, no može se postići slaganjem željene hijerarhije naslijeđivanja klasa.

```
#Ime datoteke: VanjskaKlasa.gd
extends Node

class UnutarnjaKlasa extends Node:

    func funkcija():
        print('Funkcija roditelja)
extends "VanjskaKlasa.gd".UnutarnjaKlasa

func _ready():

    print(self is Node) #true
    .funkcija() #Funkcija roditelja
    funkcija() #Funkcija dijeteta

    func funkcija():
        print('Funkcija dijeteta')
```

Naredba is provjerava dali određeni objekt je ili naslijeđuje od određenog tipa klase. Znakom . ispred imena funkcije poziva se funkcija roditelja.

Konstruktor klase definiran je automatski. Ukoliko korisnik želi definirati vlastite konstruktore mora nadjačati funkciju _init(). Naslijeđivanjem klase poziva se automatski i konstruktor naslijeđene klase te s toga potreba za pozivanjem ._init() konstruktora je rijetka. No ukoliko naslijeđena klasa nadjačava funkciju _init() sa određenim brojem ne predefiniranih argumenata tada i klasa naslijednik mora definirati vlastiti konstruktor koji će u svom najednostavnijem obliku proslijeđivati željene vrijednosti argumenata naslijeđenoj klasi posebnom sintaksom.

```
#Ime datoteke: VanjskaKlasa.gd
extends Node

func _init(atribut):
    print(atribut)
```

```
extends "VanjskaKlasa.gd"

func _init().('Vrijednost argumenta'):
    pass #Naredba koja označava prazan djelokrug za interpreter
```

Atributi klasa podržavaju i funkcionalnost predefiniranih funkcija postavljača i dobavljača.

```
extends Node

var variabla setget postaviVariablu, dohvatiVariablu

func postaviVarijablu(vrijednost):
    variabla = vrijednost

func dohvatiVarijablu():
    return variabla
```

Na taj će način bilo koje izvan klasno dohvaćanje ili mjenjanje vrijednosti varijable biti izvedeno kroz definirane funkcije, pa tako i mjenjanje njihovih vrijednosti kroz uređivač pogona. Bilo koje od dvije funkcije mogu biti izostavljene.

```
var variabla setget postaviVariablu
#ili
var variabla setget , dohvatiVariablu
```

6.6. GDScript integracija sa okolinom pogona

Osim standardnih funkcionalnosti klasičnog dinamičkog objektno orijentiranog programskog jezika koje su u potpunosti iskoristive i funkcionalne u arbitralnoj samostalnoj okolini, GDScript kao dio dubinske integracije sa Godot pogonom, nudi nekoliko funkcionalnosti koje nisu iskoristive izvan okoline samog pogona.

6.6.1. Učitavanje resursa

Godot pogon nudi dva za učitavanje resursa unutar memorije, pa samim time i GDScript također. Funkcija load() dinamički učitava željene resurse prilikom izvođenja igre. Funkcija preload() to radi za vrijeme kompajliranja aplikacije. Iako GDScript koristi interpreter za izvođenje, ostatak aplikacije izrađene u Godot pogonu se kompajlira prije izvođenja, te se svaki dio aplikacije pretvara u svoj binarni format. Prednost load() funkcije je ta što korisnik može učitati određene resurse, točno kad mu trebaju i time uštedjeti memoriju. No učitavanje većih količina resursa prilikom izvođenja aplikacije može dovesti do zastoja u izvedbi aplikacije, jer korištenje load() metode blokira trenutnu nit izvođenja do završetka učitavanja. Pomoću preload() funkcije, resursi će biti učitani prije pokretanja scene te će se time izbjeći zastoj u izvođenju aplikacije, no produžiti će se vrijeme učitavanja same scene. Osim toga postoji i ResourceInteractiveLoader klasa čija je namjena pozadinsko učitavanje resursa prilikom izvođenja aplikacije na jednoj ili više izvedbenih niti.

```
extends Node

var scenaDijeteResurs = preload('res://scene/Scena.tscn')
var teksturaResurs = load('res://putanja/do/direktorija/tekstura.png')

func _ready():
```

```
var scenaDijeteInstanca = scenaDijeteResurs.instance()
add_child(scenaDijeteInstanca)

scenaDijeteInstanca.texture = teksturaResurs
```

U primjeru učitana scena dijete učitava se prilikom učitavanja scene koja sadrži čvor koji učitava scenu dijete, dok se tekstura učitava prilikom inicijalizacije skripte koja pripada čvoru koji učitava teksturu.

Za razliku od ostalih resursa, učitavanje scene rezultira resursom tipa PackedScene iz razloga što scene kao resursi imaju specifičniju primjenu od ostalih resursa. Da bi se scena mogla dodati u stablo scene, treba se prije odpakirati odnosno instancirati u scenu funkcijom instance() koja pripada klasi PackedScene. Jednom instancirana scena može se dodati kao djete bilo kojem čvor funkcijom add_child() klase Node.

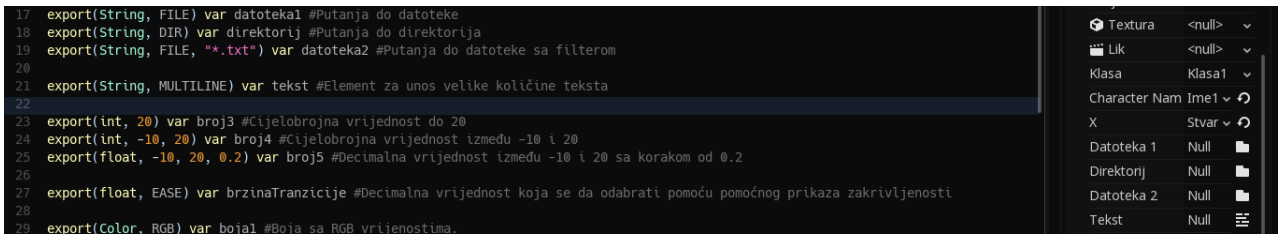
U primjeru vidimo da je ostale tipove resursa dovoljno pridodati određenom atributu čvora kako bi se ti resursi iskoristili.

