Implementacija igre za učenje koncepata programiranja u razvojnom okruženju Unreal engine 4

Drozina, Adrian

Master's thesis / Diplomski rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Pula / Sveučilište Jurja Dobrile u Puli**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:137:660788

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-25**

Repository / Repozitorij:

Digital Repository Juraj Dobrila University of Pula

Sveučilište Jurja Dobrila u Puli
Fakultet informatike u Puli

Adrian Drozina

Implementacija igre za učenje koncepata programiranja u razvojnom okruženju Unreal Engine 4

Diplomski rad

Pula, svibanj, 2019.

Sveučilište Jurja Dobrila u Puli
Fakultet informatike u Puli

Adrian Drozina

Implementacija igre za učenje koncepata programiranja u razvojnom okruženju Unreal Engine 4

Diplomski rad

JMBAG: 0303046442, redoviti student
Studijski smjer: Informatika

Predmet: Dizajn i programiranje računalnih igara

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Tihomir Orehovački

Pula, svibanj, 2019.

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Adrian Drozina, kandidat za magistra informatike, ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____, _____ godine

IZJAVA
o korištenju autorskog djela

Ja, Adrian Drozina dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom Implementacija igre za učenje koncepata programiranja u razvojnom okruženju Unreal Engine 4 koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, _____ (datum)

Potpis

_____

Sadržaj

1 Uvod 1

2 CodeBreaker 3

3 Kreiranje koda 5

3.1 Pokretanje koda 11

4 Leksička analiza 12

4.1 UToken 12

4.2 ULexer 13

5 Parser 15

6 Semantička analiza 21

6.1 USemanticAnalysisPhase 22

6.2 USAP_ClassAnalysis 23

6.3 USAP_ConvertVariableCallsToType 27

6.4 USAP_FunctionAnalysis 29

6.5 USAP_OperatorAnalysis 34

6.6 USAP_StoreStatics 36

6.7 USAP_ExpressionAnalysis 39

6.7.1 EReturnMode 39

6.7.2 FExpressionReturn 40

6.7.3 FExpressionPath 41

6.8 USAP_StaticAnalysis 48

6.9 USAP_FlowControlAnalysis 51

6.10 USAP_ReturnAnalysis 52

6.11 USAP_Final 56

7 Pokretanje koda 57

7.1 UValue 57

7.2 FStackFrame 58
7.3 UCodeRunner 59
7.4 URuntimeState 61
7.4.1 URTS_VariableCall 62
7.4.2 URTS_MemberAccess 64
7.4.3 URTS_FunctionCall 65
7.4.4 URTS_ChainedFunctionCall 69
7.4.5 URTS_FunctionDefinition 72
7.4.6 URTS_Return 73
8 CodePlayerController 74
9 CodeComponent 76
9.1 Klase u CodeComponent-u 76
9.1.1 UPredefinedClass 77
9.1.2 Ugrađene klase 78
9.2 Funkcije 79
9.2.1 Kompleksna funkcija 80
9.2.2 Jednostavna funkcija 81
9.2.3 Ugrađene funkcije 83
10 Spremanje podataka 86
10.1 Trajno spremanje podataka 86
10.2 Privremeno spremanje podataka i resetiranje razina 87
10.2.1 UPersistentLevelData 88
10.2.2 Sustav zagonetki 89
11 Grafičko sučelje 92
12 Zaključak 94
13 Literatura 96
14 Popis koda 96

15 Popis slika 99
16 Sažetak 100
17 Summary 100
18 Ključne riječi 101
19 Keywords 101

1 Uvod

Znanje o programiranju je vrlo traženo, no često samo učenje programiranja može biti vrlo teško, pogotovo u početku. Sam odabir jezika može bit težak izbor za početnika, pošto treba uzeti u obzir sintaksu te same mogućnosti jezika. Neki jezici su kompleksniji od drugih, te sam velik izbor može biti barijera pri početku učenja. Učenje se u početku često svodi na prepisivanje koda, te zatim njegovo objašnjavanje. Iako se na taj način može naučiti sintaksa jezika, to često dovodi do nemogućnosti samostalnog rješavanja problema.

Učenje programiranja uz rješavanje problema je vrlo korisno, no za početnike vrlo teško. Znat rješenje na neki problem, a ne bit u mogućnosti ga implementirati zbog sintaktičkih grešaka može biti vrlo frustrirajuće. Osim sintaktičkih grešaka mogu se pojavljivati i drugi problemi, poput nenapisanog Boilerplate koda. Iz tih razloga je Python vrlo popularan jezik, ima jednostavnu sintaksu i ne sadrži gotovo nikakav Boilerplate. No Python je dinamičan jezik, te korisnici koji žele naučit neki drugi jezik, poput C#, morat će se prilagodit i sintaksi jezika i načinu rješavanja problema, pošto je C# statičan jezik.

Postoje rješenja za te probleme, no ona nisu idealna. Jedno od popularnijih je vizualno programiranje, tj. kreiranje koda bez pisanja teksta. Korištenjem grafičkog sučelja kao alata za programiranje se mogu izbjeći problemi pisanja koda, točnije učenja sintakse. Tijekom kreiranja koda se odmah može javiti greška korisniku, ukoliko se ona dogodi, pošto nije potrebno kompajlirati cijeli kod da bi se ona pronašla. Također je puno jednostavnije „zaštititi korisnika od samog sebe“, pošto je moguće ne dopustit da se greška kreira na način da se ponude samo validne opcije. To je teško, ili nekad čak i nemoguće kod pisanja koda.

Česti problem kod vizualnog programiranja je sličan Python-u. Korisnik se mora priviknuti klasičnom pisanju koda, te mora učiti sintaksu jezika. Pošto je korisnik već upoznat sa konceptima programiranja, vjerojatno će lakše rješavati probleme namijenjene početnicima, no opet se javlja isti problem gdje je nemogućnost rješavanja problema samo zbog nedostatka znanja o sintaksi frustrirajući. Iako je učenje korištenjem vizualnog programiranja često jednostavnije od korištenja isključivo klasičnog programiranja, činjenica da je potrebno naučiti koncepte, te zatim pisanje koda zasebno zahtjeva više vremena i prilagođavanja jeziku.

Jedno od elegantnih rješenja za te probleme je interaktivno programiranje, tj. okolina u kojoj korisnik piše kod te odmah može vizualno otkriti šta taj kod radi. Na taj način korisnik može početi učit koristeći jednostavnu sintaksu, te eksperimentirati kako bi točno otkrio kako kod radi, na način da dodaje, briše ili mijenja naredbe. Samim eksperimentiranjem korisnik uči, te piše sve kompleksniji i kompleksniji kod. Drugi naziv za to je učenje uz igru.

Iako učenje programiranja uz igru nije novi koncept, mnoge igre se više fokusiraju na samo rješavanje problema, te često imaju neki način vizualnog programiranja. Postoje igre koje koriste pisani kod, no one se često ne fokusiraju na samo učenje koncepata programiranja. Iz tog razloga je kreirana CodeBreaker igra.

CodeBreaker je igra kreirana sa svrhom da nauči korisnika koncepte programiranja na interaktivan način. To radi na način da daje slobodu korisnika da eksperimentira sa kodom, dok u isto vrijeme zadaje zagonetke koje korisnik mora riješiti koristeći novo naučene koncepte, poput kreiranja funkcija, nasljeđivanja i sl. Više o igri u sljedećem poglavlju.

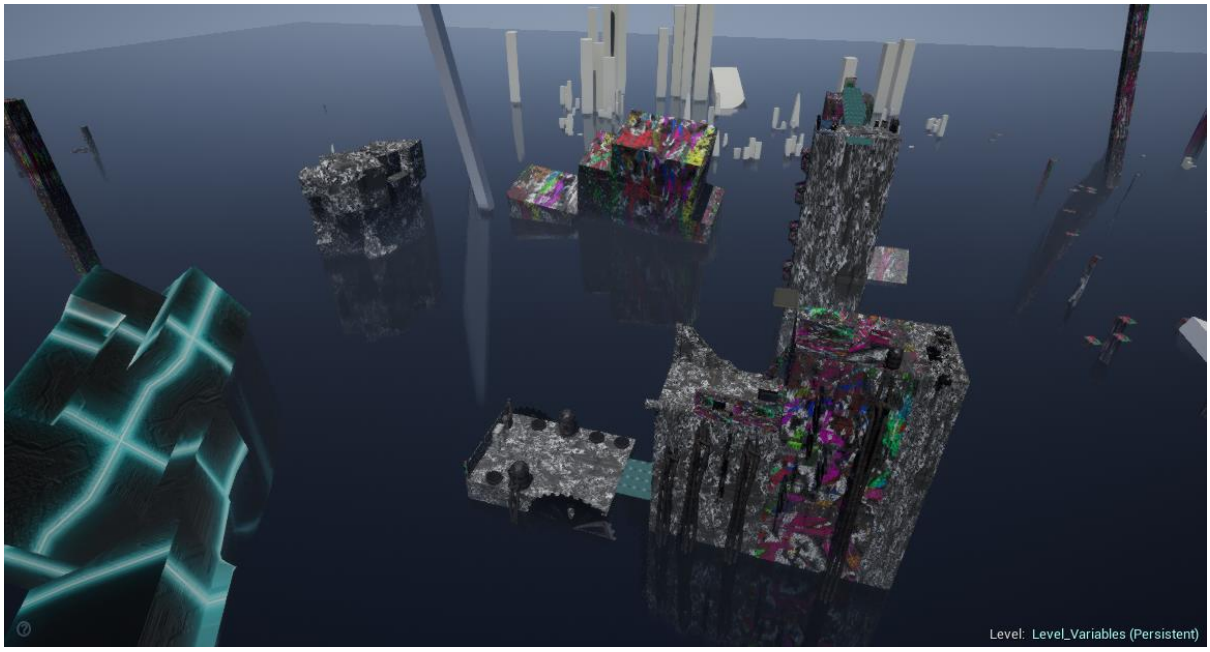
CodeBreaker je kreiran koristeći Unreal Engine 4 verziju 4.19.2. Unreal Engine je Game Engine otvorenog koda(iako se ne koristi isključivo u kreiranju igara, nego se koristi i kod arhitekatske vizualizacije, u auto industriji, u simulacijama, u filmskoj industriji, itd.). Iako je kreiran u C++ jeziku, sadrži svoj integrirani skupljač smeća (garbage collector). Samim time, kreiranjem CodeBreaker koda koristeći Unreal Engine framework sam kod ima pristup skupljaču smeća.

Ovaj rad će se većinom koncentrirati na objašnjavanje same implementacije igre, ponajviše na način na koji je kod koji korisnik koristi unutar igre implementiran. No prije toga će sljedeća dva poglavlja dati više podataka o samoj igri, te o razlozima na koji je sintaksa koda igre definirana na način na koji je.

Za implementaciju koda će se opisati cijeli sustav od analize teksta koda (Lexer), te način na koji se kreiraju tokeni iz tog teksta, sama analiza tih tokena (Parser), opis stabla apstraktne sintakse (Apstract syntax tree, AST) te njegovu semantičku analizu (Semantic Analysis), te na kraju samo pokretanje koda. Uz to će se prikazati i način na koji je kod implementiran u ostatak igre, te kako omogućuje korisniku da preko njega modificira igrin svijet.

2 CodeBreaker

CodeBreaker je igra koja primarno služi za učenje koncepata programiranja. Igra se odvija unutar simulacije kreirane sa strane Advanced Security Solutions tvrtke, koja služi za kreiranje i testiranje raznih sigurnosnih mjera. Samo testiranje se provodilo preko androida unutar simulacije koji su zatim pokušavali zaobići te sigurnosne sustave. S vremenom se simulacija počela kvariti i korumpirati, te je stoga napuštena skupa sa androidima. Dijelovi simulacije koji su korumpirani većinom imaju jednostavne oblike, te sadrže „pokvarene“ teksture na raznim objektima.



Slika 1. Primjer dijela igrinog svijeta unutar korumpirane simulacije.

Izvor: vlastita izrada.

Igrač igra kao jedan od tih androida koji se probudio unutar pokvarene simulacije, te pokušava izaći iz nje. Kako bi to postigao, mora probiti sigurnosne mjere unutar simulacije. To čini hakiranjem raznih objekata unutar igrinog svijeta te manipulacijom koda manipulara svoju okolinu. Kada igrač hakira neki objekt, on dobije pristup svih funkcijama i varijablama tog objekta, te ih zatim koristi kako bi se sigurnosna mjera zaobišla.

Igra se sastoji od 8 glavnih razina, te svaka uči korisnika određeni koncept programiranja, poput varijabli, petlji, grananja, nasljeđivanja i slično. Svaka razina se sastoji od nekoliko zagonetki koje korisnik mora riješiti kako bi došao do kraja te razine. Nakon što se riješe sve zagonetke dobiva se pristup portalu koji vodi na sljedeću razinu, te se prolazom kroz njega rješava cijela razina. Kako igrač prolazi kroz razine, tako je utjecaj korupcije unutar simulacije sve manji.

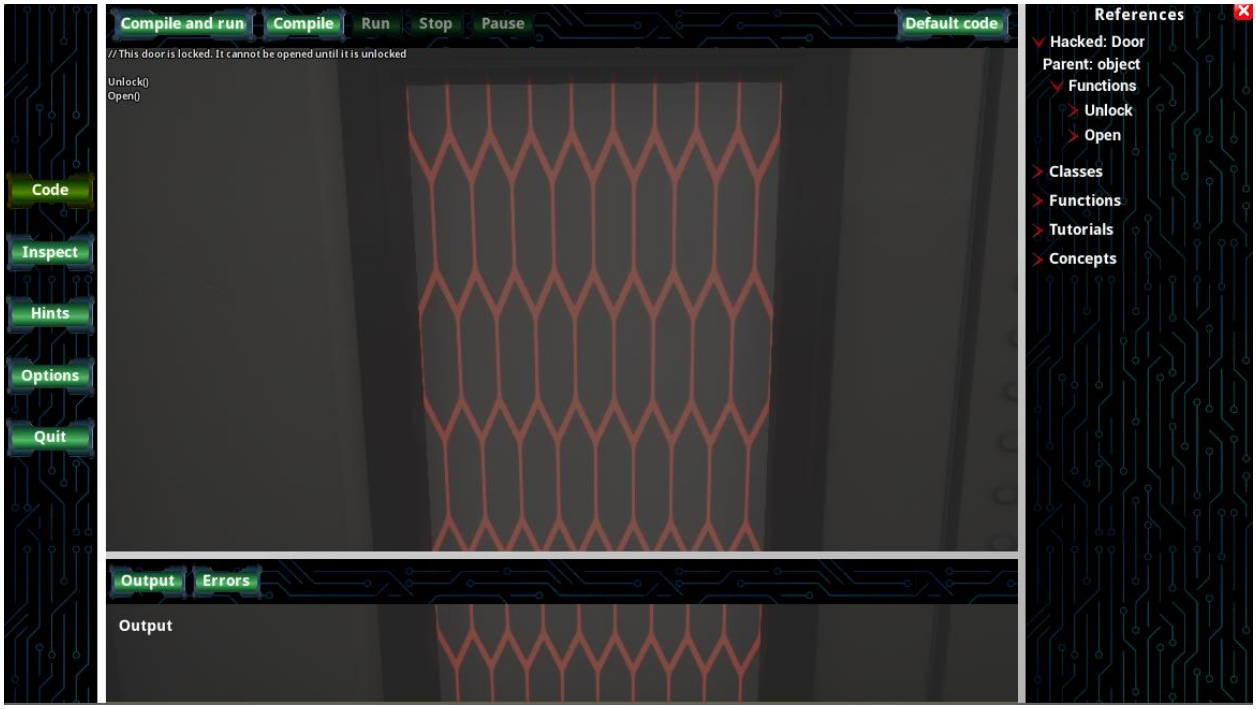
Slika 2. Primjer dijela simulacije unutar igre koja nije korumpirana.

Izvor: vlastita izrada

Na svakoj razini korisniku se pojasni određeni koncept programiranja koji je potrebno koristiti kako bi se ta razina riješila. Uz samo objašnjenje koncepta, korisniku se daju i primjeri koda koje može analizirati, pokrenuti i modificirati, kako bi ga lakše naučio. Nakon što ih dobije prvi put, te informacije su dostupne u svakom trenutku, neovisno o razini u kojoj se nalazi.

Glavna mehanika igre je, dakle, pisanje i pokretanje koda. Koristeći kod, korisnik manipulira okolišem te rješava zagonetke. Iako korisnik može pisati kod u bilo kojem trenutku te na bilo kojoj lokaciji unutar razine, najčešće je potrebno hakirati određene objekte u igri kako bi se zagonetka riješila. Hakiranje je proces u kojem korisnik

direktno piše i pokreće kod u nekom objektu, te samim time korisnik ima pristup svim funkcijama i varijablama unutar tog objekta. Međutim, kako bi korisnik uspješno hakirao neki objekt, potrebno je da bude unutar određene udaljenosti od tog objekta, te da ima jasan pogled na njega (line of sight). Samim time korisnik ne može rješavati sve zagonetke na jednoj lokaciji, te mora koristiti okoliš kako bi došao do lokacije gdje može hakirati potrebne objekte. Primjer hakiranja je prikazan na sljedećoj slici.



Slika 3. Primjer hakiranja unutar CodeBreaker igre.

Izvor: vlastita izrada

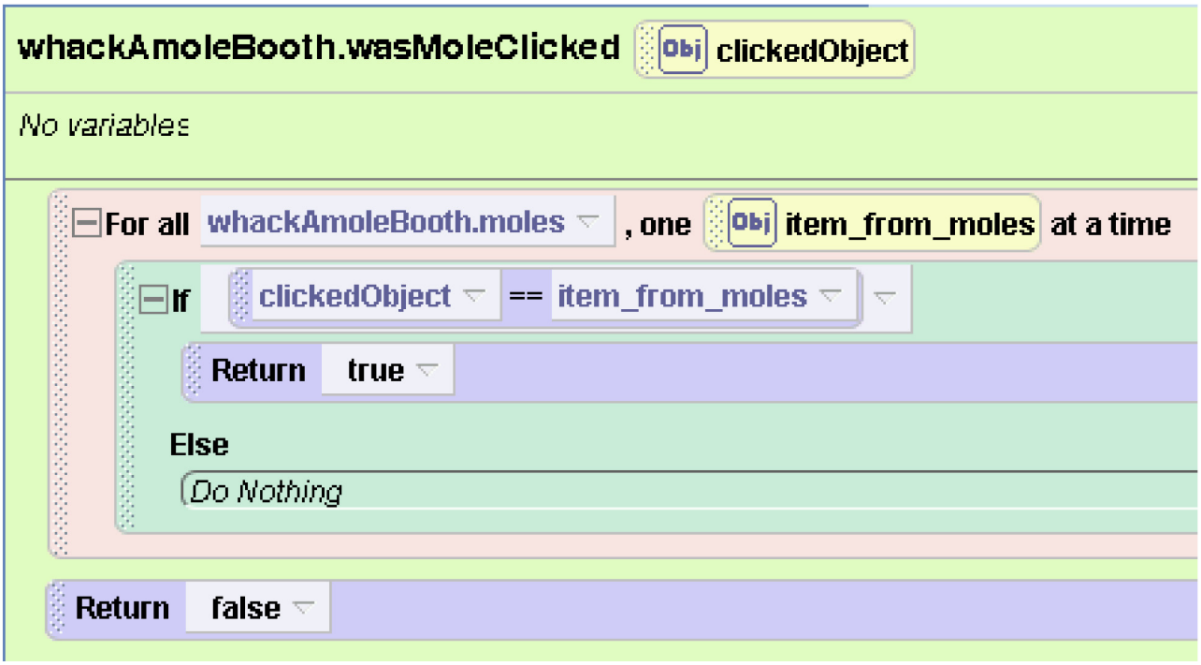
Više informacija o dizajnu igre može se pronaći u radu kolege Tonija Bilete koji se bavi dizajnom CodeBreaker igre, te o korištenju igara u svrhu učenja (Bileta, 2019).