6.6.2. Izvoz atributa klase

Atributi klase mogu se izvoditi naredbom export. Vrijednost izvedene varijable spremiti će se zajedno sa čvorom koji sadrži tu varijablu. Osim toga varijabla će se pojaviti i u uređivaču pogona kao promjenjivi atribut čvora kojem pripada. Izvedena varijabla mora biti incijalizirana na određenu vrijednost ili sadržavati nagovjest tipa. Tip varijable može podrazumijevati i određen tip resursa. Ukoliko se uz tip navede još dodatnih vrijednosti određenog tipa, te će vrijednosti biti prikazi za prave enumerirane vrijednosti unutar pogona. Osim toga export naredba prima i mnogo drugih argumenata koji služe za rad sa intervalima, datotekama, putanjama, bojama i čvorovima. Naredbom se može izvesti više binarnih vrijednosti u obliku jedne cijelobrojne vrijednosti. Enumerirane konstante za određene binarne vrijednosti trebale bi se razlikovati po exponencijalnim vrijednostima broja 2, pošto se njihove vrijednosti zbrajaju u jednu vrijednost po kojoj se može diferencirati stanje svih binarnih vrijednosti.

Osim navedenog export naredba radi i sa poljima. No kako se polja u tom kontekstu gotovo isključivo koriste kao granične vrijednosti različitih geometrijskih oblika, reprezentiranih u obliku matričnih zapisa, vrijednosti izvezenih polja djele se među svim instancama određenog čvora. Izvedena polja nemogu se incijalizirati varijablama već samo konstantnim vrijednostima.

```
extends Node

export var broj1 = 5
export(int) var broj2

# Export can also take a resource type to use as a hint.
export(Texture) var textura
export(PackedScene) var lik

#Opcije numeriranja:
export(int, "Klasa1", "Klasa2", "Klasa3") var klasa #0, 1, 2
export(String, "Ime1", "Ime2", "Ime3") var character_name #Dani znakovni nizovi

enum NamedEnum {STVAR_1, STVAR_2, STVAR_3 = -1} #0, 1, -1
export (NamedEnum) var x #Prikazana vrijednost će se formatirati

export(String, FILE) var datoteka1 #Putanja do datoteke
export(String, DIR) var direktorij #Putanja do direktorija
export(String, FILE, "*.txt") var datoteka2 #Putanja do datoteke sa filterom

export(String, MULTILINE) var tekst #Element za unos velike količine teksta

export(int, 20) var broj3 #Cijelobrojna vrijednost do 20
export(int, -10, 20) var broj4 #Cijelobrojna vrijednost između -10 i 20
export(float, -10, 20, 0.2) var broj5 #Decimalna vrijednost između -10 i 20 sa korakom od
0.2

export(float, EASE) var brzinaTranzicije #Decimalna vrijednost koja se da odabrati pomoću
pomoćnog prikaza zakrivljenosti

export(Color, RGB) var boja1 #Boja sa RGB vrijenostima.
export(Color, RGBA) var boja2 # Boja sa RGBA vrijenostima.

export(NodePath) var cvor #Cvor

#Zapis četiri binarnih vrijednosti u jednu cijelobrojnu
const ELEMENT_1 = 1
const ELEMENT_2 = 2
const ELEMENT_3 = 4
const ELEMENT_4 = 8
export(int, FLAGS, 'element_1', 'element_2', 'element_3', 'element_4') var elementi = 0

#Primejri vektora
export(Array, Array, float) var polje2D = [[1, 2], [3, 4]]
export var vector3D = PoolVector3Array()
```



Slika 9: Primjer izvoza varijabli u uređivač

6.6.3. Alat

Sam uređivač pogona je također aplikacija napravljena u Godot pogonu. Iz tog razloga skripte se u cijelosti mogu izvoditi i u samoj okolini uređivača. Skriptu koju korisnik želi namjeniti za okolinu

uređivača označuje naredba tool na samom vrhu skripte. Po predefiniranim postavkama skripte mogu isključivo izvoziti varijable vlastitih klasa u uređivač pogona, dok sa tool naredbom, uređivač kroz skriptu ima pristup cijeloukupnom GDScript jeziku. Česta uporaba ovoga je za pisanje dodatnih modula jednostavne namjene uređivača.

```
Tool

extends ResourcePreloader
export(String, FILE, GLOBAL, "*.png") var slika
```

6.6.4. Signali

GDScript ima direktnu integraciju sa signalima Godot API sučelja. Pomoću signala mogu se definirati vlastite povratne funkcije na određene događaje unutar određenih instanci. Signali se u GDScriptu definiraju sa naredbom signal, te se potom moraju povezati sa određenom funkcijom određene instance ili preko uređivača ili direktno u kodu.

```
extends Node

signal signalBezArgumenata
signal signalSaArgumentom1(argument)
signal signalSaArgumentom2(argument)


func _ready():

    connect('signalBezArgumenata', self, 'povratnaFunckija1')
    connect("signalSaArgumentom1", self, "povratnaFunckija2")
    connect("signalSaArgumentom2", self, "povratnaFunckija3", [get_parent()])

    emit_signal('signalBezArgumenata') #Poziv signala bez argumenata.
    emit_signal('signalSaArgumentom1', self) #Poziv signala od strane cvora: cvor
    emit_signal('signalSaArgumentom2') #Poziv signala od strane cvora: root

func povratnaFunckija1():
    print('Poziv signala bez argumenata.')

func povratnaFunckija2(argument):
    print('Poziv signala od strane cvora imenom: ', argument.get_name())

func povratnaFunckija3(argument):
    print('Poziv signala od strane cvora imenom: ', argument.get_name())
```

Iz priloženog koda se vidi da signali mogu prenositi i argumente povratnim funkcijama. No za razliku od funkcija signali rade razliku odakle se vrijednost argumenta proslijeđuje. Signal može proslijediti argument povratnoj funkciji u funkciji za spajanje signala connect() ili u funkciji za emitiranje signala emit_signal(). Obje funkcije naslijeđuju se od Object klase.

6.6.5. Korutine

Po uzoru na Python, GDScript implementira korutine ugrađenom funkcijom yield.

```
extends Node

func korutina():
    print("Početak funkcije")
    yield() #Prekid
    print("Nastavak funkcije")
    print(yield()) #Prekid
    print("Kraj funkcije")

func _ready():
    var korutina = korutina()
    print("Prekid funkcije")
    print(korutina) #Stanje funkcije
    korutina = korutina.resume()
    print("Prekid funkcije")
    korutina = korutina.resume('Proslijeđeni argument korutini')
```

Funkcija resume() nastavlja sa izvođenjem korutine ukoliko se pozove na instancu klase koji reprezentira valjano stanje funkcije. Funkciji resume() moguće je proslijeđivati argumente te će oni zamjeniti mjesto yield() funkcije unutar korutine.

U praksi se signali i korutine često koriste u paru kako bi se određeni signal sinkrono propagirao kroz sustav.

6.6.6. Datotečni sustav

Kad su u pitanju datotečni sustavi, Godot a time i GDScript podržava isključivo UNIX separatore putanje „/“, koji se potom interno konvertiraju u pravu putanje ovisno o platformi na kojoj se aplikacija izvodi. Korijenski direktorij godot projekta nalazi se u direktoriju u kojem se nalazi i njegova project.godot datoteka koja sadrži osnove postavke projekta. Apsolutna putanja korijenskog direktorija određenog projekta na određenoj platformi nalazi se spremljena pod „res://“ vrijednosti znakovnog niza.

Prilikom korištenja aplikacija na modernim operativnim sustavima, čest je slučaj da za direktorij u kojem je pohranjena aplikacija, trenutni korisnik ili pokrenuta aplikacija ima isključivo mogučnost čitanja podataka. S toga pogon nudi i „user://“ vrijednost znakovnog niza kao alias apsolutne putanje do predefiniranog direktorija određenog operativnog sustava koji je predviđen da daje korisniku ili aplikaciji dozvolu za pisanje podataka.

```
extends Node

func _ready():
    load('res://scene/Scena.tscn') #Učitavanje resursa pomoću korijenskog direktorija
projekta
```

```
    var datoteka = File.new()
    if datoteka.open('user://saved_game.sav', File.WRITE) != 0: #Učitavanje datoteke
pomoću predefiniranog direktorija zapisa
        print('Greška prilikom otvaranja datoteke')

    else:
        datoteka.store_line('Zapis')
        datoteka.close()
```

Vrijednost za „user://“ znakovni niz može se prilagoditi vlastitim potrebama.