3 Kreiranje koda

Jedna od najvažnijih odluka je način na koji će korisnik kreirati kod, pošto će na taj način korisnik provesti većinu vremena. Stoga postoje dvije opcije: klasično pisanje koda i vizualno skriptiranje. Oboje imaju svoje prednosti i mane.

Vizualno skriptiranje ima prednost u tome što korisnik ne mora učiti sintaksu jezika, što olakšava kreiranje koda. Također, vizualno skriptiranje ima prednost u tome što se tijekom samog kreiranja koda mogu otkriti određene greške, na primjer, dodavanje pogrešnog tipa podatka kao argument u funkciji. Takve greške se mogu lakše otkriti dok se kod kreira, te se jednostavno ne dopusti korisniku da učini grešku, dok je kod klasičnog pisanja koda to ponekad nemoguće (Maloney, 2010). Vizualno skriptiranje se najčešće dijeli u dvije grupe: skriptiranje bazirano na stogu i skriptiranje bazirano na grafu.

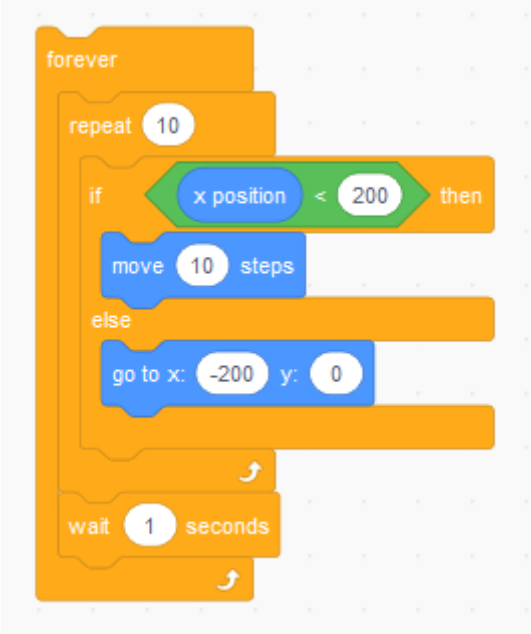
Skriptiranje bazirano na stogu koristi blokove koji predstavljaju dijelove koda, te se oni slažu u stog. Kada se kod pokreće, blokovi se izvršavaju od vrha prema dnu. Blokovi često sadrže tekst koji objašnjava koju akciju blok izvodi, te može sadržavati dodatne parametre. Dobar primjer vizualnog skriptiranja baziranog na stogu jest Alice programski jezik, koji je prikazan na slici ispod.



Slika 4. Primjer Alice vizualnog programskog jezika.
Izvor: Cooper, 2010.

Prednost vizualnog skriptiranja baziranog na stogu je u tome što blokovi sadrže prirodan jezik, zbog čega je jednostavno otkriti što taj blok radi, te sama struktura stoga

prikazuje redoslijed egzekucije koda. Dodatna prednost je u tome što blokovi mogu sadržavati druge blokove, kao u slučaju petlji. Tada je vrlo jednostavno vidjeti gdje petlja počinje i gdje završava. Međutim problemi nastaju tijekom grananja. Pošto se kod nalazi unutar stoga, tada je prikaz grananja težak, pošto se obje grane moraju prikazati, a samo jedna će se izvršiti, što se protivi samoj linearnoj prirodi stoga. Taj problem je prikazan u sljedećoj slici, koristeći programski jezik Scratch.



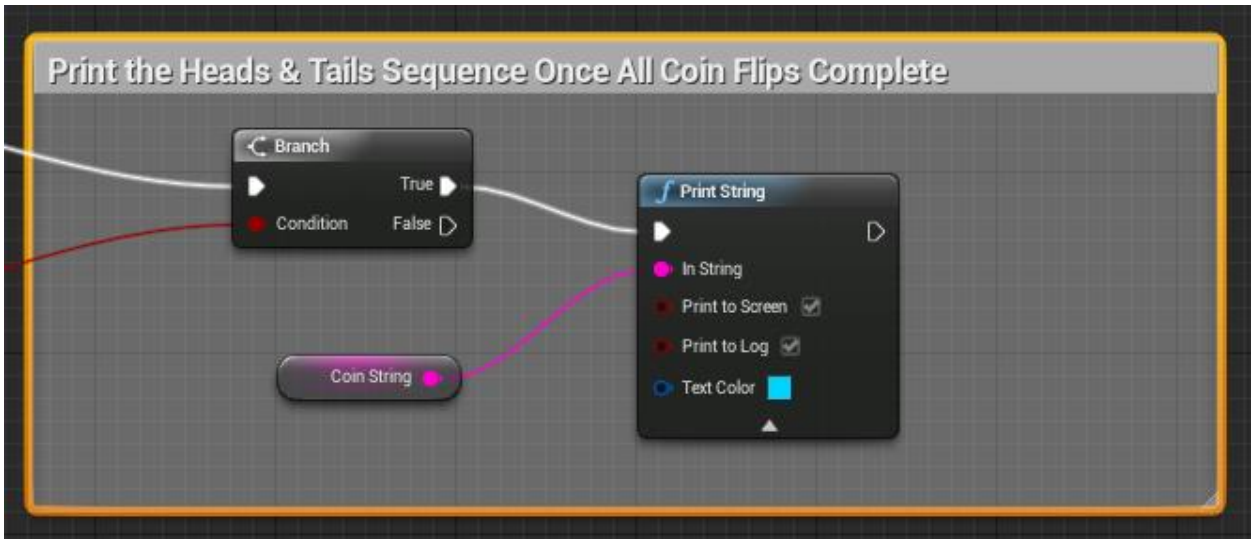
Slika 5. Primjer petlji i grananja vizualnog programskog jezika Scratch.

Izvor: vlastita izrada

Vizualno skriptiranje bazirano na grafu također sadrži blokove koji predstavljaju dijelove koda, uključujući i varijable. Blokovi efektivno predstavljaju čvorove u grafu, te najčešće sadrže ulaze sa lijeve strane, te izlaze sa desne. Prednosti virtualnog skriptiranja baziranog na grafu su mnoge.

Pošto se radi o grafu, same pozicije blokova nisu važne, što omogućuje korisniku više mogućnosti organizacije. Pošto se takvi kodovi najčešće kreiraju horizontalno, za razliku od stoga koji se kreira vertikalno, više koda se može prikazati na ekranu odjednom. Unreal Engine 4 ima vizualni programski jezik baziran na grafu zvan

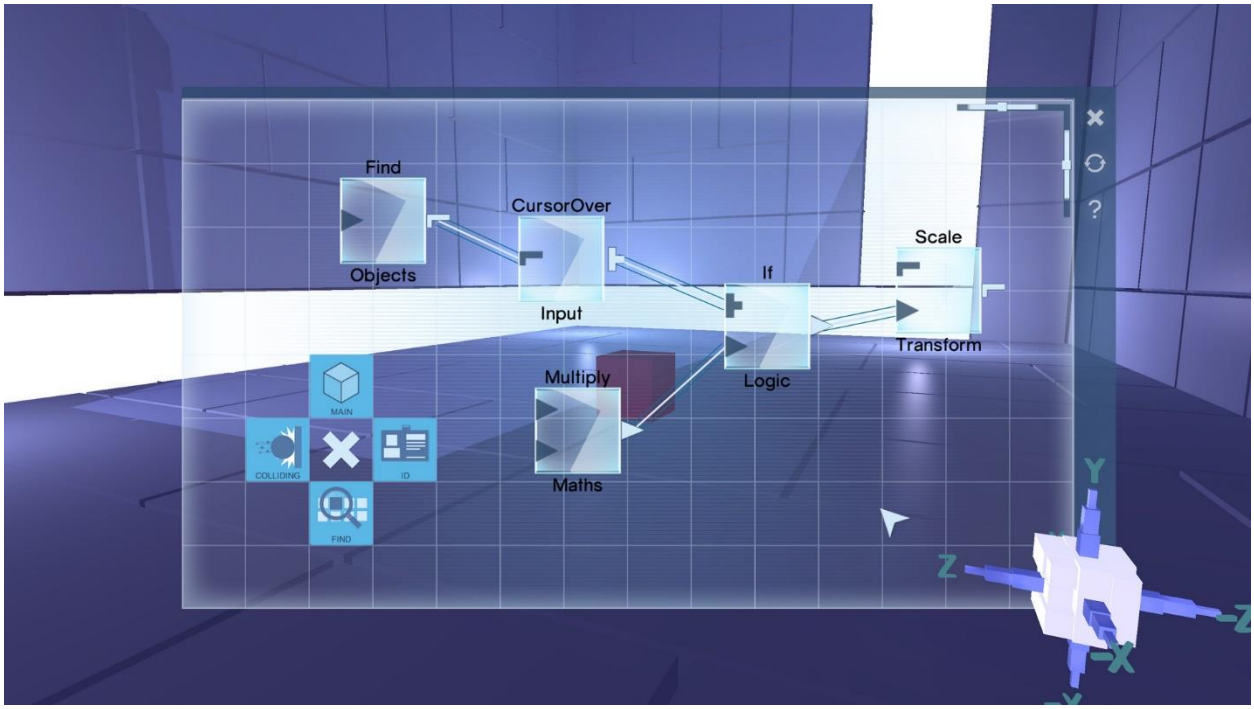
Blueprints, te ima mogućnost umetanja komentara direktno na blok, ili na skupinu blokova, kao što je prikazano na sljedećoj slici.



Slika 6. Primjer komentara koji obuhvaća nekoliko blokova u programskom jeziku Blueprints. Izvor: „komentari u Blueprint sustavu“, 2019.

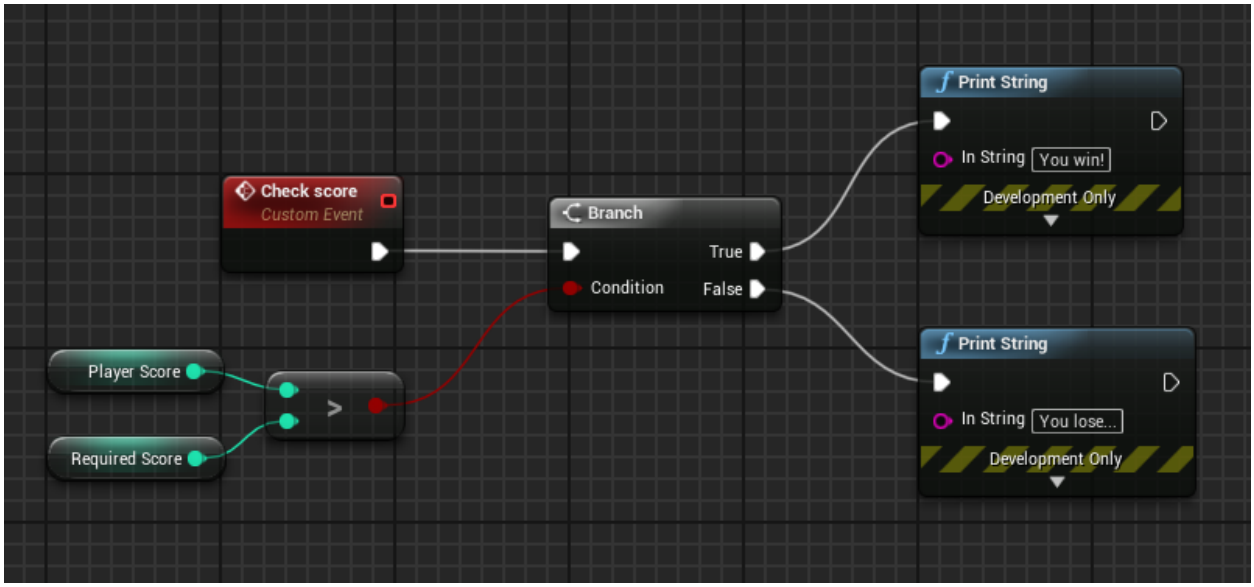
Još jedna prednost je mogućnost prikaza informacija. Blokovi kod jezika baziranog na stogu su često limitirani svojom veličinom, što samim time limitira prostor na kojem mogu prikazati informacije. Kod grafa to nije problem, pošto je moguće pomicati blokove što dopušta da zauzmu potrebnu veličinu bez da se preklapaju sa drugim blokovima.

Prikaz tipova podataka je vrlo jednostavan, Blueprints sustav koristi boje kako bi se tipovi lakše raspoznali, dok sustav koji koristi igra Glitchspace koristi različite oblike kako bi prikazali različite tipove podataka na ulazima i izlazima.



Slika 7. Programski kod igre Glitchspace. Simboli na blokovima predstavljaju različite tipove podataka. Izvor: „primjer koda Glitchspace igre“, 2019.

Također je važno napomenuti da je sam graf koda često vrlo sličan samom stablu apstraktne sintakse tog istog koda. Samim time, tijek egzekucije koda je vrlo lagano pratit, pogotovo u slučaju grananja.



Slika 8. Primjer grananja u programskom jeziku Blueprints.

Izvor: vlastita izrada

S druge strane, teže je raspoznati gdje se nalaze petlje, pošto, za razliku od verzije sa stogom, ne postoji nikakva indentacija ili neki drugi mehanizam kojim bi se prikazalo koji kod se nalazi unutar petlje. U tom slučaju je potrebno tražiti blok koji predstavlja tu petlju.

Također treba napomenuti da je potrebno naučiti koristiti korisničko sučelje, te ukoliko ono nije intuitivno i lagano za korištenje, sve prednosti vizualnog skriptiranja se gube. Također, ukoliko korisnik nauči koncepte programiranja koristeći vizualno skriptiranje, te poželi pisati kod u nekom drugom popularnom jeziku, tada opet mora učiti sintaksu tog jezika.

Samim time, CodeBreaker koristi klasičan sustav pisanja koda, no potrebno je naglasit da je sintaksa samog koda dizajnirana na način da bude intuitivna i jednostavna za korištenje. Kako bi se to postiglo, zadani su određeni ciljevi, a to su:

- Sintaksa slična jezicima iz C obitelji. Jezici iz te obitelji, poput C++, C# i Jave su vrlo popularni, te je vrlo vjerojatno da će korisnik koristiti te jezike nakon što završi igru. Pošto je sintaksa ovog koda slična sintaksi tih jezika, prebacivanje sa igre na njih je relativno jednostavan proces.

- Smanjenje korištenja boilerplate koda. Na primjer, Kako bi se ispisao „Hello, world!“ u C++ jeziku, potrebno je include-at iostream biblioteku, te napisati kod za ispisivanje unutar main funkcije. Početnici često ne razumiju zašto je sve to potrebno, što dovodi do konfuzije, ili do loših praksi, poput pisanja using namespace std unutar header-a. Eliminiranjem boilerplate-a korisnik piše samo ono što je uistinu potrebno, te smanjuje mogućnost grešaka. U tom pogledu, kod je sličan Python jeziku.
- Izbjegavanje nepotrebnih znakova. Iz sličnog razloga koji je prije naveden, jednostavniji kod je lakši za razumjet. Iz tog razloga nije potrebno upisati nikakav znak na kraju naredbe, kao u jezicima iz C obitelji, koji moraju imati ; znak na kraju. U slučaju da se ; stavi na kraju naredbi, kompajler neće baciti grešku. Razlog tome je što korisnici koji su već koristili C++ ili slične jezike će iz navike napisati ; na kraju naredbi, te bi bacanje greške u tom slučaju samo ometalo korisnika, dok bi kod svejedno mogao biti valjan. Također, to dopušta i mogućnost da se određeni dijelovi koda iz C++ i sličnih jezika mogu direktno koristiti u igri, što omogućuje korisniku lakše eksperimentiranje.
- Osim toga, također se izbjegava korištenje posebnih znakova ukoliko se mogu zamijeniti sa ključnom riječi koja objašnjava njihovo korištenje. Na primjer, prilikom definicije klase u C++ jeziku koristi se : znak kako bi se prikazalo nasljeđivanje. To bi moglo biti neintuitivno korisnicima, te se stoga umjesto : znaka koristi ključna riječ „inherits“, kao u Java jeziku.

3.1 Pokretanje koda

Osim kreiranja koda, korisniku je važno i način na koji se kod pokreće, iako na prvu to ne izgleda vrlo važno. Pošto se kod u igri koristi kako bi se manipuliralo okolišem, važno je da funkcije imaju mogućnost da traju određeno vrijeme. Na primjer, ukoliko je potrebno napisati kod koji pomiče platformu od jedne točke na drugu i nazad, važno je da funkcija koja pomiče tu platformu ne završi dok platforma ne dođe do svoje destinacije. U protivnom će se platforma početi micat prema destinaciji i odmah vratiti nazad. Stoga je važno da svaka funkcija može završit nakon nekog određenog vremena ili događaja, te se nakon toga nastavlja pokretanje ostalog koda.

Također postoji mogućnost da će korisnik slučajno (ili namjerno) kreirati beskonačnu petlju. Ukoliko to dogodi, a u petlji se ne nalazi ni jedna funkcija koja zaustavlja egzekuciju koda, tada će se igra zamrznut. Pošto je nemoguće riješit problem zaustavljanja (halting problem), tj. otkriti ukoliko će bilo koji zadani program ikada stati, potrebno je pronaći drugo rješenje za taj problem.

Stoga je potrebno da se kod može zaustaviti u bilo kom trenutku između pokretanja naredbi, te da se nakon bilo kojeg vremena egzekucija nastavi. To dopušta da se egzekucija koda u bilo kojem trenutku može pauzirati i ponovo pokrenuti, neovisno o samom kodu. Samim time, ukoliko se nakon određenog broja naredbi ili koraka kod pauzira, te se ponovo pokrene u sljedećoj sličici (frame-u), igra se nikada neće zamrznuti. Tada korisnik može ručno zaustaviti kod. To ima i dodatnu prednost što same performanse kod pokretanja koda nisu vrlo bitne, pošto se samo pokretanje može odvijati kroz nekoliko sličica. U tom slučaju sama egzekucija koda izgleda trenutna, dok bi se u slučaju da se sav kod pokrene odjedanput osjetila kratka pauza u igri koja bi bila vrlo primjetna.

4 Leksička analiza

Prva faza procesiranja koda je leksička analiza. U njoj se uzima napisani kod u obliku teksta, i pretvara se u listu tokena (Aho et al., 2006.). Ti tokeni su potrebni u sljedećoj analizi.

Sama implementacija leksičke analize se sastoji od dva dijela, Lexer-a, koji pretvara napisani tekst u listu tokena, te sami tokeni.