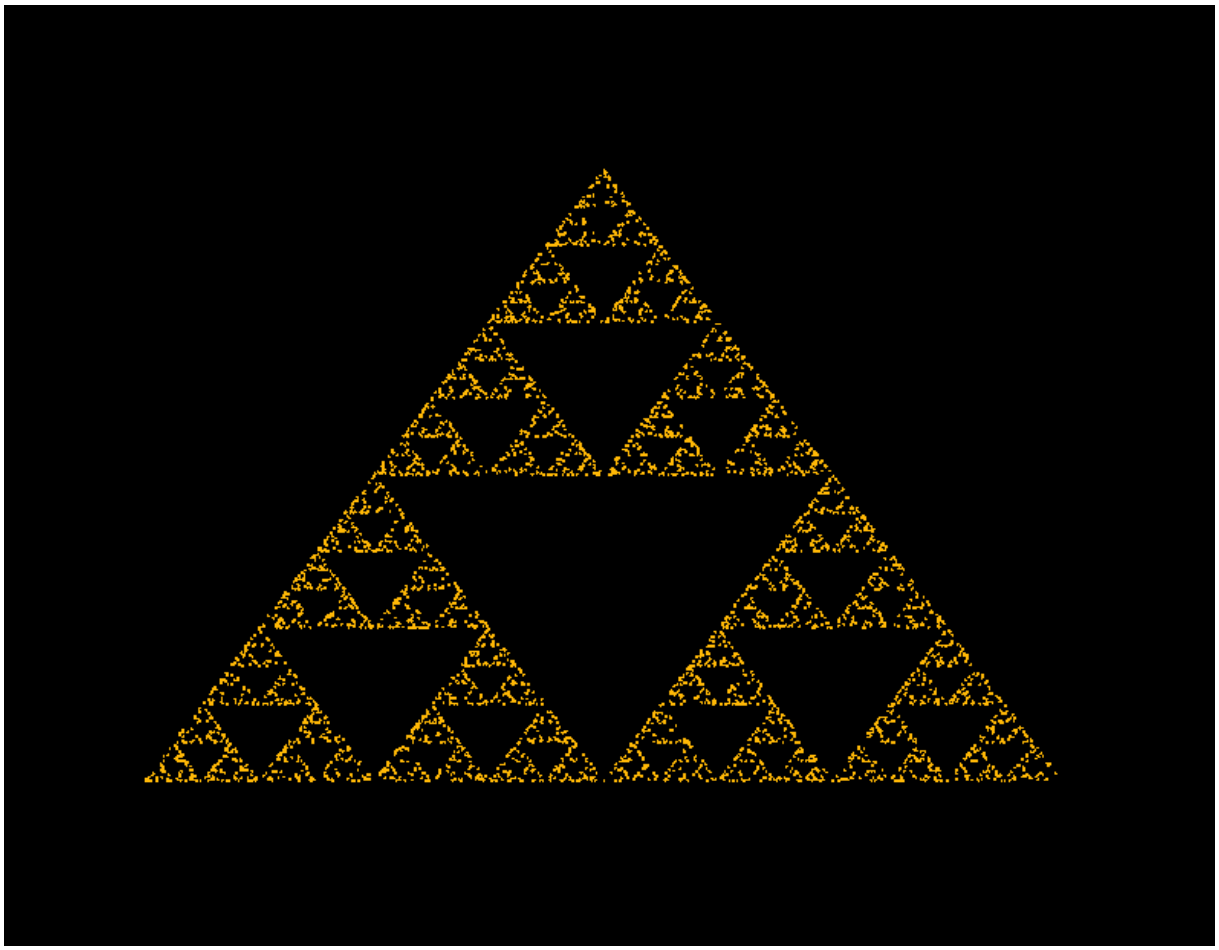
7. Prikazivanje sadržaja

Za predefinirano prikazivanje sadržaja pogon koristi korijenski čvor tipa Viewport naziva „root“. Za razliku od ostalih čvorova za određeneu scenu, u uređivaču je taj čvor skriven te su njegove postavke predefinirane postavkama projekta, ili se mogu izmjenivati pomoću koda. Taj čvor će uvijek biti korijenski odnosno prvi čvor određene scene, te će svi ostali čvorovi unutar scene biti njegova djeca, direktno ili indirektno. Scena može sadržavati više Viewport čvorova te samim time i više zasebnih prikaza sadržaja na jednom ekranu.

Osim trodimenzionalnih modela, dvodimenzionalnih slika i elemenata korisničkog sučelja koje pogon prikazuje pomoću određenih čvorova, korisnik može programatski crtati željeni sadržaj po potrebi manipulirajući same piksele prikaza ili pomoću funkcionalnosti čvora tipa CanvasItem i ostalih čvorova koji naslijeđuju spomenuti čvor. U drugom slučaju željeno crtanje se može izvesti i pomoću nadjačavanja povratne funkcije _draw() CanvasItem čvora, te će na taj način željeno iscrtavanje postati dio funkcionalnosti samog iscrtavanja čvora. Primjer moguće izvedbe Sierpinskog fraktala dan je sljedećim kodom:

```
extends Node2D

var velicinaKruga = 1
var boja = Color('#ffb405')
var tocka1 = Vector2(500,200)
var tocka2 = Vector2(200,600)
var tocka3 = Vector2(800,600)
var sljedecaTocka = Vector2(0,0)
var trenutnaTocka = Vector2(500,200)
var korakUdaljenosti = 0.5
var brojTocaka = 5000

func _ready():
    draw_circle(tocka1, velicinaKruga, boja)
    draw_circle(tocka2, velicinaKruga, boja)
    draw_circle(tocka3, velicinaKruga, boja)

func _process(delta):
    update()

func _draw():
    for n in range (brojTocaka):
        var pickPoint = randi() % 3 + 1
        if pickPoint == 1:
            sljedecaTocka = tocka1
        elif pickPoint == 2:
            sljedecaTocka = tocka2
        else:
            sljedecaTocka = tocka3
        trenutnaTocka = trenutnaTocka + (sljedecaTocka - trenutnaTocka) *
korakUdaljenosti
        draw_circle(trenutnaTocka, velicinaKruga, boja)
```

Nadjačavanjem povratne funkcija _process(delta) svakog čvora koji naslijeđuje klasu Node, opisujemo njegovu funkcionalnost u svakom okviru prikaza po sekundi aplikacije. Funkcija update() poziva ažuriranje čvora tipa CanvasItem a samim time i nadjačanu implementaciju povratne funkcije _draw(), te kako se nalazi u tijelu nadjačane povratne funkcije _process(delta) od istog čvora, ažuriranje se dešava na svakom novom okviru prikaza.



Slika 10: Nasumični Sierpinski fraktal generiran danim kodom u određenom okviru prikaza

Parametar delta je tipa float te označava diferenciju vremena u sekudama od prethodnog poziva funkcije za iscrtavanje unutar glavne petlje aplikacije. Ta će se funkcija pozivati što je brže moguće tokom izvedbe aplikacije, što znači da će delta varirati svakim pozivom te funkcije jer sama brzina izvedbe te funkcije uvjetovana je i količinom procesiranja koje funkcija mora obaviti za dani okvir. Kao i u ostalim pogonima, samim tim svojstvom delta vrijednosti možemo osigurati okvirno jednak diferencijal izvedbe procesa koji su ovisni o brzini prikaza okvira na određenom ekranu, kao što su na primjer kretnje neprijatelja. Jednaka pravila za prikazivanje vrijede i kad su u pitanju čvorovi koji prikazuju kreativne resurse poput trodimenzionalnih modela ili dvodimenzionalnih slika i sličnog sadržaja, iako su često u praksi sami procesi njihove obrade prikaza mnogo kompleksniji.

8. Fizika pogona i osnove procesiranja fizike

Godot pogon za izvršavanje izračuna simulacija fizike unutar svojih aplikacija interno koristi dva različita pogona fizike. Biblioteka Bullet je veoma popularan projekt otvorenog koda koji se trudi biti kompletna solucija za simuliranje fizike u stvarnom vremenu. Godot pogon koristi Bullet biblioteku za izračune fizike u trodimenzionalnim prostorima. Za dvodimenzionalne prostore Godot pogon koristi vlastiti pogon za fiziku prilagođenim specifičnim potrebama pogona. No rad sa oba rješenja kroz Godot C++ API sučelje, a samim time i kroz njegove skriptne jezike je veoma sličan, a jedine krucijalne razlike vezane su za tipove podataka u kontekstu dimenzija prostora i njihovih funkcionalnosti.

8.1. Povratna funkcija _physics_process(delta)

Kao i kod ostalih popularnih pogona za izradu video igara, za razliku od logike aplikacije i njenog prikaza, izračuni vezani za fiziku računaju se u fiksiranim okvirima po sekundi. U Godot pogonu po predefiniranim postavkama svaki okvir izračuna fizike pozvati će se šezdeset puta u sekundi, ukoliko je to moguće. Takvi fiksni intervali za izračun fizike potrebni su kako bi se osigurala željena konzistentnost između različitih koraka izračuna.

```
extends Node

func _process(delta):
    print("_process delta: ", delta)

func _physics_process(delta):
    print("_physics_process delta: ", delta)

#_physics_process delta: 0.016667
#_physics_process delta: 0.016667
#_physics_process delta: 0.016667
#_process delta: 0.081541
#_physics_process delta: 0.016667
#_process delta: 0.008738
#_physics_process delta: 0.016667
#_process delta: 0.016677
```

Vidiljivo je po vrijednosti varijable delta da povratna funkcija _physics_process() teži fiksnim intervalima izvođenja dok povratna funkcija _process() varira u vremenu prethodnog poziva. Isto tako povratna funkcija za procesiranje fizike neće se izvoditi više puta nego li je to propisano vrijednošću njenog intervala izvedbe, dok u koliko korisnik samostalno ne limitira broj poziva na izvedbu povratne funkcije za okvire prikaza po sekudni, one će se izvršavati što je brže moguće.

8.2. Čvorovi za fiziku

Godot pogon nudi četiri vrste čvorova koji direktno interagiraju sa poslužiteljem fizike. To su Area, StaticBody, RigidBody, KinematicBody i njihove odgovarajuće verzije za dvodimenzionalne okoline, Area2D, StaticBody2D, RigidBody2D, KinematicBody2D.

8.2.1. Čvor za prostor

Area i Area2D čvorovi služe za reprezentaciju određenog prostora sa specifičnim svojstvima unutar pogona fizike. Samim time ti čvorovi mogu nadjačavati predefinirana fizička svojstva za određeni prostor, no osim toga mogu i detektirati ulaz i izlaz određenog objekta u određenom prostoru te provjeravati međusobna preklapanja sa ostalim prostorima.

8.2.2. Čvor za statično tijelo

Čest scenarij u razvoju igara je potreba određenog objekta da bude dio sustava za koliziju no ne i pod utjecajem ostalih atributa pogona fizike. Čvor StaticBody, odnosno StaticBody2D služi upravo za takve situacije, primjerice implementacija zidova unutar prostora. Osim što će detektirati koliziju sa ostalim čvorovima koji su dio sustava fizike, te po predefiniranim svojstvima spriječiti ostale čvorove da se kreću po definiranom prostoru kolizije StaticBody čvora, čvor sadrži i nekoliko drugih atributa koji mogu utjecati na ponašanje ostalih čvorova na koje utječe pogon fizike, primjerice kutni momentum koji neće rotirati sam čvor no utjecati će na koliziju sa ostalim čvorovima kao da se statični čvor rotira.

8.2.3. Čvor za kruto tijelo

Čvorovi RigidBody i RigidBody2D služe kao instance koje su u potpunosti pod utjecajem pogona fizike. Dolaze u paketu sa mnoštvom atributa koji direktno utječu na ponašanje instance unutar pogona fizike, poput mase, frikcije ili elastičnosti čvora.

Atributi poput pozicije ili brzine gibanja čvora za kruto tijelo unutar Godot pogona ne bi se trebali direktno mijenjati, jer time narušava njihovu svrhu a i često su poslijedice tih postupaka nepredvidive. Ti atributi trebali bi biti mjenjani od strane pogona za fziku s obzirom na cijelokupne uvjete okoline, sila koje se primjenjuju na čvorove i sama svojstva čvorova. U koliko iz nekog razloga korisnik želi veću kontrolu nad tim atributima čvorova, ti čvorovi imaju posebnu povratnu funkciju za takve situacije _integrate_forces() preko koje se mogu sigurno mjenjati atributi čvora na način da čvor bude sinkroniziran sa njegovim stanjem unutar pogona fizike.

Sa implementacijom krutih tijela u svakom pogonu za fiziku pa tako i u pogonima za izradu video igara dobiva se zavidna količina realiziranog ponašanja određenog elemenata, bez da korisnik realizira imalo koda sa svoje strane. No kako se većina pogona za fiziku trudi biti što dinamičnija i pružiti što veću međusobnu interakciju instanci unutar sustava, ta su ponašanja često nepredvidivo skupa po resurse te se uz nepažljivo prekomjerno korištenje takvih instanci mogu bez problema zakrčiti resursi većine namjenjenih mašina za pokretanje aplikacije. Kako se potrebni resursi nebi uzalud trošili na tijela koja nisu u interakciji sa pogonom fizike neko određeno vrijeme, Godot pogon ugrađuje uspavana stanja za svoja kruta tijela koje se aktivira nakon određenog vremena ne aktivnosti tijela sa pogonom fizike. U tom stanju povratna funkcija _integrate_forces() neće biti zvana svaki u svakom fiksnom okviru igre te samim time pogon fizike neće vršiti izračune za to određeno tijelo. Tijelo izlazi iz uspavanog stanja ukoliko se primjeni određena sila na tijelo ili dođe do nove kolizije sa nekom drugom instancom sustava fizike. To se svojstvo krutog tijela može isključiti sa postavljanjem vrijednosti can_sleep atributa RigidBody ili RigidBody2D čvora na vrijednost false.