4.1 UToken

UToken je klasa koja predstavlja tokene, te je prilično jednostavna. Sadrži samo jednu varijablu tipa FString zvanu TokenName, te virtualnu funkciju GetTokenName, koja vraća vrijednost te varijable. Ta varijabla se koristi isključivo za debugging. Od UToken klase nasljeđuju mnoge druge klase koje predstavljaju konkretne tokene, poput raznih otvorenih i zatvorenih zagrada, vrijednosti tipa bool, int, float i string, ključnih riječi, i sl. Te klase se nazivaju po tokenu koji predstavljaju, te imaju predznak T, tako na

primjer, token koji predstavlja token zareza, se zove T_Comma, token koji predstavlja integer se zove T_IntLiteral… Svaka reprezentacija tokena je vlastita klasa što omogućuje spremanje specifičnih podataka u te tokene. Za razliku od standardnog načina kreiranja tokena, gdje se token predstavlja sa parom stringova, gdje je prvi string tip tokena a drugi vrijednost, ovaj sustav dopušta da se u token koji predstavlja float direktno spremi float vrijednost umjesto string vrijednosti. Samim time se eliminira potreba da se kasnije spremljene vrijednosti konvertiraju iz stringa u zadani tip podatka.

4.2 ULexer

ULexer je klasa koja predstavlja lexer, koja preko funkcije Tokenize pretvara napisani text u listu tokena. Kod te funkcije je prikazan ispod.

```
TArray<UToken*> ULexer::Tokenize(const FString Source)
{

	SourceText = Source.GetCharArray();
	TArray<UToken*> result;
	int32 index = 0;

	while (SourceText.IsValidIndex(index) && SourceText[index] != '\0')
	{
		UToken* tok = Lex(index);
		if (tok)
		{
			result.Add(tok);
		}
		else
		{
			ThrowError("Not all tokens processed, unprocessed token index is
" + FString::FromInt(index));
			return result;
		}
	}

	return result;
}
```

Kod 4.1 Funkcija Tokenize u klasi Lexer. Pretvara tekst koda u listu tokena.

Izvor: vlastita izrada

Tokenize funkcija uzima napisani tekst, te ga pretvara u listu character-a, te sprema indeks trenutnog character-a u toj listi. Dok god je indeks validan, poziva se Lex funkcija, koja prolazi kroz listu, te vraća token ukoliko pronađe odgovarajuću sekvencu znakova. Lex funkcija je prikazana ispod.

```
UToken* ULexer::Lex(int32& StartIndex)
{
	UToken* Token = nullptr;

	if (LexComment(StartIndex, StartIndex, Token)) return Token;
	if (LexBool(StartIndex, StartIndex, Token)) return Token;
	if (LexFloat(StartIndex, StartIndex, Token)) return Token;
	if (LexInt(StartIndex, StartIndex, Token)) return Token;
	if (LexString(StartIndex, StartIndex, Token)) return Token;
	if (LexWhitespace(StartIndex, StartIndex, Token)) return Token;

	for (FLexData currentLexData : LexData)
	{
		if (LexBySequence(StartIndex, currentLexData, StartIndex, Token)) return
Token;
	}

	if (LexIdentifier(StartIndex, StartIndex, Token)) return Token;

	if (SourceText.IsValidIndex(StartIndex))
		ThrowError("Unknown token " + FString::Chr(SourceText[StartIndex]) + "
detected.");
	else ThrowError("Error");
	return nullptr;
}
```

Kod 4.2 Funkcija Lex u Lexer klasi, koja analizira sekvencu znakova i vraća odgovarajući token. Izvor: vlastita izrada

Lex funkcija prvo pokušava pronaći da li sekvenca odgovara komentaru, bool, int, float ili string tipu podataka, te da li je whitespace (whitespace je naziv za razmak, tab ili novi red). Razlog tome je taj što svaki taj token se može spremiti neku vrijednost. Na primjer, ukoliko se otkrije sekvenca znakova 24.58, tada se kreira T_FloatLiteral token, te se u njega sprema float vrijednost 24.58.

Nakon toga se traže sve sekvence koje nemaju vrijednost, poput zagrada, ključnih riječi i slično. To se čini korištenjem liste koja sadrži podatke tipa FLexData.

FLexData je struktura koja sadrži tri varijable. Prva je Sequence, tipa string, koji predstavlja sekvencu znakova. Druga je TokenClass, koja predstavlja tip tokena, te

treća je bool AllowNextAlphanumeric, koja označuje da li je dozvoljeno da se nakon odgovarajuće sekvence pojavi neki alfanumerički znak. Na primjer, u slučaju ključne riječi „if“, AllowNextAlphanumeric ima vrijednost laž, pošto „ifa“ nije validna ključna riječ, pošto nakon znakova „if“ nije dopušten alfanumeričan znak. U drugu ruku, za sekvencu znakova „==“ je dopušten, pošto „a==b“ je validna sekvenca, kao i „a == b“.

Važno je napomenuti da lexer prolazi kroz listu, te vraća prvu odgovarajuću sekvencu. Stoga je redoslijed sekvenci u listi važan. Ukoliko je redoslijed pogrešan, te se u tekstu nalaze znakovi „>=“, lexer bi ih protumačio kao jedan token za znak „>“ i jedan token za znak „=“. Stoga je važno da se token koji predstavlja znak „>=“ nalazi na listi prije znaka „>“.

Ukoliko se ne pronađe niti jedna sekvenca koja odgovara, tada se pokuša kreirati T_Identifier token, koji predstavlja identifikator. Ukoliko sekvenca nije valjani identifikator (npr. započinje brojem), tada se baca greška i kompajliranje prestaje.

5 Parser

Sljedeća faza kompajliranja koda nakon tokenizacije je parsiranje. Parsiranje je proces u kojem se uzima lista tokena, te se iz nje kreira stablo apstraktne sintakse (Abstract syntax tree, AST). AST je stablo koje predstavlja strukturu koda na način koji je neovisan o sintaksi ili gramatici u kojoj je napisan. Na taj način je jednostavnije analizirati kod u sljedećim fazama.

Stablo apstraktne sintakse je prikazano preko čvorova koji skupa tvore stablo, zvani UAST_Node. UAST_Node je klasa koja predstavlja neki dio koda, poput definicije varijable, pozivanja funkcije, grananja, specifičnih petlji i sl. Svaki čvor ima virtualnu funkciju GetChildren koja vraća njihovu djecu u tom stablu.

Na primjer, klasa UAST_If je klasa koja nasljeđuje od UAST_Node, te predstavlja grananje. Ima tri varijable, Condition, TrueBranch i FalseBranch. TrueBranch i FalseBranch su pokazivači na UAST_Node, te predstavljaju kod koji bi se trebao pokrenuti ovisno o uvjetu grananja. Treba naglasiti da je dozvoljeno da FalseBranch bude nullptr, pošto nije nužno da svako grananje ima kod koji se izvršava ukoliko uvjet nije točan.

Condition je pokazivač na instancu klase UAST_Expression. UAST_Expression je dijete od UAST_Node, te predstavlja dio koda koji vraća neku vrijednost, poput pozivanja funkcija i varijabli, operatora, i sl.

Većina ostalih čvorova su standardni načini definiranja stabla apstraktne sintakse, uz jednu iznimku: UAST_Basic. UAST_Basic je koji ima samo tri varijable: Main, tipa UAST_Main, PredefinedClasses, koja je vektor tipa UAST_ClassDefinition, te PredifinedFunctions, koja je vektor tipa UAST_FunctionDefinition. Nakon analize koda, tekst koji je korisnik napisao se pretvara u stablo apstraktne sintakse, te UAST_Main je korijen tog stabla. UAST_Main se zatim sprema u Main varijablu UAST_Basic čvora, koji postaje novi korijen. To dopušta umetanje klasa i funkcija u stablo bez izmjene koda koji je korisnik upisao, tako da se nadodaju u PredefinedClasses i PredefinedFunction varijable.

Parser je klasa koja pretvara tokene dobivene iz Lexera u AST. Te tokene sprema u varijablu, te ih koristi kako bi pronašla odgovarajući čvor u stablu apstraktne sintakse. Listu tokena ne modificira, nego čuva indeks zadnjeg elementa. Kada analizira da li je određeni UAST_Node validan za sljedeću sekvencu tokena, koristi taj indeks, te ukoliko je, ga povećava. Jednostavan primjer je funkcija ParseScope, koja je prikazana ispod.

```
UAST_Scope* UParser::ParseScope(int32& Index)
{
	int32 CurrentIndex = Index;

	// Open curly bracket
	if (!IsTokenOfClass(CurrentIndex, UT_OpenCurlyBracket::StaticClass())) return
nullptr;

	// Construct scope
	UAST_Scope* Scope = NewObject<UAST_Scope>(GetInterpreter());

	// Scope content
	while (!IsTokenOfClass(CurrentIndex, UT_ClosedCurlyBracket::StaticClass(),
true, true, false))
	{
		// Optionaly consume semicolon
		if (IsTokenOfClass(CurrentIndex, UT_Semicolon::StaticClass())) continue;

		UAST_Node* Node = ParseNode(CurrentIndex);
		if (!Node)
		{
			ThrowError(TEXT("invalid scope content."));
			return nullptr;
		}
```

```
			Scope->Children.Add(Node);
		}

		// Closed curly bracket
		if (!IsTokenOfClass(CurrentIndex, UT_ClosedCurlyBracket::StaticClass()))
		{
			ThrowError(TEXT("Missing \"}\" from scope."));
			return nullptr;
		}

		// Finish
		Index = CurrentIndex;
		return Scope;
}
```

Kod 5.1 Funkcija ParseScope u klasi UParser.

Izvor: vlastita izrada

Ova funkcija vraća UAST_Scope pokazivač ukoliko sekvenca tokena odgovara bloku koda (scope), tj. dijelu koda koji se nalazi unutar vitičastih zagrada. Ukoliko sekvenca ne odgovara, tada vraća nullptr.

Kao i u drugim funkcijama parseraprvo se sprema lokalna kopija Index varijable . Zatim se gleda da li je sljedeći token tipa UT_OpenCurlyBracket, tj otvorena vitičasta zagrada. Ukoliko nije, tada ovo nije blok koda, te se vraća nullptr. Ukoliko je, taj dio koda mora predstavljati blok koda, te se kreira UAST_Scope instanca. Dok god sljedeći token nije UT_ClosedCurlyBracket (tj. zatvorena vitičasta zagrada) traži se sljedeći čvor pozivanjem ParseNode funkcije. Ta funkcija poziva sve ostale funkcije koje analiziraju sekvencu tokena, osim definicije klase i definicije funkcija, pošto se te definicije ne mogu definirati unutar bloka koda. Ukoliko se ne pronađe niti jedan odgovarajući čvor, tada se baca greška, te se vraća nullptr. Razlog zašto se u ovom slučaju javlja greška je taj što ako prvi znak nije otvorena vitičasta zagrada, još uvijek je moguće da sekvenca odgovara nekom drugom čvoru. Međutim, ukoliko je, tada ovo mora biti blok koda, pa u slučaju da ne završava sa zatvorenom vitičastom zagradom on nije korektno definiran, te kod zasigurno sadrži grešku. Slična situacija se nalazi u ParseFunctionDefinition funkciji, koja traži definiciju funkcije. Ukoliko nakon dva identifikatora se ne nalazi otvorena zagrada, tada se ne javlja greška, pošto dva identifikatora mogu predstavljati definiciju varijable.

Svaki čvor koji se pronađe unutar tog bloka se sprema kao njegovo dijete. Na kraju, ukoliko ne pronađemo token koji predstavlja zatvorenu vitičastu zagradu javlja se greška sa porukom da blok nije zatvoren.

Važno je napomenut da se u ovoj funkciji, kao i u ostalim funkcijama parsera koje analiziraju, na kraju nalazi kod koji sprema vrijednost varijable CurrentIndex u varijablu Indeks. Razlog zašto se koristi lokalna kopija varijable je u tome da ukoliko funkcija ne vrati čvor, Indeks ostaje nepromijenjen. Na isti način kao što ova funkcija modificira indeks kada uspješno vrati čvor, isto tako i funkcija IsTokenOfClass povećava indeks koji joj se doda preko reference ukoliko token na tom indeksu je zaista traženog tipa. Kod funkcije IsTokenOfClass je prikazan u sljedećem kodu.

```
bool UParser::IsTokenOfClass(int32& Index, TSubclassOf<UToken> Class, bool
ConsumeWhitespace /*= true*/, bool ConsumeComment /*= true*/, bool AutoIncrement /*=
true*/)
{
	int32 tempIndex = Index;
	ConsumeTokens(tempIndex, ConsumeWhitespace, ConsumeComment);
	bool result = IsTokenOfClassAtIndex(tempIndex, Class);
	if (result)
	{
		tempIndex++;
		if (AutoIncrement) Index = tempIndex;
	}

	return result;
}
```

Kod 5.2 Funkcija IsTokenOfClass iz klase UParser

Izvor: vlastita izrada

Funkcija IsTokenOfClass vraća bool koji ima vrijednost istine ukoliko je token na zadanom indeksu određenog tipa, te povećava vrijednost tog indeksa. Ukoliko nije, vraća laž, te se indeks ne mijenja. Postoje i opcionalni argumenti ConsumeWhitespace, koji ukoliko je istinit će preskočit sve whitespace tokene, ConsumeComment koji će preskočit sve tokene koji predstavljaju komentar, te AutoIncrement koji će, ukoliko je istinit, automatski povećati indeks ukoliko je zadanog tipa.

```
bool UParser::IsTokenOfClassAtIndex(int32 Index, TSubclassOf<UToken> Class) const
{
	if (!GetTokenAtIndex(Index)) return false;

	return GetTokenAtIndex(Index)->GetClass()->IsChildOf(Class);
}
```

Kod 5.3 Funkcija IsTokenOfClassAtIndex iz klase UParser

Izvor: vlastita izrada

Funkcija IsTokenOfClassAtIndex vraća istinu ukoliko je token na zadanom indeksu određenog tipa. Za razliku od IsTokenOfClass funkcije, ona ne modificira indeks, te ne preskače tokene. Pošto tokeni nasljeđuju od UObject klase, koja je definirana u Unreal Engine-u 4, tada imaju pristup GetClass funkciji, preko koje se zatim može odrediti da li je taj token zadanog tipa, ili njegovo dijete.

Također je važno da se napisanim operatorima poštuje njihov prioritet i asocijativnost, kako bi pisanje formula bilo intuitivno. O tome se brine funkcija GetRPN, što je skraćenica od Get Reverse Polish Notation. Kod te funkcije je prikazan ispod.

```
TArray<UAST_Expression*> UParser::GetRPN(int32& Index)
{
	int32 CurrentIndex = Index;

	TArray<UAST_Expression*> OutputQueue = TArray<UAST_Expression*>();
	TArray<UAST_Operator*> OperatorStack = TArray<UAST_Operator*>();

	while (true)
	{
		// Unary prefix
		while (true)
		{
			UAST_OneArgument* UnaryPrefix =
GetUnaryPrefixOperator(CurrentIndex);
			if (!UnaryPrefix) break;
			AddOperator(UnaryPrefix, OutputQueue, OperatorStack);
		}

		// brackets
		if (IsTokenOfClass(CurrentIndex, UT_OpenRoundBracket::StaticClass()))
		{
			UAST_Expression* Expression = ParseExpression(CurrentIndex);
			if (!Expression)
			{
				ThrowError("Invalid expression after \"(\".");
				return TArray<UAST_Expression*>();
			}
			if (!IsTokenOfClass(CurrentIndex,
UT_ClosedRoundBracket::StaticClass()))
```

```
                        {
                                ThrowError("Missing \")\" in expression.");
                                return TArray<UAST_Expression*>();
                        }
                        OutputQueue.Add(Expression);
                }
                else
                {
                        // Operand
                        UAST_Expression* Operand = ParseOperand(CurrentIndex);
                        if (!Operand)
                        {
                                if (OutputQueue.Num() > 0 || OperatorStack.Num() > 0)
                                        ThrowError(TEXT("Invalid expression. Expected an
operand."));

                                return TArray<UAST_Expression*>();
                        }
                        OutputQueue.Add(Operand);
                }

                // Unary suffix
                while (true)
                {
                        UAST_OneArgument* UnarySufix =
GetUnarySufixOperator(CurrentIndex);
                        if (!UnarySufix) break;
                        AddOperator(UnarySufix, OutputQueue, OperatorStack);
                }


                // Binary operator
                UAST_TwoArguments* BinaryOperator = GetBinaryOperator(CurrentIndex);
                if (!BinaryOperator) break;
                AddOperator(BinaryOperator, OutputQueue, OperatorStack);
        }

        while (OperatorStack.Num() > 0)
        {
                UAST_Operator* TopOperator = OperatorStack.Last();
                OperatorStack.RemoveAt(OperatorStack.Num() - 1);
                OutputQueue.Add(TopOperator);
        }

        Index = CurrentIndex;
        return OutputQueue;

}
```

Kod 5.4 Funkcija GetRPN iz klase UParser.

Izvor: vlastita izrada

GetRPN funkcija koristi Shanting Yard algoritam kako bi konvertirala infiks izraz u postfiks izraz, koji se zatim može konvertirati u stablo apstraktne sintakse. Ona također uzima u obzir prefiks i sufiks operatore, tako što sadrži mape u kojoj su ključevi

tipovi tokena koji predstavljaju operatore, te vrijednosti su čvorovi koji predstavljaju funkcionalnost tog operatora. Na primjer, mapa UnaryPrefixOperatorMap sadrži ključ tipa UT_Minus, tj. tip tokena koji predstavlja operator minus, dok je vrijednost UAST_UnaryMinus, tj. čvor koji predstavlja prefiks minus. Isti ključ u mapi BinaryOperatorMap ima vrijednost UAST_Subtraction, tj. čvor koji predstavlja oduzimanje dva broja. Na taj način se može otkriti kontekst operatora.

6 Semantička analiza

Sljedeći korak je provjera stabla apstraktne sintakse kako bi se ustanovilo da li je kod semantički točan. To se čini uz pomoć tri klase: USemanticAnalysis, USemanticAnalysisPhase i USymbolTable.

USymbolTable je klasa koja sprema sve podatke koji su potrebni za analizu i pokretanje koda, poput mape koja određuje koja klasa nasljeđuje od neke druge klase, koje funkcije i varijable svaka klasa sadrži, itd.

USemanticAnalysis je klasa koja započinje semantičku analizu, no ona ne analizira sam kod. Sama analiza je podijeljena u više faza, te se svaka zasebno pokreće unutar AnalyseAST funkcije, koja je prikazana ispod.

```
USymbolTable* USemanticAnalysis::AnalyseAST(UAST_Node* Root, UValue* TopOwner)
{
	// Clear previous data
	Clear();
	MainSymbolTable->TopOwner = TopOwner;

	for (USemanticAnalysisPhase* currentPhase : Phases)
	{
		currentPhase->StartAnalysis(Root);
		if (HasErrors()) return MainSymbolTable;
	}

	// Iterate through limitation
	for (USemanticLimitation* currentLimitation : Limitations)
	{
		currentLimitation->Init(this);
		currentLimitation->StartAnalysis(Root);
		if (HasErrors()) return MainSymbolTable;
	}

	return MainSymbolTable;
}
```

Kod 6.1 Funkcija AnalyseAST iz klase USemanticAnalysis. Izvor: vlastita izrada

Funkcija AnalyseAST ima dva argumenta, prvi tipa UAST_Node* koji predstavlja korijen stabla apstraktne sintakse, te drugi tipa UValue koji predstavlja objekt u kojem se kod izvodi. Više o tome kasnije.

Varijabla Phases je vektor tipa USemanticAnalysisPhase, te sadrži sve faze semantičke analize. Sama analiza koda se odvija unutar tih faza. Ukoliko jedna od njih baci grešku, tada ova klasa zaustavlja kompletnu analizu.

Varijabla Limitations je vektor tipa USemanticLimitation. USemanticLimitation nasljeđuje od USemanticAnalysisPhase, te služi kao mogućnost da se dinamički u nekom trenutku prije kompajliranja koda nadodaju dodatne semantičke analize. Jedan primjer je limitacija koja baca grešku ukoliko korisnik direktno pozove više od jedne funkcije.

6.1 USemanticAnalysisPhase

USemanticAnalysisPhase je klasa koja predstavlja jednu fazu semantičke analize. Svaka faza sadrži po jednu virtualnu funkciju zvanu Analyse za svaki čvor stabla apstraktne sintakse. Javlja se problem kada je potrebno analizirati određeni čvor, npr. tipa UAST_Main, koji je spremljen u pokazivač tipa UAST_Node. Direktno pozivanjem funkcije Analyse bi se pozvala funkcija koja ima UAST_Node kao argument, pošto je pokazivač tog tipa. Kako bi se izbjeglo nepotrebno cast-anje, koristi se Double Dispatch (Gamma et al., 1994.).

Kada USemanticAnalysisPhase mora analizirati određeni čvor, tada se tom čvoru pozove funkciju Analyse, te doda ta faza dodaje kao argument. U ovom slučaju, ako se pozove funkcija Analyse na čvor tipa UAST_Main koji se nalazi u pokazivaču tipa UAST_Node, pozvat će se Analyse funkcija iz klase UAST_Main, pošto je ta funkcija virtualna.

```
void UAST_Main::Analyse(USemanticAnalysisPhase* SemanticAnalysisPhase)
{
	if (SemanticAnalysisPhase) SemanticAnalysisPhase->Analyse(this);
}
```

Kod 6.2 Funkcija Analyse iz klase UAST_Main.

Izvor: vlastita izrada

Taj čvor tada pozove funkciju Analyse iz zadane faze. Pošto se sada dodaje pokazivač this kao argument, taj pokazivač je tipa UAST_Main, te se u USemanticAnalysisPhase klase poziva Analyse funkcija koja ima UAST_Main kao argument.

Na taj način je analiza stabla apstraktne sintakse odvojena od definicije čvora, te se izbjegava nepotrebno cast-anje. Na istom principu rade sve klase koje nasljeđuju od USemanticAnalysisPhase, te će se one najvažnije sada pobliže pojasniti. Sva djeca USemanticAnalysisPhase klase imaju prefiks SAP, te analiza započinje pozivanjem virtualne funkcije StartAnalysis, koja prima korijen stabla kao argument, te ga analizira.

6.2 USAP_ClassAnalysis

USAP_ClassAnalysis je prva faza semantičke analize. Ona ne analizira kod koji se nalazi unutar klasa, nego samo da li su klase korektno definirane. Točnije, da li postoje dvije ili više klasa sa istim imenom, da li postoji cirkularna zavisnost između klasa, i sl. Analiza počinje pozivanjem funkcije StartAnalysis

```
void USAP_ClassAnalysis::StartAnalysis(UAST_Node* RootNode)
{
	Super::StartAnalysis(RootNode);

	if (HasErrors()) return;
	StoreChildParentRelationships();

	if (HasErrors()) return;
	CheckForCircularDependency();
}
```

Kod 6.3 Funkcija StartAnalysis iz klase USAP_ClassAnalysis.

Izvor: vlastita izrada

Pošto bi parser bacio grešku ukoliko se definicija klase nalazi na nedopuštenom mjestu u kodu, nije potrebno prolaziti kroz cijelo stablo, nego samo kroz one čvorove koji mogu imati definiciju klase, a to su UAST_Basic, te UAST_Main. UAST_Basic je uvijek korijen stabla apstraktne sintakse, te se analizira pozivajući Super::StartAnalysis(RootNode), gdje Super predstavlja roditelja klase u kojem se nalazi. Pošto je potrebna samo analiza definicija klasa, ostali čvorovi se ignoriraju. Kod analize definicije klase je prikazan ispod.

```
void USAP_ClassAnalysis::Analyse(UAST_ClassDefinition* ClassDefinition)
{
	// Check if class with same name has already been found
	if (GetSymbolTable()->ClassNames.Contains(ClassDefinition->Name))
	{
		ThrowError(TEXT("Duplicate class " + ClassDefinition->Name + "
Detected."));
		return;
	}

	GetSymbolTable()->ClassNames.Add(ClassDefinition->Name, ClassDefinition);
}
```

Kod 6.4 Funkcija Analyse iz klase USAP_ClassDefinition, koja analizira čvor definicije klase. Izvor: vlastita izrada

Ovaj kod se pokreće za svaku definiciju klase. ClassNames je mapa unutar USymbolTable klase koja ima naziv klase kao ključ, te definiciju klase kao vrijednost. To znači da se nakon analize svi tipovi podataka nalaze kao ključ u toj mapi. Zato prilikom analize, ukoliko se pronađe definirana klasa sa istim imenom kao i klasa koja se trenutno analizira, znači da je od prije definirana klasa sa istim imenom, te se baca greška. Ukoliko je ime jedinstveno, tada se ClassName mapi doda par imena i definicije klase.

Nakon analize svake klase se provodi analiza nad njihovom vezom, kako bi se utvrdilo da li neka klasa nasljeđuje od neke druge. Razlog zašto se to radi nakon što se prikupe sva imena klasa, umjesto tijekom je ta što ovaj način dopušta da se klase definiraju bilo kojim redoslijedom, dok bi u protivnom bilo potrebno definirati dijete nakon roditelja.

Ta veza između klasa se sprema u ClassInheritance mapu u USymbolTable, gdje je ključ dijete, te vrijednost roditelj. Ukoliko klasa nema roditelja, u vrijednost se sprema nullptr. Taj dio analize se pokreće pozivanjem StoreChildParentRelationship funkcije, koja je prikazana ispod.

```
void USAP_ClassAnalysis::StoreChildParentRelationships()
{
	// Loop through every class
	for (TMap<FString, UAST_ClassDefinition*>::TConstIterator it = 
GetSymbolTable()->ClassNames.CreateConstIterator(); it; ++it)
	{
		FString ParentName = it->Value->ParentName;

		// If this class has no parent
		if (ParentName == "")
		{
			// nullptr as parent means this is base class
			GetSymbolTable()->ClassInheritance.Add(it->Value, nullptr);
			continue;
		}

		// Check if class with ParentName exists
		UAST_ClassDefinition** parent = GetSymbolTable()-
>ClassNames.Find(ParentName);
		if (!parent)
		{
			ThrowError("Class \"" + it->Value->Name + "\" inherits from 
unknown class \"" + ParentName + "\".");
			return;
		}

		// Store child parent relationship
		GetSymbolTable()->ClassInheritance.Add(it->Value, *parent);
	}
}
```

Kod 6.5 Funkcija StoreChildParentRelationship iz klase USAP_ClassAnalysis.

Izvor: vlastita izrada

Ova funkcija iterira kroz sve klase, te za svaku traži naziv roditelja te klase. Ukoliko je naziv prazan string, tada klasa nema roditelja, te se sprema u ClassInheritance mapu u klasu kao ključ, te nullptr kao vrijednost. Ukoliko string nije prazan, pogleda se da li je naziv roditelja definiran, te se baca greška ukoliko nije. Ukoliko je, tada se sprema u ClassInheritance mapu.

Zadnji dio analize klasa je provjera da li postoji ciklična zavisnost između dvije ili više klasa, tj. da li neka klasa direktno ili indirektno nasljeđuje od neke druge, koja zatim

direktno ili indirektno nasljeđuje od ove prve. To se provjerava pozivanjem funkcije CheckForCircularDependencies, koja je prikazana ispod.

```
void USAP_ClassAnalysis::CheckForCircularDependency()
{
	// Set of all the visited classes. If a class is revisited, then there is a
circular dependency
	TSet<UAST_ClassDefinition*> VisitedClasses;

	// Loop through every class
	for (TMap<FString, UAST_ClassDefinition*>::TConstIterator it =
GetSymbolTable()->ClassNames.CreateConstIterator(); it; ++it)
	{
		UAST_ClassDefinition* currentClass = it->Value;
		TSet<UAST_ClassDefinition*> visitedChain;

		// If this class has been visited, then we already checked this
inheritance chain
		if (VisitedClasses.Contains(currentClass)) continue;

		// Go to parent
		currentClass = GetSymbolTable()->ClassInheritance[currentClass];

		// Loop through children
		while (currentClass)
		{
			// If this class has been visited, then there is a cyclic
dependency
			if (visitedChain.Contains(currentClass))
			{
				ThrowError("Circular dependency detected with class \"" +
currentClass->Name + "\".");
				return;
			}
			// mark this class as visited
			VisitedClasses.Add(currentClass);
			visitedChain.Add(currentClass);

			// Go to next class
			currentClass = GetSymbolTable()->ClassInheritance[currentClass];
		}
	}

}
```

Kod 6.6 Funkcija CheckForCircularDependency iz klase USAP_ClassAnalysis.

Izvor: vlastita izrada

Ova funkcija sadrži skup definicija klasa zvan VisitedClasses, te predstavlja sve klase koje su provjerene tijekom ovog dijela analize. Analiza započinje iteriranjem kroz svaku definiciju klase. Ukoliko se ta klasa nalazi u skupu VisitedClasses, tada smo ju

već provjerili, te proces ponavljamo sa sljedećom klasom. Ukoliko se ne nalazi, tada nastavljamo sa analizom. Iteriramo kroz lanac nasljeđivanja, te svaku klasu spremimo u skup VisitedChain i VisitedClasses. Ukoliko se jedna od tih klasa već nalazi u VisitedChain skupu, to znači da se kroz nju iteriralo dvaput, te se dakle u tom lancu nasljeđivanja nalazi ciklična zavisnost. Ukoliko se to dogodi, baca se greška, te analiza prestaje.

6.3 USAP_ConvertVariableCallsToType

Tijekom parsiranja se ne mogu prikupiti sve potrebne informacija kako bi se čvorovi stabla mogli točno definirati.. Na primjer, kod A.B može značit da pozivamo varijablu B iz instance spremljene u varijabli A. No također može značit da pozivamo statičnu varijablu B iz klase tipa A. Iako u oba slučaja je B poziv varijable, kontekst identifikatora A se mijenja. Pošto se tijekom parsiranja ne može otkriti kontekst identifikatora pošto u to vrijeme se ne znaju sva imena klasa, svi ti čvorovi su kreirani kao poziv varijable. Pošto su se u prijašnjoj fazi sakupili svi nazivi klasa, sada se može iterirati kroz stablo i zamijeniti čvorove pozivanja varijabli sa čvorovima koji predstavljaju tip tamo gdje je to prikladno. Izmjena stabla tijekom analize nije poželjna, no u ovom slučaju je neizbježna.

Ova klasa sadrži pokazivač tipa UAST_VariableNode zvan VariableCallToConvert. Važno je sačuvati pokazivač na čvor koji mijenjamo, pošto taj čvor moramo zamijeniti sa novim u njegovom roditelju. Stoga nije dosta analizirati samo UAST_VariableNode čvorove, nego sve čvorove koji bi mogli imati taj čvor kao dijete.

Također, ova klasa ima još dodatne dvije funkcije, CanConvert i CreateType. CanConvert je jednostavna pomoćna funkcija koja vraća istinu ukoliko VariableCallToConvert nije nullptr. Funkcija CreateType je također jednostavna.

```
UAST_Type* USAP_ConvertVariableCallsToType::CreateType(UAST_Node* Parent)
{
	UAST_Type* Type = NewObject<UAST_Type>(Parent->GetOuter());
	Type->Type = VariableCallToConvert->VariableName;
	VariableCallToConvert = nullptr;
	return Type;
}
```

Kod 6.7 Funkcija CreateType iz klase USAP_ConvertVariableCallsToType.

Izvor: vlastita izrada

Funkcija uzima čvor stabla apstraktne sintakse kao argument, te kreira novi čvor tipa UAST_Type, te vrača vrijednost VariableCallToConvert na nullptr. Sama analiza čvora poziva varijable je prikazana u sljedećem kodu:

```
void USAP_ConvertVariableCallsToType::Analyse(UAST_VariableCall* VariableCall)
{
	if (!GetSymbolTable()->ClassNames.Contains(VariableCall->VariableName)) return;

	// Mark variable call for conversion
	VariableCallToConvert = VariableCall;
}
```

Kod 6.8 Funkcija Analyse iz klase USAP_ConvertVariableCallsToType, koja analizira čvor tipa USAT_VariableCall. Izvor: vlastita izrada

Kod je vrlo jednostavan. Ukoliko je naziv varijable jednak nazivu jedne od klasa, tada znamo da se ne radi o varijabli nego o tipu podataka. Stoga vrijednost varijable VariableCallToConvert mijenjamo u taj čvor. Sama zamjena čvora se dešava u njegovom roditelju, što je također jednostavno, pošto iteriramo kroz stablo tako da prvo ulazimo u dubinu. Za primjer je odabran čvor UAST_Scope, koji predstavlja blok koda unutar vitičastih zagrada.

```
void USAP_ConvertVariableCallsToType::Analyse(UAST_Scope* Scope)
{
	for (int i = 0; i < Scope->Children.Num(); ++i)
	{
		Scope->Children[i]->Analyse(this);
		if (CanConvert()) Scope->Children[i] = CreateType(Scope);
	}
}
```

Kod 6.9 Funkcija Analyse iz klase USAP_ConvertVariableCallsToType, koja analizira čvor tipa USAT_Scope. Izvor: vlastita izrada

Funkcija iterira kroz svako dijete, te ga analizira. Nakon analize djeteta se poziva funkcija CanCovert, te se preko nje saznaje da li je potrebno zamijeniti taj čvor. Ukoliko je to potrebno, to dijete zamijenimo novim čvorom koji kreiramo u funkciji CreateType. Pošto čvor poziva varijable nema djece, nakon njene analize se egzekucija koda mora vratiti u njegovog roditelja, te VarijableCallToConvert može označavat samo zadnje analizirani čvor. Sličan kod se nalazi u analizi svih ostalih čvorova.

6.4 USAP_FunctionAnalysis

Ova faza analize provjerava da li su funkcije pravilno definirane. To uključuje provjeru tipova podataka za argumente i vraćene vrijednosti, provjera da ne postoje dvije iste funkcije unutar jedne klase, provjera da je tip podataka koji nadjačana (override-ana) funkcija vraća je isti kao i onaj iz funkcije koju nadjačava, ili da je taj tip podataka dijete originalne funkcije, provjerava da li svi argumenti u funkciji imaju različite nazive, te provjerava da li su svi konstruktori definirani unutar klase.

Ova klasa također sadrži i pokazivač tipa UAST_ClassDefinition zvan owningClass. Ta varijabla je potrebna kako bi se zapamtilo u kojoj klasi se nalazi funkcija koja se analizira dok kod prolazi kroz stablo apstraktne sintakse. Vrijednost te varijable se sprema tijekom analize klasa, kao što je prikazano u sljedećoj funkciji.

```
void USAP_FunctionAnalysis::Analyse(UAST_ClassDefinition* ClassDefinition)
{
	// Store this class as parent
	owningClass = ClassDefinition;

	// Loop through all function definitions
```

```
	for (UAST_FunctionDefinition* functionDefinition : ClassDefinition-
>FunctionDefinitions)
	{
		functionDefinition->Analyse(this);
	}

	// Remove class as parent
	owningClass = nullptr;
}
```

Kod 6.10 Funkcija Analyse iz klase USAP_FunctionAnalysis, koja analizira čvor UAST_ClassDefinition. Izvor: vlastita izrada

Svaka klasa koja se analizira se sprema u owningClass varijablu, te kada analiza te klase završi, vrijednost owningClass varijable se vraća na nullptr. Na taj način se može otkriti u kojoj klasi se nalazi funkcija koja se analizira. Sama analiza funkcije je prikazana u sljedećem kodu.

```
void USAP_FunctionAnalysis::Analyse(UAST_FunctionDefinition* FunctionDefinition)
{
	// Check if it is a constructor
	if (FunctionDefinition->FunctionData.FunctionName == "constructor")
	{
		if (!owningClass)
		{
			ThrowError("Constructors must be defined inside a class.");
			return;
		}

		// Store constructor
		FunctionDefinition->FunctionData.FunctionName = owningClass->Name;
		FunctionDefinition->FunctionData.ReturnType = owningClass->Name;
		GetSymbolTable()-
>Constructors.FindOrAdd(owningClass).constructors.Add(FunctionDefinition);
	}

	CheckTypeValidity(FunctionDefinition, owningClass);
	if (HasErrors()) return;

	CheckDuplicateFunctions(FunctionDefinition, owningClass);
	if (HasErrors()) return;

	CheckDuplicateArgumentName(FunctionDefinition, owningClass);
	if (HasErrors()) return;

	// If there is no owning class, this function exists inside global scope,
therefore it cannot be overriden
	if (owningClass) CheckOverrideValidity(FunctionDefinition, owningClass);


	// Add Function id to symbol table
	FFunctionSignature sig = FunctionDefinition->FunctionData.GetSignature();
```

```
	GetSymbolTable()->FunctionIDs.Add(FFunctionID(sig, owningClass),
FunctionDefinition);

}
```

Kod 6.11 Funkcija Analyse iz klase USAP_FunctionAnalysis, koja analizira čvor tipa USAT_FunctionDefinition. Izvor: vlastita izrada

Prvi korak analize je provjera da li je funkcija konstruktor. Pošto se konstruktor u klasi definira tako da se za ime funkcije upiše ključna riječ „constructor“ bez povratnog tipa, te je podatke potrebno nadopuniti. Stoga se naziv funkcije te njen povratni tip mijenjaju u naziv klase u kojoj se nalazi. Ukoliko se ne nalazi unutar klase, baca se greška pošto konstruktor definiran van klase nije validan. Zatim se taj konstruktor sprema u mapu Constructors u SymbolTable-u kako bi se kasnije lakše dohvatila.

Zatim se pregledava da li su svi argumenti validnog tipa, te da li je povratna vrijednost važeća, kao što je prikazano na sljedećem kodu.

```
void USAP_FunctionAnalysis::CheckTypeValidity(UAST_FunctionDefinition*
FunctionDefinition, UAST_ClassDefinition* ownerClass)
{
	// Check parameter types
	for (FParameterData paramData : FunctionDefinition->FunctionData.ParameterData)
	{
		if (!GetSymbolTable()->ClassNames.Contains(paramData.Type))
		{
			ThrowError("Undefined type \"" + paramData.Type + "\" detected in
function \"" + FunctionDefinition->FunctionData.FunctionName + "\".");
		}
	}

	// Check return value
	FString returnType = FunctionDefinition->FunctionData.ReturnType;

	if (!GetSymbolTable()->ClassNames.Contains(returnType) && returnType != "void")
	{
		ThrowError("Undefined return type \"" + returnType + "\" detected in
function \"" + FunctionDefinition->FunctionData.FunctionName + "\".");
	}
}
```

Kod 6.12 Funkcija CheckTypeValidity iz klase USAP_FunctionAnalysis.