8.2.4. Čvor za kinematična tijela

Iako većina modernih pogona danas se odlučuje za soluciju kinematičkih tijela kad su u pitanju tijela kojima je svrha da budu upravljani preko koda, Godot pogon nudi i alternativnu, danas manje popularnu soluciju dinamičnog tijela uz vlastitu izvedbu kinematičnog tijela. Izvedba dinamičnog tijela moguća je uporabom jednog od četiri predefinirana stanja RigidBody i RigidBody2D čvora. Iako ta solucija ima svoje prednosti, često je odbojna jer zahtjeva veoma dobro poznavanje internog rada pogona za fiziku te ne nudi direktnu kontrolu nad tijelom u potpunosti. Predefinirani način i model za izvedbu direktnog upravljanja tijela je kroz KinematicBody čvor odnosno KinematicBody2D čvor.

U kontekstu pogona za fiziku, kinematično tijelo je tijelo koje registrira kolizije s ostalim tjelima no na koje ne utječu ostale sile pogona fizike. Na taj način korisnik ima tijelo s kojim može sam izgraditi željena ponašanja ovisno o okolini i svojstvima unutar pogona fizike. Kinematička tijela su najčešće korištena tijela kada su u pitanju pokretni likovi ili elementi igre.

8.3. Kolizije i kolizijski oblici

Kako bi bilo koji od spomenutih čvorova mogao detektirati određenu koliziju a s time i reagirati na istu, potrebno je da posjeduje barem jedan kolizijski oblik, u suprotnom detekcija kolizije neće se

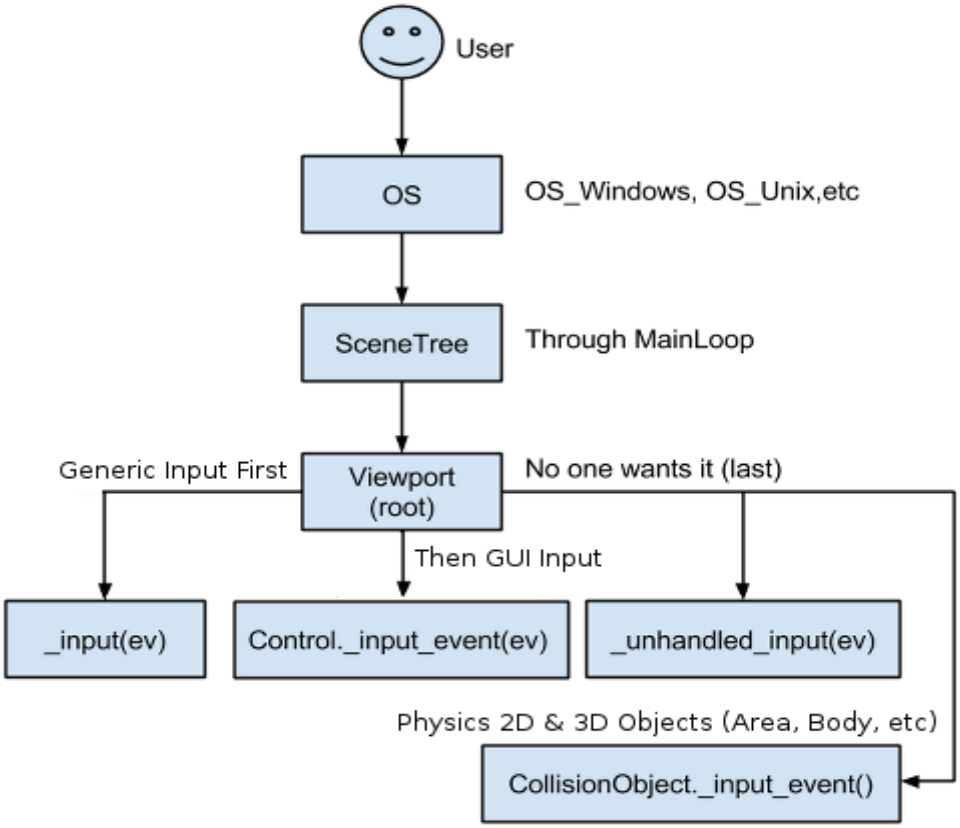
izvršiti. Kolizijski oblici su razni geometrijski oblici koji se mogu definirati vrijednostima polja, ostalim ugrađenim matematičkim tipovima unutar pogona ili crtati u uređivaču pogona. Kolizijski oblik reprezentira prostor kolizijskog djelovanja određenog čvora koji je dio sustava fizike. Taj prostor nemora nužno oslikavati sam oblik prikaza čvora, iako je to najčešći slučaj. Svaki čvor može imati više kolizijskih oblika te samim time više definiranih ponašanja ovisno o specifičnom kolizijskom obliku koji je detektirao koliziju.

Kolizijski oblici omogućuju čvorovima da se nalaze i na takozvanim kolizijskim slojevima te da posjeduju kolizijske maske. Pomoću njih može se izgraditi kompleksan sustav logike izvedbe i reagiranja na kolizije. Primjer jednostavnog krutog tijela i njegovog kolizijskog oblika dan je sljedećim sadržajem pripadajuće .tscn datoteke za scenu koja sadrži spomenuto tijelo.

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

[ext_resource path="res://test.gd" type="Script" id=1]

[node name="Cvor" type="RigidBody2D" index="0"]

position = Vector2( 57.0153, 54.5718 )
input_pickable = false
collision_layer = 1
collision_mask = 1
mode = 0
mass = 1.0
friction = 1.0
bounce = 0.0
gravity_scale = 1.0
custom_integrator = false
continuous_cd = 0
contacts_reported = 0
contact_monitor = false
sleeping = false
can_sleep = true
linear_velocity = Vector2( 0, 0 )
linear_damp = -1.0
angular_velocity = 0.0
angular_damp = -1.0
script = ExtResource( 1 )

[node name="CollisionPolygon2D" type="CollisionPolygon2D" parent="." index="0"]

position = Vector2( 304.625, 113.216 )
build_mode = 0
polygon = PoolVector2Array( 13.2063, 14.5536, 86.5117, 25.1422, -43.809, -40.8327, -
149.695, 126.955, -42.9945, 32.4727, -68.2441, 132.657, -21.8174, 81.343, 9.94832,
109.036, 30.3109, 88.6736, 35.198, 71.569, 28.6819, 48.7628, 34.3835, 40.6178, 9.94832,
36.5453, -10.4143, 37.3598, -17.7448, 28.4002, 0.174271, 12.9246 )
```

9. Sustav ulaznih signala

Svaki ulazni signal u kontekstu Godot pogona, bilo da se radi o ulaznom signalu krajnjeg korisnika ili signalu emuliranog kodom mora proći kroz OS instancirani objekt za specifičnu platformu. Ta će klasa proslijediti ulazni signal instanci SceneTree klase koja će na poslijetku dodjeliti signal korijenskom prikazu stabla scene.



Slika 11: Tok ulaznog signala unutar pogona Godot

Korjesnki prikaz je taj koji propagira ulazni signal ostalim čvorovima u predefiniranom redoslijedu. Redoslijed obrade signala kreće najprije sa čvorovima koji nadjačavaju povratnu funkciju _input(InputEvent event) prvi obraditi signal. U koliko korisnik želi da propagiranje signala stane nakon određene obrade, SceneTree klasa nudi funkciju set_input_as_handled() koju korisnik može pozvati unutar djelokruga koji obrađuje određeni signal. U suprotnom će se signal probati propagirati na čvorove koji naslijeđuju Control čvor (čvorovi namjenjeni za izgradnju korisničkih sučelja) ukoliko je signal u samom kontekstu interakcije tih čvorova, te korisnik može postići dodatnu obradu tih signala nadjačavanjem povratne funkciju _gui_input(InputEvent event). Jednom obrađeni signal od strane Control čvora u kontekstu vlastite interakcije neće se propagirati dalje.

Signal se naposlijetku obrađuje u svim čvorovima koji nadjačavaju _unhandled_input(InputEvent event) povratnu funkciju. Jednom obrađeni signal unutar ove povratne funkcije se ne propagira dalje. Ukoliko do tada niti jedan čvor nije obradio signal, signal se šalje sljedećem prikazu unutar stabla scene na daljne propagiranje ili se ignorira.

U slučaju da je signal klik miša a trenutni prikaz posjeduje dodjeljenu kameru u obliku čvora Camera, prije zadnjeg opisanog koraka zraka detekcije kolizije biti će izračunata između ishodišta kamere i točke na kojoj se desio klik miša. U koliko zraka pogodi objekt, pozvati će se povratna funkcija _input_event(Object camera, InputEvent event, Vector3 click_position, Vector3 click_normal, int shape_idx) na pogođenom objektu.

9.1. Događaj ulaza

Svi signali ulaza naslijeđuju klasu InputEvent. Kalsa InputEvent sadrži isključivo pogonov identifikacijski broj uređaja signala ulaza. Klase koje naslijeđuju InputEvent sadrže ostale informacije poput kojeg je tipa događaj. To su InputEventKey, InputEventMouseButton, InputEventMotion za miš i tipkovnicu, InputEventJoypadMotion i InputEventJoypadMotion za upravljače igara te InputEventScreenTouch i InputEventScreenDrag za uređaje sa ekranima osjetljivim na dodir. Postoji i dodatna klasa InputEventAction koja služi za definirane korisničke akcije.

```
extends Node2D


func _input(event):
    if event is InputEventKey:
        if event.pressed and event.scancode == KEY_SPACE:
            print(event)
```

Svaki tip ulaznog događaja ima i vlastita stanja u kojem može biti. Primjerice InputEventKey može biti u pritisnutom, držanom ili odpuštenom stanju koje se regulira pripadnim varijablama pressed i echo.

Iz priloženog lako se da uočiti da se upravljanje ulazima vrlo brzo može zakomplicirati te da kod vezan uz upravljanje ulaza može postati nepregledan. Osim toga na ovaj način krajnji korisnik nema kontrolu nad samim mapiranjem kontrola. S toga Godot pogon nudi klasu InputEventAction koja predstavlja abstrakciju akcije pokrenute određenim ulaznim signalom, te klasu jedinca InputMap za učitavanje eksternih vrijednosti mapiranja akcija na jedan ili više različitih ulaznih signala. Na taj način pogon omogućuje da se bilo koja određena akcija mapira na ulazne signale sa više uređaja

odjednom, te se odvaja proces mapiranja ulaza od procesa obrade određenog tipa ulaza. Predefinirano stanje InputMap instance može se uređivati direktno iz uređivača pogona, te će se ono očitati prilikom pokretanja aplikacije iz project.godot datoteke.