Izvor: vlastita izrada

Kod je vrlo jednostavan. Iterira se kroz svaki argument funkcije, te se provjerava da li se naziv tipa tog argumenta nalazi u skupu naziva svih klasa. Ukoliko se nalazi, tada je taj tip podatka definiran, u protivnom nije i baca se greška. Isto se čini i sa tipom povratne vrijednosti.

Nakon provjere valjanosti tipova podataka u funkciji poziva se funkcija CheckDuplicateFunctions, koja provjerava da neka funkcija nije dvaput definirana. Kod te funkcije je prikazan ispod.

```
void USAP_FunctionAnalysis::CheckDuplicateFunctions(UAST_FunctionDefinition*
FunctionDefinition, UAST_ClassDefinition* ownerClass)
{
	// Store signature
	FFunctionSignature sig = FunctionDefinition->FunctionData.GetSignature();

	// if parent class is nullptr, then this class is defined in global scope
	FFunctionID ID = FFunctionID(sig, ownerClass);

	// Throw error if this function is a duplicate
	if (GetSymbolTable()->FunctionIDs.Contains(ID))
	{
		FString location = ownerClass ? "class \"" + ownerClass->Name + "\"" :
"global scope";
		ThrowError("Duplicate function \"" + sig.FunctionName + "\" detected in
" + location + ".");
		return;
	}
}
```

Kod 6.13 Funkcija CheckDuplicateFunctions iz klase USAP_FunctionAnalysis.

Izvor: vlastita izrada

U ovoj funkciji se po prvi puta pojavljuju dvije važne strukture, FFunctionSignature i FFunctionID. FFunctionSignature sadrži naziv funkcije, te sve podatke o argumentima, dok FFunctionId sadrži FFunctionSignature i pokazivač klase u kojoj se ta funkcija nalazi. Na taj način se funkcije mogu razlikovati iako bi mogle imati isti potpis. Stoga funkcija potraži da li je Id trenutne funkcije već pohranjen u SymbolTable, te ako je, to znači da je već definirana jedna druga funkcija u istoj klasi sa istim potpisom, te se stoga baca greška.

Sljedeći dio analize funkcije je provjera da li argumenti imaju različita imena, kao što je prikazano u sljedećoj funkciji.

```
void USAP_FunctionAnalysis::CheckDuplicateArgumentName(UAST_FunctionDefinition*
FunctionDefinition, UAST_ClassDefinition* ownerClass)
{
	TSet<FString> ArgumentNames;
	for (FParameterData paramData : FunctionDefinition->FunctionData.ParameterData)
	{
		if (ArgumentNames.Contains(paramData.Name))
		{
			FString location = ownerClass ? "class \"" + ownerClass->Name +
"\"" : "global scope";
			ThrowError("Function \"" + FunctionDefinition-
>FunctionData.FunctionName + "\" in " + location
				+ " has multiple arguments with same name: " +
paramData.Name);
		}
		else ArgumentNames.Add(paramData.Name);
	}
}
```

Kod 6.14 Funkcija CheckDuplicateArgumentName iz klase USAP_FunctionAnalysis.

Izvor: vlastita izrada

Funkcija sadrži skup string-ova koji predstavljaju imena argumenata. Iterirajući kroz svaki argument funkcije, provjerava se da li se naziv trenutnog argumenta nalazi unutar skupa. Ukoliko se nalazi, znači da postoje dva ili više argumenta sa istim nazivom, te se baca greška. Ukoliko se ne nalazi, dodaje se u skup i pregledava se sljedeći argument. Ukoliko funkcija iterira kroz sve argumente, te se ne pojavljuje greška, tada svi argumenti imaju jedinstvena imena.

Zadnji korak ove analize je provjera da li trenutna funkcija nadjačava neku drugu funkciju, i ako da, da li to radi na pravilan način. Traži se funkcija koja ima isti potpis kao i funkcija koja se trenutno analizira, ali da pripada klasi od koje nasljeđuje klasa trenutne funkcije. Ukoliko se ta funkcija pronađe, tada je dovoljno provjeriti tipove povratnih vrijednost, te ukoliko su jednaki, ili povratni tip nasljeđuje od povratnog tipa iz funkcije roditelja, tada je nadjačavanje validno. U protivnom se baca greška.

Ukoliko niti jedan dio analize na baci grešku, tada se potpis funkcije i klasa kojoj pripada spremaju kao FFunctionID struktura u SymbolTable.

6.5 USAP_OperatorAnalysis

USAP_OperatorAnalysis je faza semantičke analize koja analizira da li su operatori korektno preopterećeni (Overload-ani). Preopterećivanje operatora koristi slična pravila kao i C# jezik, točnije, zadane funkcije moraju biti statične, te definirane u jednoj od klasa koje operator koristi kao argument. Također kao ime moraju imati riječ „operator“ nakon koje slijedi sam operator. Također je potrebno naglasiti da neki operatori moraju vratiti bool vrijednost, poput operatora usporedbe.

Kao bi znala koji operatori moraju vračati bool vrijednost, klasa sadrži dvije mape: UnaryOperatorReturns i BinaryOperatorReturns. Obje mape sadrže string kao ključ, koji predstavlja naziv operatora, te bool vrijednost koja je istinita ukoliko taj operator mora vratiti bool vrijednost, ili laž u protivnom.

Pošto je USAP_FunctionAnalysis faza već spremila sve funkcije u SymbolTable, nije potrebno iterirati kroz cijelo stablo apstraktne sintakse, nego je dovoljno iterirati kroz listu funkcija. Za svaku funkciju pregledavamo da li joj naziv započinje sa riječi operator, te ukoliko je nastavljamo sa analizom.

Prvi dio analize je provjera argumenata. Ovisno o broju argumenata funkcije, tražimo da li se naziv operatora nalazi u odgovarajućoj mapi. Ukoliko se ne nalazi, tada ova funkcija pokušava preopteretit operator koji se ne može preopteretit, te bacamo grešku. Ukoliko se naziv operatora pronađe, tada spremamo bool vrijednost za kasniju analizu. U slučaju da funkcija nema argumente, ili ih ima više od dva, također bacamo grešku i završavamo sa analizom.

Sljedeći dio analize je provjera da li je funkcija korektno definirana, kao što je prikazano na sljedećem isječku koda.

```
// Check return type
if (ForcedBoolReturn && currentPair.Value->FunctionData.ReturnType != "bool")
{
	ThrowError("Functions that overload operator \"" + operatorName + "\" must
return a bool.");
	return;
}

// Is function declared inside a proper class (skip array)
if (currentFunctionID.ClassDefinition->Name != "array" &&
!IsInProperClass(currentFunctionID))
{
```

```
		ThrowError("When overloading operators, function must be declared inside a
class that is used as an argument.");
		return;
	}

	// Is function static
	if (!currentPair.Value->FunctionData.IsStatic)
	{
		ThrowError("Functions that overload operators must be static.");
		return;
	}
```

Kod 6.15 Isječak koda iz klase USAP_OperatorAnalysis koji provjerava broj argumenata funkcije. Izvor: vlastita izrada

Varijabla ForcedBoolReturn vraća istinu ukoliko operator mora vratiti bool vrijednost. Te ukoliko mora, a funkcija ne vraća bool, baca se greška. Nakon toga se pregledava da li se definicija funkcije nalazi u klasi od jednog od argumenata, te ukoliko ne, ponovo se baca greška. To se čini pozivanjem IsInProperClass funkcije koja vraća istinu ukoliko je barem jedan tip argumenata jednak nazivu klase u kojoj se funkcija nalazi, pošto nije dozvoljeno definirati preopterećenje operatora van klase jednog od argumenata. Na kraju, pregledamo da li je funkcija označena kao statična funkcija, te se opet baca greška ukoliko nije.

Zadnji dio ove analize je provjera duplikata, tj. da li je isti funkcija koja je preopterećena definirana u dvije različite klase. Pošto preopterećivanje operatora može biti definirano isključivo u klasi jednog od argumenata, te operatori mogu imati najviše dva argumenta, dovoljno je provjeriti operatore sa dva argumenta, kao što je prikazano na sljedećem isječku koda.

```
	// Check for duplicates
	FFunctionSignature overloadSignature = currentFunctionID.Signature;

	// Only overloads with two arguments can be duplicates
	if (overloadSignature.ParameterSignatures.Num() == 2)
	{
		FFunctionID* funID = GetSymbolTable()->Operators.Find(overloadSignature);
		if (funID)
		{
			ThrowError("Operator overload for operator \"" + operatorName + "\" with
arguments of types " + overloadSignature.ParameterSignatures[0].Type + " and " +
overloadSignature.ParameterSignatures[1].Type + " defined in both classes.");
			return;
		}
	}
```

```
// Store
GetSymbolTable()->Operators.Add(overloadSignature, currentPair.Key);
```

Kod 6.16 Isječak koda iiz USAP_OperatorAnalysis klase koji provjerava da li je preopterećenje operatora definirano dvaput. Izvor: vlastita izrada

Ovaj dio analize je relativno jednostavan. Ukoliko SymbolTable već sadrži spremljeno preopterećenje operatora, tada znači da je trenutna funkcija koja se analizira duplikat jedne prijašnje, te se baca greška. U protivnom se sprema u SymbolTable za lakše dohvaćanje.

6.6 USAP_StoreStatics

USAP_StoreStatics je jednostavna faza semantičke analize koja iterira kroz stablo te provjerava da li postoje lokalne varijable koje su označene kao statične. Također sve globalne funkcije i varijable označava kao statične. To omogućava da se globalne funkcije i varijable pozivaju unutar drugih funkcija bez potrebe da korisnik označi da su statične.

Pošto je struktura stabla važna tijekom ove analize, potrebno je spremiti trenutnu lokaciju čvorova u stablu tijekom analize. To se čini pomoću dvije varijable: pokazivača tipa UAST_ClassDefinition zvan CurrentClass koji predstavlja klasu u kojoj se nalazi čvor koji trenutno analiziramo, te enumeraciju tipa ECurrentScope zvanu CurrentScope, koja predstavlja trenutnu lokaciju na stablu. ECurrentScope enumeracija može imati jednu od tri vrijednosti: InClass, koja označuje da se analizirani čvor nalazi unutar klase, koja je spremljena u CurrentClass varijabli, Global, koja označuje da se čvor nalazi u globalnom bloku koda, te Local, koji označuje da se čvor nalazi unutar lokalnog bloka, poput funkcije ili petlje. Sljedeći kod pokazuje iteriranje kroz djecu čvora UAST_Main.

```
void USAP_StoreStatics::Analyse(UAST_Main* Main)
{
	for (UAST_Node* currentChild : Main->GetChildren())
	{
		CurrentScope = ECurrentScope::Global;
		currentChild->Analyse(this);
	}
}
```

Kod 6.17 Funkcija Analyse iz klase USAP_StoreStatics, koja analizira čvor tipa UAST_Main. Izvor: vlastita izrada

Pošto UAST_Main čvor predstavlja globalni blok, potrebno je naznačiti varijablom CurrentScope da se dijete nalazi u globalnom bloku. Razlog zašto se vrijednost te varijable postavlja tijekom svake iteracije je ta što i djeca mogu promijeniti njenu vrijednost. Većina ostalih čvorova ima vrlo sličan kod, uz iznimku što mijenjaju vrijednost CurrentScope varijable u Local. Potrebno je i naglasiti da čvorovi koji ne mogu imati definiciju funkcije i varijable kao dijete se preskaču, pošto nisu potrebni. Važne iznimke su analize čvora UAST_ClassDefinition, gdje osim što mijenja vrijednost CurrentScope varijable u InClass, također sprema sebe u CurrentClass varijablu, te analize za čvorove UAST_FunctionDefinition i UAST_VariableDefinition. Kod analize za čvor UAST_FunctionDefinition je prikazan u sljedećem kodu.

```
void USAP_StoreStatics::Analyse(UAST_FunctionDefinition* FunctionDefinition)
{
	switch (CurrentScope)
	{
		// Function is defined in global scope
	case ECurrentScope::Global:
		FunctionDefinition->FunctionData.IsStatic = true;
		AddStaticFunction(FunctionDefinition, "");
		break;

		// Function exists in currentClass
	case ECurrentScope::InClass:
		if (FunctionDefinition->FunctionData.IsStatic)
			AddStaticFunction(FunctionDefinition, CurrentClass->Name);
		break;

		// Function can only be declared inside a class or global scope
	case ECurrentScope::Local:
		break;

	default:
		break;
	}
```

```
	CurrentScope = ECurrentScope::Local;

	// Scope can be nullptr if the function is predefined
	if (FunctionDefinition->Scope) FunctionDefinition->Scope->Analyse(this);
}
```

Kod 6.18 Funkcija Analyse iz klase USAP_StoreStatics, koja analizira čvor tipa UAST_FunctionDefinition. Izvor: vlastita izrada

Tijekom analize je važno u kojem bloku koda se nalazi definicija funkcije, što se može otkrit koristeći vrijednost ECurrentScope varijable. Ukoliko se funkcija nalazi u globalnom bloku, tada se označuje kao statična, te se sprema u SymbolTable, koristeći pomoćnu AddStaticFunction funkciju. Prvi argument AddStaticFunction funkcije je sama definicija te funkcije, dok je drugi naziv klase u kojoj se nalazi. Pošto se funkcija nalazi u globalnom bloku, tada naziv klase ostaje prazan. Ukoliko se funkcija nalazi unutar klase, provjerava se da li je statična. Ukoliko je se sprema u SymbolTable. Nakon toga se označuje CurrentScope kao Local, te se analizira tijelo funkcije.

Sličan kod se koristi za analizu definicije varijabli, kao što je prikazano ispod.

```
void USAP_StoreStatics::Analyse(UAST_VariableDefinition* VariableDefinition)
{
	switch (CurrentScope)
	{
		// Variable is defined in global scope
	case ECurrentScope::Global:
		VariableDefinition->IsStatic = true;
		AddStaticVariable(VariableDefinition, "");
		break;

		// Variable is defined in class
	case ECurrentScope::InClass:
		if (VariableDefinition->IsStatic)
			AddStaticVariable(VariableDefinition, CurrentClass->Name);
		break;

		// Variable is defined locally, therefore it cannot be static
	case ECurrentScope::Local:
		if (VariableDefinition->IsStatic)
		{
			ThrowError("Variable " + VariableDefinition->Name + " is local
but is defined as static.");
		}
		break;

	default:
		break;
```

```
        }
}
```

Kod 6.19 Funkcjia Analyse iz klase USAP_StoreStatics, koja analizira čvor tipa UAST_VariableDefinition. Izvor: vlastita izrada

Ukoliko je varijabla definirana u globalnom bloku, tada se označuje kao statična, te se sprema u SymbolTable koristeći pomoćnu funkciju AddStaticVariable, koja se ponaša slično kao i AddStaticFunction funkcija. Ukoliko se varijabla nalazi unutar klase, tada se provjerava da li je statična, te ukoliko je se također sprema u SymbolTable. Ukoliko je varijabla lokalna i statična, tada se baca greška.

Potrebno je napomenut da je nakon ove analize još uvijek moguće da statična funkcija poziva varijable iz klase u kojoj se nalazi, što nije korektno. Taj problem, i slični drugi će se kasnije riješit u jednoj drugoj fazi.

6.7 USAP_ExpressionAnalysis

USAP_ExpressionAnalysis je najkompleksnija faza semantičke analize koda. Njena glavna svrha je da analizira stablo apstraktne sintakse, te da u SymbolTable spremi podatke o povratnoj vrijednosti svakog čvora koji može vratit vrijednost. Kako bi se to postiglo potrebno je analizirati cijelo stablo u dubinu, te od listova dobiti prve tipove povratnih vrijednosti, te se od njih postepeno vračati prema korijenu.

Jedan od mnogih problema koji se pojavljuju su mogući pozivi varijabli ili funkcija koji su definirani van bloka (out-of-scope), pozivanje ne statičnih varijabli i funkcija iz tipova, i sl. Te dodatne analize povećavaju kompleksnost ove faze, zbog čega prikaz cijelog koda postaje nepraktičan, te će se prikazati samo pojednostavljeni isječci. Kako bi se analiza mogla pojasnit, potrebno je prvo objasnit određene pomoćne strukture i enumeracije.

6.7.1 EReturnMode

Prva pomoćna enumeracija je EReturnMode, koja može imati 4 vrijednosti. Invalid, koji predstavlja nevažeće vraćanje korišteno za provjeru grešaka, Type, vrijednost koja označava da čvor vrača tip podatka, Value, vrijednost koja označava da čvor vrača vrijednost, te Array, vrijednost koja označava da čvor vraća polje.

6.7.2 FExpressionReturn

Sljedeća pomoćna struktura je ExpressionReturn. Ona sadrži samo dvije varijable, ReturnMode, enumeraciju tipa EReturnMode, te string ReturnType, koji predstavlja naziv tipa. Na primjer, ukoliko neki čvor vraća tip integer-a, tada bi ReturnType bio int, a ReturnMode bi imao vrijednost Type. Ukoliko čvor vraća vrijednost tipa float, ReturnType bi bio float, a ReturnMode bi imao vrijednost Value. Ukoliko čvor vraća polje tipa bool, ReturnType bi bio bool, dok bi ReturnMode imao vrijednost Array. U slučaju greške, ReturnMode bi bio Invalid, i returnType bi se ignorirao.

FExpressionReturn ima nekoliko pomoćnih funkcija koje služe kako bi se brzo ustanovilo koji način vračanja struktura predstavlja. To su funkcije ReturnsValue, koja vraća istinu ukoliko je ReturnMode Value, ReturnsType, koja vraća istinu ukoliko je ReturnMode Type, ReturnsArray, koji vraća istinu ukoliko je ReturnMode Array, te IsClear, koji vraća istinu ukoliko je ReturnMode Invalid. Još jedna korisna funkcija koja se koristi u bacanju grešaka AsString je prikazana u kodu ispod.

```
FString AsString() const
{
	FString result = ReturnType;
	switch (ReturnMode)
	{
	case EReturnMode::Type:
		result += " type";
		break;
	case EReturnMode::Value:
		result += " value";
		break;
	case EReturnMode::Array:
		result += " array";
		break;
	default:
		break;
	}
	return result;
}
```

Kod 6.20 Funkcija AsString iz strukture FExpressionReturn.

Izvor: vlastita izrada

6.7.3 FExpressionPath

Sljedeća pomoćna struktura je FExpressionPath. Ona predstavlja definicije funkcije i varijabli unutar jednog čvora. Sadrži samo tri varijable: pokazivač tipa UAST_Node zvan CurrentStep, koji predstavlja čvor, skup pokazivača tipa UAST_FunctionDefinition zvan Functions, koji predstavlja definicije funkcije u tom čvoru, te mapu zvanu Variables, koja sadrži string kao ključ, koji predstavlja ime varijable, te FExpressionReturns kao vrijednost, koja sadrži podatke o povratnoj vrijednost te varijable.

Sama USAP_ExpressionAnalysis faza sadrži dvije varijable: Vektor tipa FExpressionPath, koji predstavlja trenutni put kroz stablo tijekom analize, te sve funkcije i varijable definirane u njemu. Druga varijabla je tipa FExpressionReturn zvana Caller. Ona predstavlja podatke koji se koriste prilikom pozivanja funkcija i varijabli. Na primjer, ukoliko neka instanca klase pozove neku funkciju, koristeći tu varijablu može se otkriti koja klasa ju je pozvala, te samim time odabrati pravu funkciju za analizu.