```
extends Node2D


func _process(delta):
    if Input.is_action_pressed("Hodaj"):
        print('Hodam')
```

Na taj se način ulazni signali mogu obrađivati i unutar _process() i _physics_process() povratnih funkicja.

10. Izvoz projekta

Godot pogon u svom kopresiranom obliku pripremljenom za preuzimanje sa službene stranice nosi svega nešto ispod dvadeset megabajta podataka. Dekomprimiran te sa nekolicinom generiranih konfiguracijskih datoteka na određenom operativnom sustavu sadrži tek nešto ispod pedeset megabajta podataka. Uzevši u obzir ostale moderne pogone za izradu video igara Godot pogon zauzima i do nekoliko stotina puta manje prostora na tvrdom disku nego ostali moderni popularni pogoni.

No ukoliko se određeni projekt želi izvesti kao samostalna aplikacija, potrebno je skinuti još i takozvane predloške izvoza za izvoz projekta. Ti su predlošci određeni lanci alata izgradnje, koji se mogu konfigurirati i koji izvode proces pakiranja godot projekta za željenu platformu u samostalnu aplikaciju. Službeni predlošci izvoza projekta za sve podržane platforme zauzimaju dodatnih 361 megabajta zapisa na disku, no za određene platforme poput iOS i Android operativnih sustava, predlošci zahtjevaju pripadne SDK alate razvoja za uspješan izvoz. Čest je slučaj da korisnici pišu vlastite predloške izvoza prema vlastitim potrebama.

Na ovaj način korisnici pogona nemoraju znati previše o platformama na koje projekt cilja da bi sam projekt jednom u produkciji bio osposobljen za rad na istoj. No ukoliko žele korisnici imaju i dalje mogućnost utjecaja na svaki korak izgradnje projekta za određenu paltformu.

Kako bi olakšao automatizaciju upravljanja projektima a posebice distribuciju, Godot dolazi sa vlastitim jednostavnim alataom za komandnu liniju koji služi za upravljanje projektima te se izvoz projekta može vršiti tim alatom iz komandne linije, naredbom godot sa parametrom --export. Izvoz se također može vršiti direktno iz uređivača pogona.

Pogon nudi i sustav oznaka za svojstva projekata. Osim davanja informativnog opisa i popisa mogućnosti i svojstava projekta, korisnik može slagati specifične postavke za izvoz projekta na temelju oznaka svojstva. Također, pogon nudi mogućnost dodavanja vlastitih oznaka uz predefinirane.

Osim vlastitih SDK alata određene platforme poput Android operativnog sustava i Windows Universal Platform platforme podrazumijevaju potpisivanje projekata vlastitim sigurnosnim sustavima potpisa autentičnosti. Te funkcionalnosti Godot pogon ne pruža te se očekuje od korisnika da sam iskoristi adekvatne alate ovisno o platformi na koju projekt cilja.

11. Zaključak

Uzevši u obzir sadržaj ovog rada smatram da pokazuje kako je Godot u potpunosti prikladna solucija ne samo za izradu video igara već i za ostale vrste aplikacija koje moraju dinamično prikazivati određeni sadržaj u realnom vremenu. Unatoč svemu opisanom, pogon nudi još mnoštvo funkcionalnosti te se bavi i sa dodatnim aspektima izrade video igara i sličnih aplikacija koje ovaj rad nije spomenuo, kao što su zvuk, virtualna stvarnost, mreže i ostalo...

Unatoč tome, da se uvidjeti da prevelika nedorečenost načina strukturiranja koda i logike projekata kao i manjak naprednih funkcionalnosti prikaza za izvođenje video igara, može godot pogon činiti odbojnim za velike tvrtke i zajednice kojima je cilj izraditi igru velikih opsega u što kraćem roku na već provjerene načine. No kad su u pitanju većina ostalih potencijalnih korisnika pogona, a pogotovo početnici, hobisti i korisnici koji žele veoma brzo razviti određen prototip, s obzirom na cijenu, izvedbu i otvorenost koda smatram da je Godot najoptimalniji izbor od svih trenutno ponuđenih pogona na tržištu.

Sustav čvorova i scena dovoljno je jednostavan i intuitivan da bilo tko može krenuti raditi s njim nešto, a dovoljno moćan da se kompleksne logike i strukture mogu elegantno realizirati kao malo kada do sada u području razvoja video igara. Osim toga pogon je radi svega navedenog odličan kandidat za akademsko proučavanje i korištenje kao i za uvođenje i korištenje na akademskim institucijama za mnogobrojne domene, poput razvijanja simulacija, simulatora, raznih pogona za druge namjene, didaktičkih igara i ostalog.

Treba napomenuti kako je zajednica također sve aktivnija te broji svakim danom sve više članova, te kako je gotovo sigurno da će se konstantno dodavati nove funkcionalnosti i mogućnosti pogona a postojeće dorađivati i proširivati u narednih nekoliko godina.

Literatura

[1] En.wikipedia.org. (2018). *Godot (game engine)*. [online] Available at: https://en.wikipedia.org/wiki/Godot_(game_engine)

[2] Engine, G. (2018). *Godot Engine - Free and open source 2D and 3D game engine*. [online] Godotengine.org. Available at: https://godotengine.org/

[3] Docs.godotengine.org. (2018). *Godot Docs – 3.0 branch — Godot Engine latest documentation*. [online] Available at: http://docs.godotengine.org/en/3.0/

[4] Engine, G. (2018). *Godot Engine - Why does Godot use Servers and RIDs?*. [online] Godotengine.org. Available at: https://godotengine.org/article/why-does-godot-use-servers-and-rids