Analizom stabla u dubinu dodajemo svaki čvor u Path vektor pozivajući funkciju AddStepTopath, te nakon analize djece se poziva funkcija RemoveLastStep, kako bi se taj čvor maknuo sa polja. U tom slučaju vektor Path se ponaša kao stog. Primjer analize čvora grananja je prikazan u sljedećem kodu.

```
void USAP_ExpressionAnalysis::Analyse(UAST_If* If)
{
	// Add step
	AddStepToPath(If);

	// Analyse condition
	If->Condition->Analyse(this);

	// Check if condition returns a bool
	FExpressionReturn* conditionData = GetSymbolTable()->ExpressionReturns.Find(If-
>Condition);
	if (!conditionData) return;

	if (!(GetSymbolTable()->CanConvertTypes(conditionData->ReturnType, "bool") &&
conditionData->ReturnsValue()))
	{
		ThrowError("Expected a bool value in if condition, found " +
conditionData->AsString() + " instead.");
	}

	// Analyse true branch
	If->TrueBranch->Analyse(this);
```

```
	// Analyse false branch, if it exists
	if (If->FalseBranch)
	{
		If->FalseBranch->Analyse(this);
	}

	// Remove from path
	RemoveLastStep();
}
```

Kod 6.21 Funkcija Analyse iz klase USAP_ExpressionAnalysis, koja analizira čvor tipa UAST_If. Izvor: vlastita izrada

Kao analiza svakog drugog čvora u ovoj fazi, započinje se pozivanjem funkcije AddStepToPath, kako bi se taj čvor dodao na trenutni put stabla. Nakon toga se analizira dijete u varijabli Condition, koje predstavlja uvjet grananja. Nakon toga se dohvaćaju podatci o povratnoj vrijednosti, te ukoliko ona nije vrijednost ili se ne može konvertirati u vrijednost tipa bool, baca se greška. Provjera da li se neki tip podataka može implicitno konvertirati u neki drugi se dešava u pomoćnoj funkciji unutar SymbolTable-a. Nakon toga se analizira čvor koji bi se pokrenuo ukoliko je uvjet istinit (TrueBranch), te čvor koji predstavlja kod koji bi se pokrenuo ukoliko uvjet nije istinit, ako on postoji (FalseBranch). Nakon toga se čvor grananja miče sa puta pozivajući funkciju RemoveLastStep.

Analiza djece se nastavlja dok se ne dođe do lista stabla, koji je najčešće poziv varijable ili UAST_Literal, tj. čvor koji predstavlja vrijednost upisanu sa strane korisnika, poput 3.14 ili true. Ukoliko se radi o upisanoj vrijednosti, analiza je vrlo jednostavna, kao što je prikazano u sljedećem kodu.

```
void USAP_ExpressionAnalysis::Analyse(UAST_FloatLiteral* FloatLiteral)
{
	GetSymbolTable()->ExpressionReturns.Add(FloatLiteral,
FExpressionReturn("float", EReturnMode::Value));
}
```

Kod 6.22 Funkcija Analyse iz klase USAP_ExpressionAnalysis, koja analizira čvor tipa UAST_FloatLiteral. Izvor: vlastita izrada

Funkcija jednostavno dodaje taj čvor u ExpressionReturns mapu unutar SymbolTable-a. ExpressionReturns je mapa koja sadrži ključ koji predstavlja čvor, dok je vrijednost struktura tipa FExpressionReturn. Mapa predstavlja podatke o povratnoj vrijednosti određenog čvora. U ovom slučaju se radi o UAST_FloatLiteral, koji je čvor koji predstavlja decimalni broj, te se on dodaje kao ključ, te kao vrijednost se dodaje FExpressionReturn koji ima „float“ string kao ReturnType i Value kao ReturnMode.

Analiza poziva varijable je nešto kompleksnija, pošto treba uzeti u obzir da se može pozivati varijabla iz određene instance klase, ili iz samog tipa podatka ukoliko se radi o statičnoj varijabli. Kako bi se to razumjelo, potrebno je prvo pokazati kod analize UAST_MemberAccess čvora, koji predstavlja operator pristupa članovima. Taj kod je prikazan ispod.

```
void USAP_ExpressionAnalysis::Analyse(UAST_MemberAccess* MemberAccess)
{
	// Get caller
	MemberAccess->lhs->Analyse(this);

	// Store caller type for call
	FExpressionReturn* ExpReturn = GetSymbolTable()-
>ExpressionReturns.Find(MemberAccess->lhs);
	if (!ExpReturn) return;
	Caller = *ExpReturn;

	// Analyse call
	MemberAccess->rhs->Analyse(this);

	// Reset caller type
	Caller.Clear();

	// Store
	FExpressionReturn* rhsReturn = GetSymbolTable()-
>ExpressionReturns.Find(MemberAccess->rhs);
	if (!rhsReturn) return;
	GetSymbolTable()->ExpressionReturns.Add(MemberAccess, *rhsReturn);
}
```

Kod 6.23 Funkcija Analyse iz klase USAP_ExpressionAnalysis, koja analizira čvor tipa UAST_MemberAccess. Izvor: vlastita izrada

Analiza počinje analizom lijevog operatora, koji je u ovom slučaju pozivatelj. Podatci o povratnoj vrijednosti se zatim spremaju u Caller varijablu, te se analizira desni operator, koji predstavlja pozvanu varijablu ili funkciju. Tijekom te analize se koristi vrijednost Caller kako bi se otkrilo od kuda se funkcija ili varijabla poziva, te se odvija

odgovarajuća analiza ovisno o rezultatu. Nakon toga se Caller varijabla „briše“ pozivanjem funkcije Clear, te se podatci o povratnoj vrijednosti od desnog operatora spremaju kao podatci ovog čvora. Pošto je objašnjeno na koji način se sprema vrijednost Caller varijable, može se pojasniti analiza poziva varijable. U sljedećem kodu je prikazan prvi dio analize poziva varijable.

```
// Locally store caller
FExpressionReturn localCaller = Caller;
Caller.Clear();

// If caller is a type
if (localCaller.ReturnsType())
{
	FStaticVariables* statVar = GetSymbolTable()-
>StaticVariables.Find(localCaller.ReturnType);
	if (!statVar)
	{
		ThrowError("Class of type \"" + localCaller.ReturnType + "\" does not
have a static variable named \"" + VariableCall->VariableName + "\".");
		return;
	}

	for (UAST_VariableDefinition* varDef : statVar->Variables)
	{
		// If names match, the correct variable exists
		if (varDef->Name == VariableCall->VariableName)
		{
			// Link this static variable call to variable definition
			GetSymbolTable()->StaticVariableCalls.Add(VariableCall, varDef);
			EReturnMode returnMode = varDef->IsArray ? EReturnMode::Array :
EReturnMode::Value;
			GetSymbolTable()->ExpressionReturns.Add(VariableCall,
FExpressionReturn(varDef->Type, returnMode));
			return;
		}
	}

	// No matching variables
	ThrowError("Class of type \"" + localCaller.ReturnType + "\" does not have a
static variable named \"" + VariableCall->VariableName + "\".");
	return;
}
```

Kod 6.24 Isječak koda iz klase USAP_ExpressionAnalysis, koji prikazuje analizu čvora UAST_VariableCall, ukoliko je pozivatelj tip podatka.

Izvor: vlastita izrada

Prvo se vrijednost varijable Caller sprema u lokalnu varijablu LocalCaller, te se Caller invalidira kako bi se djeca mogla analizirati. Zatim se provjerava da li je pozivatelj tip

podatka. Ukoliko je, preko SymbolTable-a se dohvaćaju sve statične varijable koje pripadaju klasi tog tipa. Ukoliko se pronađe statična varijabla koja ima isti naziv kao i čvor poziva varijable koji se trenutno analizira, tada se pronašla odgovarajuća varijabla. Nadopunjuju se podatci o povratnoj vrijednosti za tu varijablu, te pošto je statična, također se povezuje sa pronađenom statičnom varijablom, kako bi se kasnije mogla brže pronaći. Ukoliko se ne pronađe statična varijabla istog imena u toj klasi, tada se baca greška i analiza završava.

Nakon ovog koda slijedi kod koji analizira čvor ukoliko se varijabla poziva bez pozivatelja. Taj kod je prikazan ispod.

```
// Implicit caller
if (localCaller.IsClear())
{
	// Move up the path until we find the definition
	for (int i = Path.Num() - 1; i >= 0; --i)
	{
		FExpressionReturn* expReturn = Path[i].Variables.Find(VariableCall-
>VariableName);
		if (!expReturn) continue;

		GetSymbolTable()->ExpressionReturns.Add(VariableCall, *expReturn);
		return;
	}

	// Check for top owner
	if (GetSymbolTable()->TopOwner)
	{
		UAST_ClassDefinition** classDefinition = GetSymbolTable()-
>ClassNames.Find(GetSymbolTable()->TopOwner->Type);
		if (!classDefinition) return;

		UAST_ClassDefinition* currentClass = *classDefinition;
		while (currentClass)
		{
			for (UAST_VariableDefinition* currentVarDefinition :
currentClass->VariableDefinitions)
			{
				if (currentVarDefinition->Name != VariableCall-
>VariableName) continue;

				// Found variable definition
				EReturnMode returnMode = currentVarDefinition->IsArray ?
EReturnMode::Array : EReturnMode::Value;
				GetSymbolTable()->ExpressionReturns.Add(VariableCall,
FExpressionReturn(currentVarDefinition->Type, returnMode));
				GetSymbolTable()->TopOwnedVariableCalls.Add(VariableCall);
				return;
			}
			// Check if variable exists in parent
			currentClass = GetSymbolTable()->ClassInheritance[currentClass];
		}
```

```
	}

	// variable undefined
	ThrowError("Unknown variable \"" + VariableCall->VariableName + "\"
detected.");
	return;
}
```

Kod 6.25 Isječak koda iz klase USAP_ExpressionAnalysis, koji prikazuje analizu čvora UAST_VariableCall, ukoliko je pozivatelj implicitan.

Izvor: vlastita izrada

Ukoliko nema direktnog pozivatelja preko operatora pristupa članova, tada on ili ne postoji, ili je pozivatelj implicitan, te ga treba pronaći. To se čini tako da se prvo prolazi kroz trenutni put u stablu prema korijenu. Prvi čvor koji sadrži definiciju varijable istog imena kao i varijabla ovog čvora je definicija te varijable. Pošto iz definicije je poznata povratna vrijednost, nije potrebno znati u kojem čvoru je ona zapravo definirano, nego je dovoljno da se spremi u SymbolTable i da analiza završi.

Ukoliko se definicija varijable ne nalazi unutar trenutnog puta, tada je potrebno provjeriti klasu instance iz varijable TopOwner iz SymbolTable. TopOwner je instanca klase koja izvršava napisani kod, te se sprema u SymbolTable izvana tijekom kompajliranja koda. Ona predstavlja implicitni „this“ pokazivač iz C++ jezika.

Ukoliko je TopOwner validan, tada se pronalazi njegova klasa, te od nje se prolazi lancem nasljeđivanja, gdje se za svaku klasu pregledava da li sadrži definiciju varijable koja je analizirana. Ukoliko se pronađe, tada se spremaju njeni podatci o povratnoj vrijednosti u SymbolTable i analiza završava. U protivnom se baca greška.

Zadnja mogućnost je da postoji eksplicitni pozivatelj, tj. vrijednost nekog tipa. Analiza u tom slučaju je prikazana u kodu ispod.

```
// Explicit caller
UAST_ClassDefinition** classDefinition = GetSymbolTable()-
>ClassNames.Find(localCaller.ReturnType);
if (!classDefinition)
{
	ThrowError("Unknown caller of type \"" + localCaller.ReturnType + "\" calls a
variable named \"" + VariableCall->VariableName + "\".");
	return;
}
```

```
UAST_ClassDefinition* currentClass = *classDefinition;
while (currentClass)
{
	for (UAST_VariableDefinition* currentVarDefinition : currentClass-
>VariableDefinitions)
	{
		if (currentVarDefinition->Name != VariableCall->VariableName) continue;

		// Found variable definition
		EReturnMode returnMode = currentVarDefinition->IsArray ?
EReturnMode::Array : EReturnMode::Value;
		GetSymbolTable()->ExpressionReturns.Add(VariableCall,
FExpressionReturn(currentVarDefinition->Type, returnMode));
		return;
	}
	// Check if variable exists in parent
	currentClass = GetSymbolTable()->ClassInheritance[currentClass];
}

// No variable with that name is defined
ThrowError("Undefined variable \"" + VariableCall->VariableName + "\" detected.");
```

Kod 6.26 Isječak koda iz klase USAP_ExpressionAnalysis, koji prikazuje analizu čvora UAST_VariableCall, ukoliko je pozivatelj instanca neke klase.

Izvor: vlastita izrada

U ovom slučaju je poznat tip podatka koji je pozvao varijablu, te se stoga preko SymbolTable-a dohvaća pripadajuća klasa. Ponovno se iterira kroz lanac nasljeđivanja, te u svakoj klasi se traži pripadajuća definicija varijable. Ako se pronađe, tada se spremaju čvor i podatci o povratnoj vrijednosti u SymbolTable i analiza završava. Ukoliko se ne pronađe odgovarajuća definicija, baca se greška.

Analiza pozivanja funkcije se odvija na sličan način, uz iznimku da je potrebno pronaći odgovarajuću funkciju ne samo po nazivu nego i po tipovima argumenata. U to treba uzeti u obzir i implicitno konvertiranje tipova podataka. Tu se funkcija nalazi tako da se prvo traži funkcija sa zadanim imenom i zadanim tipovima argumenata. Ukoliko se ne pronađe, potrebno je iterirati kroz sve funkcije istog naziva i istog broja argumenata, te pronaći odgovarajuće funkcije preko kojih se mogu argumenti konvertirati u tražene tipove. Važno je napomenuti da se baca greška u slučaju da postoji više od jedne takve funkcije, pošto se ne možemo znati koju je korisnik želio zvat. Kod analiziranja funkcija je relativno dug i kompleksan, te ga stoga nećemo prikazati.

6.8 USAP_StaticAnalysis

USAP_ExpressionAnalysis je faza koja je, među ostalim, provjerila da su svi pozivi varijabli validni, tj. da je varijabla definirana prije nego se pozove. Međutim, klasa može sadržavati statičnu funkciju, koja bi u tom slučaju mogla zvati ne statičnu varijablu definiranu unutar te klase. Klasa USAP_StaticAnalysis uzima to u obzir prilikom analize.

Kako bi ova faza korektno radila, potrebne su joj dvije varijable. Pokazivač tipa UAST_ClassDefinition zvan owningClass, koji predstavlja klasu iz koje je statična funkcija definirana, te skup string-ova zvan LocalVariableNames, koji predstavlja sve lokalne varijable u funkciji.

Pošto je USAP_StoreStatics faza spremila sve statične funkcije i varijable u SymbolTable, nije potrebno iterirati kroz cijelo stablo apstraktne sintakse, nego samo kroz listu svih statičnih funkcija. Iteracija je prikazana u sljedećem kodu.

```
void USAP_StaticAnalysis::StartAnalysis(UAST_Node* Root)
{
	// Since we already passed through Store statics phase,
	// we don't need to loop through the AST tree to find static functions.
	for (auto it = GetSymbolTable()->StaticFunctions.CreateConstIterator(); it;
++it)
	{
		for (UAST_FunctionDefinition* currentFunctionDefinition :
it.Value().Functions)
		{
			// Skip if constructor
			if (GetSymbolTable()-
>ClassNames.Contains(currentFunctionDefinition->FunctionData.FunctionName)) continue;

			// Add arguments as local variables
			LocalVariableNames.Empty();
			for (FParameterData params : currentFunctionDefinition-
>FunctionData.ParameterData)
			{
				LocalVariableNames.Add(params.Name);
			}

			// Get owning class if it exists
			UAST_ClassDefinition** owner = GetSymbolTable()-
>ClassNames.Find(it.Key());
			if (owner) owningClass = *owner;
			else owningClass = nullptr;

			// Analyse function body if it exists
			if (currentFunctionDefinition->Scope) currentFunctionDefinition-
>Scope->Analyse(this);
		}
	}
```

```
}
```

Kod 6.27 Funkcija StartAnalysis iz klase USAP_StaticAnalysis.

Izvor: vlastita izrada

Analiza počinje iteracijom kroz sve statične funkcije. Konstruktori su poseban slučaj, pošto se pozivaju kao statične funkcije, ali sadrže podatke instance. Stoga su označeni kao statične funkcije, ali se njihova analiza u ovoj fazi preskače. Ukoliko funkcija nije konstruktor, nastavlja se sa analizom. Prvo se spremaju imena svih argumenata u LocalVariableNames skup, pošto se mogu smatrati lokalnim varijablama. Zatim se sprema klasa u owningClass varijablu, ukoliko ta klasa postoji. Nakon toga se analizira tijelo funkcije.

Analizom svakog čvora se analiziraju i njihova djeca, uz par iznimki. Prilikom analize operatora pristupa članovima se analizira samo pozivatelj, pošto je to jedini argument koji je bitan u ovoj fazi. Također za svaku definiciju varijable se sprema njihovo ime u LocalVariableNames skup.

Vrlo bitna analiza je analiza pozivanja varijable. Tu se određuje da li su pozvane varijable statične, globalne, ili lokalne, te u protivnom bacamo grešku. Sljedeći isječak koda prikazuje prvi korak analize poziva varijable.

```
// Find static variable in class
if (owningClass)
{
	FStaticVariables* statVars = GetSymbolTable()-
>StaticVariables.Find(owningClass->Name);
	if (statVars)
	{
		for (UAST_VariableDefinition* varDef : statVars->Variables)
		{
			// Static variable definition found in class
			if (varDef->Name == VariableCall->VariableName) return;
		}
	}
}
```

Kod 6.28 Isječak koda iz klase USAP_StaticAnalysis, koji analizira da li se statična varijabla nalazi unutar klase. Izvor: vlastita izrada

Ovaj dio analize započinje provjerom da li se trenutna statična funkcija nalazi u nekoj klasi. Ukoliko se nalazi, tada se iz te klase uzimaju sve statične varijable, te se traže ona koja ima isti naziv. Ukoliko se pronađe, tada je poziv varijable validan, te njena analiza završava. Ukoliko se definicija ne pronađe, traži se u globalnom bloku, kako što je prikazano u sljedećem isječku koda.

```
// Find static variable in global scope
FStaticVariables* statVars = GetSymbolTable()->StaticVariables.Find("");
if (statVars)
{
	for (UAST_VariableDefinition* varDef : statVars->Variables)
	{
		// Static variable definition found global scope
		if (varDef->Name == VariableCall->VariableName) return;
	}
}

// Find as local variable
if (LocalVariableNames.Contains(VariableCall->VariableName)) return;
```

Kod 6.29 Isječak koda iz klase USAP_StaticAnalysis, koji analizira da li se varijabla nalazi unutar globalnog bloka ili je lokalna. Izvor: vlastita izrada

Kod je sličan kao i prijašnji dio analize, uz razliku da se uzimaju sve globalne statične varijable. Ukoliko se ne pronađe, varijabla se potraži unutar LocalVariableNames skupa. Ukoliko se naziv varijable nalazi u njemu, tada je varijabla lokalna, te se može pozvati. Ukoliko još uvijek nije pronađena, tada se potraži u TopOwner-u, kao što je prikazano u sljedećem isječku koda.

```
// Find as a variable in top owner
UAST_ClassDefinition* CurrentClass = GetSymbolTable()->ClassNames[GetSymbolTable()-
>TopOwner->Type];
while (CurrentClass)
{
	for (UAST_VariableDefinition* currentVarDef : CurrentClass-
>VariableDefinitions)
	{
		if (currentVarDef->Name == VariableCall->VariableName) return;
	}

	// Go to parent
	CurrentClass = GetSymbolTable()->ClassInheritance[CurrentClass];
}
```

```
ThrowError("Cannot call variable \"" + VariableCall->VariableName + "\" from inside a
static function \"" + currentFunction->FunctionData.FunctionName + "\".");
```

Kod 6.30 Isječak koda iz klase USAP_StaticAnalysis, koji analizira da li se varijabla nalazi u TopOwner-u. Izvor: vlastita izrada

U ovom se dijelu ponovo iterira kroz sve statične varijable, ovoga puta iz klase TopOwner-a ukoliko on postoji. Ako ne postoji, ili definicija nije pronađena, baca se grešku da se ta varijabla ne može pozvati unutar statične funkcije.

Analiza za pozivanje funkcija u drugoj statičnoj funkciji je slična, te se stoga neće prikazati.

6.9 USAP_FlowControlAnalysis

USAP_FlowControlAnalysis je faza semantičke analize koja provjerava da li su „break“ i „continue“ ključne riječi korektne, tj. da li se nalaze unutar petlji. Pošto petlje mogu sadržavati u sebi druge petlje, potrebno je držati listu svih posjećenih petlji tijekom putovanja kroz stablo. Stoga ova klasa ima vektor tipa UAST_Loop zvan Loops, koji služi kao stog. Analiza čvora za for petlju je prikazana u sljedećem kodu.

```
void USAP_FlowControlAnalysis::Analyse(UAST_For* For)
{
	Loops.Add(For);
	For->Body->Analyse(this);
	Loops.Pop();
}
```

Kod 6.31 Funkcija Analyse iz klase USAP_FlowControlAnalysis, koja analizira čvor tipa UAST_For. Izvor: vlastita izrada

Funkcija je vrlo jednostavna. Prvo se dodaje petlja u vektor, analizira se tijelo petlje, te se ona vadi iz vektora. Nije potrebno analizirati ostalu djecu petlji, poput uvjeta, pošto se „break“ i „continue“ naredbe ne mogu nalaziti u njima. Da se nalaze, semantička analiza bi u prijašnjim fazama bacila grešku.

Ostali čvorovi koji se analiziraju također analiziraju svoju djecu, osim onih čvorova koji ne mogu imati petlju kao dijete, poput operatora, uz par iznimki. Među njima su UAST_Break i UAST_Continue čvorovi koji predstavljaju „break“ i „continue“ naredbe.

Kod analize UAST_Break čvora je prikazan u sljedećem kodu.

```
void USAP_FlowControlAnalysis::Analyse(UAST_Break* Break)
{
	// Break should exist within a loop
	if (Loops.Num() <= 0)
	{
		ThrowError("Break keyword found outside a loop.");
		return;
	}

	// Add break to the closest loop
	GetSymbolTable()->Breaks.Add(Break, Loops.Top());
}
```

Kod 6.32 Funkcija Analyse iz klase USAP_FlowControlAnalysis, koja analizira čvor UAST_Break. Izvor: vlastita izrada

Funkcija je vrlo jednostavna. Ukoliko je analiza došla do UAST_Break čvora, a da nije unutar petlje, baca se grešk. U protivnom UAST_Break čvor i zadnja petlja spremaju u Breaks mapu iz SymbolTable-a. Ta mapa ima UAST_Break čvor kao ključ, te petlju kao vrijednost, te služi kako bi se brzo i jednostavno pronašlo kojoj petlji pripada taj čvor. Vrlo sličan kod se nalazi i u funkciji za analizu UAST_Continue čvora, uz iznimku da se čvor i petlja spremaju u Continues mapu. Ona ima istu svrhu kao i Breaks mapa, no ima UAST_Continue čvorove kao ključeve.

6.10 USAP_ReturnAnalysis

USAP_ReturnAnalysis je klasa koja analizira korektnost „return“ ključnih riječi. Među ostalim, u to spada i provjera da sve funkcije, osim onih koje vraćaju void, imaju return, te da svaki put unutar funkcija vodi do njih. Također provjerava da se vraća vrijednost tipa koji je definiran u funkciji, ili nekog tipa koji se može implicitno konvertirati u tip funkcije.

Ova klasa ima dvije varijable, pokazivač tipa UAST_FunctionDefinition zvan CurrentFunction, koji predstavlja funkciju koju analiziramo, te bool zvan PathHasReturn. Ova varijabla označava da li trenutni put ove analize uvijek završava sa return. Početak analize je prikazan u sljedećem kodu.

```
void USAP_ReturnAnalysis::StartAnalysis(UAST_Node* Root)
{
	for (auto pair : GetSymbolTable()->FunctionIDs)
	{
		CurrentFunction = pair.Value;
		PathHasReturn = false;

		if (!CurrentFunction->Scope) continue;
		CurrentFunction->Scope->Analyse(this);

		// Constructors should not have a return keyword.
		bool IsConstructor = false;
		FConstructors* constructors = GetSymbolTable()-
>Constructors.Find(pair.Key.ClassDefinition);
		if (constructors)
		{
			IsConstructor = constructors-
>constructors.Contains(CurrentFunction);
		}

		if (!IsConstructor && !PathHasReturn && !IsInVoidFunction())
		{
			ThrowError("Not all execution paths return a value in \"" +
CurrentFunction->FunctionData.FunctionName + "\".");
		}
	}
}
```

Kod 6.33 Funkcija StartAnalysis iz klase USAP_ReturnAnalysis.

Izvor: vlastita izrada

Na početku analize se iterira kroz sve funkcije. Zatim se označuje da trenutna funkcija ne završava uvijek sa return. Zatim se tijelo funkcije analizira. Nakon analize provjerava se da li je funkcija konstruktor. To je poseban slučaj, pošto konstruktori vračaju vrijednost, ali nemaju return u sebi. Na kraju ukoliko funkcija nije konstruktor, ne vraća void, te ne vode svi putevi do return-a, baca se greška

Analiza ostalih čvorova je jednostavna, pošto većina njih ne utječe na ovu analizu. Iznimke su petlje, pošto je moguće da se nikad neće pokrenut. Druga važna iznimka

je grananje, pošto jedna i druga grana moraju završit sa return kako bi grananje uvijek završilo sa return. Kod analize grananja je prikazan ispod.

```
void USAP_ReturnAnalysis::Analyse(UAST_If* If)
{
	bool LocalTrueHasReturn = false;
	bool LocalFalseHasReturn = false;
	PathHasReturn = false;

	If->TrueBranch->Analyse(this);
	LocalTrueHasReturn = PathHasReturn;
	// Reset return path for false branch
	PathHasReturn = false;

	// If there is no false branch, then branch does not have a return path
	if (If->FalseBranch)
	{
		If->FalseBranch->Analyse(this);
		LocalFalseHasReturn = PathHasReturn;
	}

	PathHasReturn = LocalTrueHasReturn && LocalFalseHasReturn;
}
```

Kod 6.34 Funkcija Analyse iz klase USAP_ReturnAnalysis, koja analizira čvor tipa UAST_If. Izvor: vlastita izrada

Analiza grananja počinje sa definiranjem dvije varijable tipa bool, LocalTrueHasReturn i LocalFalseHasReturn. One označavaju koje grane završavaju sa return. Nakon analize TrueBranch grane, LocalTrueHasReturn poprima vrijednost PathHasReturn varijable te se PatHasReturn varijabla postavlja na laž. Isto se čini sa LocalFalseHasReturn tijekom analize FalseBranch grane, ukoliko ona postoji. Zatim se PathHasReurn varijabla stavlja na istinu, ako i samo ako obje grane završavaju sa return. Ukoliko ne postoji FalseBranch grana, tada grananje nikad neće garantirat da završava sa return, pošto se može preskočit ako uvjet nije točan.

Zadnji dio ove analize je analiza samog UAST_Return čvora. Isječak tog koda je prikazan ispod.

```
// Mark that we reached the return
PathHasReturn = true;

// Check return type
if (IsInVoidFunction())
```

```
{
	// No return expression in void function is valid
	if (!Return->ReturnExpression) return;

	ThrowError(" A void function \"" + CurrentFunction->FunctionData.FunctionName
		+ "\" has a return statement that returns a value.");
	return;
}

// No return expression in non-void function
if (!Return->ReturnExpression)
{
	ThrowError("A non-void function \"" + CurrentFunction-
>FunctionData.FunctionName
			+ "\" has no return value.");
	return;
}
```

Kod 6.35 Isječak koda iz klase USAP_ReturnAnalysis koji provjerava da li void funkcija vraća vrijednost, ili da li funkcija ne vraća vrijednost kada bi trebala.

Izvor: vlastita izrada

Analiza počinje tako da se vrijednost PathHasReturn varijable stavlja na istinu, pošto je analiza došla do return. Nakon toga se provjerava da li funkcija u kojoj se čvor nalazi vraća void. Ukoliko vraća, a čvor ima djecu, tj. vraća vrijednost, tada se baca greška. Ukoliko funkcija ne vraća void, a čvor nema djecu, znači da ne vraća vrijednost, te se isto baca greška. Nakon toga se analizira povratna vrijednost, kao što je prikazano na sljedećem isječku koda.

```
FExpressionReturn ExpressionReturn = GetSymbolTable()->ExpressionReturns[Return-
>ReturnExpression];

// Incorrect type returned
if (!GetSymbolTable()->CanConvertTypes(ExpressionReturn.ReturnType, CurrentFunction-
>FunctionData.ReturnType))
{
	ThrowError("Function \"" + CurrentFunction->FunctionData.FunctionName
		+ "\" returns a wrong type in a return statement. Cannot convert from
\""
		+ ExpressionReturn.ReturnType +"\" to \"" + CurrentFunction-
>FunctionData.ReturnType + "\".");
}

// Returning a type instead of value
if (!ExpressionReturn.ReturnsValue())
{
	ThrowError("Returning a type in function \"" + CurrentFunction-
>FunctionData.FunctionName + "\" is not allowed.");
```

```
}
```

Kod 6.36 Isječak koda iz klase USAP_ReturnAnalysis koji provjerava da li funkcija vraća korektnu vrijednost. Izvor: vlastita izrada

Prvo se dohvaća vrijednost koju ovaj čvor vraća. Ta vrijednost je spremljena tijekom USAP_ExpressionAnalysis faze. Ukoliko se ne može konvertirati u tip koji funkcija vraća, tada se baca greška. Na kraju se još provjerava da li se vraća vrijednost ili tip podataka.

6.11 USAP_Final

USAP_Final je faza semantičke analize koja provjerava da li postoji nadjačana funkcija koja je označena kao final, te ukoliko postoji, baca grešku (funkcije označene kao final se ne smiju nadjačavat). Ova faza iterira kroz sve funkcije, te za svaku dohvaća klasu kojoj pripada. Zatim iterira kroz lanac nasljeđivanja te klase dok ne naiđe na funkciju istog potpisa. Sama analiza funkcije je prikazanu u sljedećem isječku koda.

```
while (parent)
{
	// Check if parent contains a function with the same signature
	UAST_FunctionDefinition** parentFunction = GetSymbolTable()-
>FunctionIDs.Find(FFunctionID(currentFunction->FunctionData.GetSignature(), *parent));
	if (!parentFunction)
	{
		// Go to next parent
		parent = GetSymbolTable()->ClassInheritance.Find(*parent);
		continue;
	}

	// If parent function is marked as final, then current function cannot override
it
	if ((*parentFunction)->FunctionData.IsFinal)
	{
		ThrowError("Class " + currentClass->Name + " overrides a function " +
currentFunction->FunctionData.FunctionName + " from its parent " + currentClass-
>ParentName + ", but its function is marked as final.");
	}

	// We found the parent function, so we break from the loop.
	break;
```

```
}
```

Kod 6.37 Isječak koda iz klase USAP_Final, koji provjerava da li postoji nadjačana funkcija koji je označena kao final. Izvor: vlastita izrada

Za svakog roditelja klase kojoj ta funkcija pripada se traži funkcija s istim potpisom. Ukoliko se pronađe, tada je poznato da trenutna funkcija nadjačava tu pronađenu. Te stoga se provjerava da li je ona označena kao final, te ako je se baca greška. Tada se može zaustavit petlja. Pošto se analizira svaka funkcija, nije potrebno nastaviti po lancu nasljeđivanja.

7 Pokretanje koda

Tijekom pokretanja koda važne su 4 klase. Prva je UValue, koja predstavlja instancu neke klase. Ona sadrži sve njene varijable. Druga je FStackFrame struktura. Ona sadrži podatke i varijable funkcije. Treća je UCodeRunner. To je klasa koja pokreće kod. Sadrži call stack, povratnu vrijednost posljednje pozvane funkcije ili varijable i sl. Zadnja važna klasa je URuntimeState (skraćeno RTS). Pošto se kod može izvršavati kroz dulji period vremena, te se u bilo kojem trenutku može zaustaviti i nastaviti, potrebno je sačuvati trenutno stanje koda. URuntimeState služi toj svrsi na način da se izvršavanje koda izvršava u koracima koji mogu trajati određeni period vremena, te se samo pokretanje sljedećeg koraka ne mora izvršiti odmah nakon završetka prijašnjeg.

7.1 UValue

UValue je klasa koja predstavlja vrijednost. Sadrži varijablu tipa string naziva Type koji sadrži naziv tipa ove vrijednosti, te sadrži i pokazivač tipa UVariable zvan Variables. UVariable je klasa koja predstavlja instancu spremljenu u UValue klasi, te sadrži sve varijable te instance. Razlika između UValue i UVariables je važna pošto je moguće da različite vrijednosti imaju istu instancu, kao u slučaju pokazivača (dva različita pokazivača mogu pokazivati na istu instancu).

UValue klasa sadrži i funkciju IsNullable, koja označava da li ova vrijednost ima instancu koja direktno ili indirektno nasljeđuje od klase object. Ukoliko da, slično kao i u C# jeziku, ova vrijednost će se dinamički alocirati, te će se kreirati kao pokazivač. Ukoliko ne nasljeđuje od object klase, tada će se ponašati kao da je kreirana na stogu. To je važno ukoliko je potrebno kreirati kopiju vrijednosti, kao što je prikazano u sljedećem kodu.

```
UValue* UValue::Duplicate()
{
	UValue* duplicate = NewObject<UValue>(CodeRunner);
	duplicate->Init(Type, CodeRunner);
	if (IsNullable())
		duplicate->Variables = Variables;
	else
	{
		duplicate->CreateVariables();
		for (auto it = Variables->Variables.CreateConstIterator(); it; ++it)
		{
			duplicate->Variables->Variables.Add(it.Key(), it.Value()-
>Duplicate());
		}
	}

	return duplicate;
}
```

Kod 7.1 Funkcija Duplicate iz klase UValue, koja duplicira vrijednost.

Izvor: vlastita izrada

Funkcija Duplicate u UValue klasi kreira kopiju same sebe. Nakon kreiranja i inicijaliziranja nove vrijednosti, vrši se provjera da li je ova vrijednost dinamički alocirana. Ukoliko je, tada se dupliciranoj vrijednosti prebacuje pokazivač na Variables objekt. Samim time nova kopija vrijednosti pokazuje na isti objekt. Ukoliko nije pokazivač, potrebno je kreirati novu instancu te rekurzivno duplicirati sve vrijednosti u varijabli.

7.2 FStackFrame

Kako bi se pojasnila svrha i implementacija FStackFrame strukture, prvo je potrebno spomenuti FScope strukturu. FScope struktura predstavlja blok koda, te sadrži sve

varijable koje žive u tom bloku. Njih sprema u mapu ValuesInScope, gdje je ključ naziv varijable, te vrijednost pokazivač tipa UValue.

Pošto tijekom izvršavanja funkcije se mogu koristiti samo varijable koje su statične, lokalne, ili pripadaju klasi te funkcije, potrebno je svako pozivanje funkcija odvojiti od prijašnjih varijabli. Također je potrebno izbrisati sve lokalne varijable nakon što funkcija završi. Stoga se koristi FStackFrame struktura koja izolira izvršavanje funkcije na način da sprema sve njene relevantne podatke u sebe, te se nakon izvršenja mogu lako izbrisati.

FStackFrame sadrži samo tri varijable, vektor tipa FScope, koji predstavlja ugniježđene blokove unutar funkcije. Pokazivač tipa UValue zvan Owner. On sadrži vrijednosti instance koja je pozvala tu funkciju, ukoliko ona postoji, te se može smatrati kao „this“ pokazivač iz C++ jezika. Zadnja varijabla je vektor pokazivača tipa URuntimeState. Ona služi kao stog koji sadrži sva stanja koda dok se izvršava. Samo izvršavanje koda započinje u UCodeRunner klasi.

7.3 UCodeRunner

UCodeRunner klasa pokreće korake odvijanja koda. Ona također sadrži i podatke o cijelom trenutnom stanju koda, uključujući sve URuntimeState instance. To čini koristeći vektor tipa FStackFrame zvan Stack.

Dvije najvažnije funkcije UCodeRunner klasi su RunNextRTSStep i RTSStepCompleted. U sljedećem kodu je prikazana RunNextRTSStep funkcija.

```
void UCodeRunner::RunNextRTSStep()
{
	if (CurrentCodeState != ECodeState::WaitingForNextStep) return;

	StepCounter++;

	while (HasRTS() && GetRTS()->GetIsCompleted())
	{
		PopRTS();
	}

	if (!HasRTS())
	{
		CodeCompleted();
		return;
	}
```

```
	CurrentCodeState = ECodeState::Running;
	GetRTS()->RunStep();
}
```

Kod 7.2 Funckija RunNextRTSStep iz klase UCodeRunner.

Izvor: vlastita izrada

Prvo se provjerava da li UCodeRunner očekuje pokretanje sljedećeg koraka provjeravajući vrijednost CurrentCodeState varijable. Ta varijabla je enumeracija tipe ECodeState, te može sadržavati jednu od 3 vrijednosti: NotRunning, koje označuje da kod još nije pokrenut ili je završio, WaitingForNextStep, koji označuje da se očekuje pokretanje sljedećeg koraka, te Running, koji označava da se trenutno izvršava korak.

Ukoliko UCodeRunner očekuje pokretanje sljedećeg koraka, tada se povećava brojač koraka. Zatim se dohvaća trenutni URuntimeState. On je uvijek „na vrhu“, tj. zadnji uneseni element u zadnjem unesenom FStackFrame-u. Ukoliko je gotov sa egzekucijom koda se vadi sa stoga, te ukoliko više nema URuntimeState elemenata, znači da je kod završio. U protivnom se ponovo dohvaća novi element „na vrhu“, te se pokreće izvršavanje njegovog koraka pozivanjem RunStep funkcije.

Nakon što se taj korak izvrši, URuntimeState poziva RTSStepCompleted funkciju u UCodeRunner-u, koja je prikazana ispod.

```
void UCodeRunner::RTSStepCompleted()
{
	if (Aborting)
	{
		CurrentCodeState = ECodeState::NotRunning;
		Aborting = false;
		Stack.Empty();
		Interpreter = nullptr;
		return;
	}

	CurrentCodeState = ECodeState::WaitingForNextStep;
	if (AutoRun)
	{
		if (StepCounter <= MaxStepsPerFrame)
		{
			RunNextRTSStep();
			return;
		}
		else
		{
```

```
                        StepCounter = 0;
                        FrameDelay();
                        return;
                }
        }
        else
        {
                StepCounter = 0;
        }
}
```

Kod 7.3 Funkcija RTSStepComplete iz klase UCodeRunner.

Izvor: vlastita izrada

Kod poziva ove funkcije se prvo provjerava vrijednost bool varijable Aborting, koja označava da bi kod trebao stati. On se najčešće koristi kada korisnik želi zaustaviti kod, što je korisno ukoliko je napravio beskonačnu petlju.

Zatim se označuje da UCodeRunner očekuje pokretanje sljedećeg koraka tako da se varijabli CurrentCodeState daje vrijednost WaitingForNextStep. Zatim se provjerava vrijednost varijable AutoRun. Ona označuje da li bi se sljedeći korak trebao pokrenuti odmah nakon završetka prijašnjeg. Ukoliko da, pregledava se da li je broj koraka koji su izvršeni za redom ispod određenog praga, te ukoliko je se poziva RunNextRTSStep funkcija koja pokreće sljedeći korak. Ukoliko je broj koraka iznad zadanog praga, tada se broj koraka resetira na 0, te se poziva FrameDelay funkcija. Ona odgađa egzekuciju koda za jedan „frame“ igre, te zatim pozove RunNextRTSStep funkciju. To je vrlo važno, pošto bi se u protivnom svi koraci pokrenuli unutar istog frame-a igre. Ukoliko se radi o kompleksnijem kodu, ili o kodu koji sadrži petlju, tada bi se igra zamrznula dok se cijeli kod ne izvrši. U još gorem slučaju, kod bi mogao uči u beskonačnu petlju, te bi ga bilo nemoguće zaustavit. Iako je korisniku dana mogućnost zaustavljanja koda, ukoliko je igra zamrznuta, ne bi ju mogao iskoristiti.

7.4 URuntimeState

URuntimeState je klasa koja predstavlja trenutno stanje koda i njegovo ponašanje. Svaki čvor stabla apstraktne sintakse ima pripadajuću podklasu URuntimeState klase. To je važno pošto čvorovi stabla apstraktne sintakse predstavljaju strukturu koda, dok URuntimeState predstavlja trenutno stanje. Postoji jedna definicija funkcije, no više

istih funkcija mogu bit pozvane istodobno (na primjer prilikom rekurzije). Stoga je potrebno odvojiti ta dva koncepta.

URuntimeState se kreira pozivanjem CreateRuntimeState funkcije u čvoru stabla apstraktne sintakse. Svaki čvor tada kreira svoju odgovarajuću podklasu URuntimeState klase, te ju inicijalizira.

Jedna od najvažnijih funkcija URuntimeState klase je RunStep funkcija. Ona pokreće sljedeći korak, no pošto taj korak ovisi o samom dijelu koda koji ta klasa predstavlja, svaka podklasa ima drugačije ponašanje. Kasnije će se prikazati neke od njih.

Druga važna funkcija je StepCompleted. Ona jednostavno dohvaća spremljeni pokazivač na UCodeRunner, te mu poziva RTSStepCompleted funkciju. Funkcija Finished čini istu stvar, no prvo označuje da je ova instanca URuntimeState-a završila.

Na kraju, funkcija AddRTSToCodeRunner kreira novu instancu URuntimeState-a te ju dodaje na vrh u UCodeRunneru, kao što je prikazano u sljedećem kodu.

```
void URuntimeState::AddRTSToCodeRunner(UAST_Node* Node)
{
	GetCodeRunner()->AddRTS(Node->CreateRuntimeState(GetCodeRunner()));
}
```

Kod 7.4 Funkcija AddRTSToCodeRunner iz klase URuntimeState.

Izvor: vlastita izrada

Jednostavna funkcija koja uzima čvor stabla apstraktne sintakse, kreira odgovarajuću instancu URuntimeState-a, te ga nadodaje na vrh stoga u CodeRunner.

Kako bi se pojasnio princip na kojem URuntimeState radi, potrebno je prikazati nekoliko konkretnih primjera.

7.4.1 URTS_VariableCall

URTS_VariableCall je URuntimeState koji predstavlja ponašanje poziva varijable. To ponašanje je prikazano u kodu ispod.

```
void URTS_VariableCall::RunStep()
{
	Owner = GetCodeRunner()->ConsumeTempOwner();
	if (!Owner && GetCodeRunner()->GetSymbolTable()-
>TopOwnedVariableCalls.Contains(VariableCall))
		Owner = GetCodeRunner()->GetSymbolTable()->TopOwner;

	if (Owner)
	{
		UValue* val = Owner->GetVariable(VariableCall->VariableName);
		AddDebugMessage("returning a member variables named " + VariableCall-
>VariableName + " of type " + val->Type + ".");
		GetCodeRunner()->SetValue(val);
		Finished();
		return;
	}

	UValue* val = GetCodeRunner()->GetVariable(VariableCall->VariableName);
	GetCodeRunner()->SetValue(val);
	AddDebugMessage("Returning a variable named " + VariableCall->VariableName + "
of type " + GetCodeRunner()->GetValue()->Type + ".");
	Finished();
}
```

Kod 7.5 Funkcija RunStep iz klase URTS_VariableCall.

Izvor: vlastita izrada

Prvo je potrebno pronaći vlasnika varijable, tj. instancu kojoj pripada. Ukoliko je ona eksplicitno definirana, tj. putem operatora pristupa članovima, tada je ona spremljena u CodeRunner-u, te se dohvaća preko ConsumeTempOwner funkcije. Ukoliko nije eksplicitno definiran, tada se traži preko TopOwner-a. tj. instance u kojoj se kod izvršava. Ukoliko ona sadrži varijablu s tim imenom, ta varijabla joj pripada.

Ukoliko vlasnik postoji, tada se u njemu traži vrijednost s nazivom varijable. Tu vrijednost zatim spremimo u CodeRunner od kud ju neki drugi URuntimeState može iskoristiti, te funkcija završava pozivanjem funkcije Finished. Ukoliko vlasnik ne postoji, tada je varijabla ili globalna, ili lokalna u funkciji, te u oba slučaja će se pronaći pozivanjem GetVariable funkcije nad CodeRunner-om.

Važno je primijetiti da se cijelo ponašanje ovog dijela koda odvija u jednom koraku, pošto se odmah poziva Finished funkcija. Primjer URuntimeState-a koji se odvija u više od jednog koraka je URTS_MemberAccess, koji predstavlja ponašanje operatora pristupa članovima.

7.4.2 URTS_MemberAccess

URTS_MemberAccess je URuntimeState od UAST_MemberAccess, čvora, tj. operator pristupa članovima. Njegovo ponašanje je prikazano u sljedećem kodu.

```
void URTS_MemberAccess ::RunStep()
{

	if (!CallerHandled)
	{
		// Solving lhs (caller)
		AddDebugMessage("Solving the caller in member access.");
		AddRTSToCodeRunner(MemberAccess->lhs);
		CallerHandled = true;
		StepCompleted();
		return;
	}

	// Store lhs value as owner of calling function or variable
	lhsVal = GetCodeRunner()->GetValue();
	AddDebugMessage("Storing caller, which is of type " + lhsVal->Type + ", and
solving callee.");
	GetCodeRunner()->SetTempOwner(lhsVal);
	AddRTSToCodeRunner(MemberAccess->rhs);
	Finished();
}
```

Kod 7.6 Funkcija RunStep iz klase URTS_MemberAccess.

Izvor: vlastita izrada

Bool varijabla CallerHandled ima vrijednost laž prilikom izvođenja prvog koraka. Stoga se poziva funkcija AddRTSToCodeRunner te joj se dodaje pozivatelj kao argument. Ta funkcija kreira URuntimeState, te ga dodaje na vrh stoga u CodeRunner-u. Nakon toga se označuje da je prvi korak gotov na način da se varijabli CallerHandled postavlja istina. Nakon toga korak završava pozivanjem funkcije StepCompleted.

Pošto je sada pozivatelj na vrhu stoga, on se sljedeći pokreće, što može dovesti do pokretanja drugih URuntimeState-ova. Nakon što se svi izvrše, vraća se vrijednost vlasnika, koja se sprema u CodeRunner, te se vlasnik vadi sa stoga. Sada je ponovo URTS_MemberAcces na vrhu stoga, te se ponovo pokreće njegov sljedeći korak.

U ovom slučaju je CallerHandled varijabla istinita, što znači da se trenutno izvršava drugi korak. Dohvaća se prijašnja vraćena vrijednost iz CodeRunner-a, te se ponovo sprema u njega, ali kao privremeno spremljeni vlasnik. Nakon toga se pozvana

varijabla ili čvor dodaje na stog, te URTS_MemberAccess završava pozivanjem funkcije Finished.

Ukoliko pretpostavimo da je pozvani čvor poziv varijable, tada će se pokrenuti prijašnji kod iz URTS_VariableCall klase, te će se vlasnik kojeg je URTS_MemberAccess spremio dohvatiti preko ConsumeTempOwner funkcije. Nakon što se ona izvrši, tada će se izvaditi iz stoga. Samim time, URTS_MemberAccess će ponovo biti na vrhu stoga, no pošto je završio, će se i on izvadit. Nakon toga se pokreće sljedeći URuntimeState, dok se stog ne isprazni.

7.4.3 URTS_FunctionCall

URTS_FunctionCall je URuntimeState pozivanja funkcije. Pošto funkcija može imati više argumenata, ova klasa može imati puno koraka, zbog čega može biti relativno kompleksna. Kao i kod pozivanja varijable, pozivanje funkcije započinje traženjem vlasnika, kao što je prikazano u sljedećem isječku koda.

```
if (!OwnerSet)
{
	owner = GetCodeRunner()->ConsumeTempOwner();
	if (!owner && GetCodeRunner()->GetSymbolTable()-
>TopOwnedFunctionCalls.Contains(FunctionCall))
		owner = GetCodeRunner()->GetSymbolTable()->TopOwner;

	OwnerSet = true;
}
```

Kod 7.7 Isječak koda iz klase URTS_FunctionCall koji traži vlasnika funkcije.

Izvor: vlastita izrada

Prvo se provjerava da li je definiran vlasnik, koristeći vrijednost bool varijable OwnerSet. Prilikom prvog koraka ta vrijednost će biti laž, te će se stoga taj kod pokrenuti. Sam vlasnik se traži na sličan način kao i kod poziva varijable. Neovisno o tome da li vlasnik postoji ili ne, varijabli OwnerSet se pridodaje vrijednost istine, te se zbog toga ovaj dio koda neće ponovo pokretati u sljedećim koracima. Važno je koristiti OwnerSet varijablu umjesto provjere da li vrijednost owner varijable validna ili je

nullptr, pošto je moguće da je funkcija definirana u globalnom bloku, te da nema vlasnika.

Nakon traženja vlasnika se pokreću argumenti, kao što je prikazano u sljedećem isječku koda.

```
// The argument value is stored in CodeRunner
if (Parameters.Num() < ParamIndex)
{
	AddDebugMessage("Store returned argument of type " + GetCodeRunner()-
>GetValue()->Type + " in function call named " + FunctionCall->FunctionName + ".");
	Parameters.Add(GetCodeRunner()->GetValue());
}

if (FunctionCall->Arguments.Num() > ParamIndex)
{
	AddDebugMessage("Run the expression at index " + FString::FromInt(ParamIndex) +
" for function call named " + FunctionCall->FunctionName + ".");
	AddRTSToCodeRunner(FunctionCall->Arguments[ParamIndex]);
	ParamIndex++;
	StepCompleted();
	return;
}
```

Kod 7.8 Funkcija iz klase URTS_FunctionCall koja pokreće argumente funkcije.

Izvor: vlastita izrada

Tu su važne dvije varijable, vektor Parameters tipa UValue, koji sadrži vrijednosti argumenata, te ParamIndex, koji predstavlja indeks argumenta koji je potrebno pokrenut. Parameters započinje prazan, te ParamIndex započinje sa vrijednošću 0.

Stoga tijekom prvog koraka će se prvi dio koda preskočit. Ukoliko funkcija nema argumenata, tada će se i drugi dio koda preskočit. No ukoliko ima, tada će se dohvatiti prvi argument, te nadodati na stog. ParamIndex će se povećati za jedan, te korak završava. Nakon što se argument izvršio, URTS_FunctionCall će se ponovo nalaziti na vrhu stoga, te će se ponovo pozvati ova funkcija. U ovom slučaju će ParamIndex, koji ima vrijednost 1 biti veći od broja elemenata vektora Parameters, koji je prazan. Stoga će se dohvatiti vrijednost koja je spremljena u CodeRunner-u, tj. vrijednost prvog argumenta, te će se spremiti u Parameters vektor. Ciklus dohvaćanja vrijednosti argumenata te pokretanje sljedećeg se nastavlja dok se svi argumenti ne pokrenu i spreme.

Nakon što su se svi argumenti spremili, pokreće se sljedeći isječak koda.

```
Definition = GetFunctionDefinition();

if (!Definition)
{
	// Special case if there is no default constructor
	AddDebugMessage("Initializing default constructor for instance of type " +
FunctionCall->FunctionName + ".");
	URTS_InstanceCreator* instanceCreator =
NewObject<URTS_InstanceCreator>(GetCodeRunner());
	instanceCreator->Init(GetCodeRunner());
	instanceCreator->SetValues(FunctionCall->FunctionName, false,
TArray<UValue*>());
	GetCodeRunner()->AddRTS(instanceCreator);
	Finished();
	return;
}
```

Kod 7.9 Isječak koda iz klase URTS_FunctionCall koji poziva default konstruktor.

Izvor: vlastita izrada

Funkcija GetFunctionDefinition koristi podatke iz SymbolTable-a te vrijednosti parametra kako bi pronašla odgovarajuću definiciju funkcije. Ukoliko definicija nije pronađena, pošto je kod već analiziran, te stoga znamo da je korektan, jedina mogućnost je da se radi o default konstruktoru. U tom slučaju kreiramo instancu URTS_InstanceCreator. URTS_InstanceCreator je posebna podklasa URuntimeState klase koja nasljeđuje od ImplicitRTS klase. To znači da ona nema pripadajući čvor stabla apstraktne sintakse, te služi kao pomoćna klasa. O njima ćemo reći nešto više kasnije.

URTS_InstanceCreator služi za kreiranje instanci pozivajući odgovarajući konstruktor. Funkcija SetValues ima tri argumenta, prvi je tip instance koji se kreira, drugi je da li se kreira polje te instance ili samo jedna instanca, te treći je vektor vrijednosti koje se koriste kao argumenti konstruktora. Nakon što se vrijednosti postave, tada se URTS_InstanceCreator dodaje na stog, te URTS_FunctionCall završava.

Ukoliko je pronađena definicija klase, pokreće se kod koji je prikazan ispod.

```
if(GetCodeRunner()->GetSymbolTable()->ClassNames.Contains(Definition-
>FunctionData.FunctionName))
```

```
{
	//Special case if this is a constructor
	AddDebugMessage("Initializing constructor for instance of type " + Definition-
>FunctionData.FunctionName + ".");
	URTS_InstanceCreator* instanceCreator =
NewObject<URTS_InstanceCreator>(GetCodeRunner());
	instanceCreator->Init(GetCodeRunner());
	instanceCreator->SetValues(Definition->FunctionData.FunctionName, Definition-
>FunctionData.ReturnsArray, Parameters);
	GetCodeRunner()->AddRTS(instanceCreator);
	Finished();
	return;
}
```

Kod 7.10 Isječak koda iz klase URTS_FunctionCall koji poziva konstruktor definiran sa strane korisnika. Izvor: vlastita izrada

Ovaj dio koda je vrlo sličan kao i prijašnji. U ovom slučaju se gleda da li postoji definirana klasa koja ima isto ime kao i funkcija. Ukoliko ima, tada se isto radi o konstruktoru. Kreira se novi InstanceCreator, te se ovaj put predaju parametri za konstruktor.

Ukoliko se ne radi o konstruktoru, tada se odvija sljedeći isječak koda.

```
// Call the function
AddDebugMessage("Initialize call for function named " + FunctionCall->FunctionName
+".");
URTS_ChainedFunctionCall* ChainedFunctionCall =
NewObject<URTS_ChainedFunctionCall>(GetCodeRunner());

ChainedFunctionCall->Init(GetCodeRunner());
ChainedFunctionCall->SetData(Definition, owner, Parameters);
GetCodeRunner()->AddRTS(ChainedFunctionCall);
Finished();
```

Kod 7.11 Isječak koda iz klase URTS_FunctionCall koji pokreće instance URTS_ChainedFunctionCall klase. Izvor: vlastita izrada

U ovom slučaju se kreira druga instanca tipa URTS_ChainedFunctionCall. URTS_ChainedFunctionCall je također implicitni URuntimeState, te služi za pokretanje funkcija. Nakon što se kreira se poziva SetData funkcija koja također ima tri parametra. Prvi je definicija funkcija koja se poziva, drugi je vlasnik, te treći je vektor

sa parametrima. Zatim se taj ChainedFunkcionCall nadoda na stog, te URTS_FuncionCall završava.

7.4.4 URTS_ChainedFunctionCall

Tijekom pokretanja URTS_FuncionCall klase može se primijetiti da su se pokrenuli svi argumenti, te da su se dohvatile njihove vrijednosti. Međutim, moguće je da je te vrijednosti potrebno konvertirati. Na primjer, funkcija koja prima string kao argument može primiti i float. Taj float je potrebno konvertirati u string kako bi se ta funkcija uspješno izvršila. Tome služi URTS_ChainedFunctionCall. On iterira kroz sve parametre, te poziva druge funkcije kako bi ih konvertirao u traženi tip, ukoliko je to potrebno.

Prvi dio tog procesa je prikazan u sljedećem isječku koda.

```
if (ConvertingValue)
{
	AddDebugMessage("Conversion of given value to type " + FunctionDefinition-
>FunctionData.ParameterData[ArgumentIndex].Type + " for argument " +
FunctionDefinition->FunctionData.ParameterData[ArgumentIndex].Name   + " in function "
+ FunctionDefinition->FunctionData.FunctionName + " completed.");
	ConvertingValue = false;
	Values[ArgumentIndex] = GetCodeRunner()->GetValue();
	ArgumentIndex++;
}

// Loop through values and convert them if necessary
while (ArgumentIndex < Values.Num())
{
	FString currentType = Values[ArgumentIndex]->Type;
	FString wantedType = FunctionDefinition-
>FunctionData.ParameterData[ArgumentIndex].Type;
	if (wantedType == "element type") wantedType = currentType;
	AddDebugMessage("For function " + FunctionDefinition->FunctionData.FunctionName
+ " argument " + FunctionDefinition->FunctionData.ParameterData[ArgumentIndex].Name +
" is of type " + wantedType + ". Passed value is of type " + currentType + ".");

	if (currentType == wantedType || GetCodeRunner()->GetSymbolTable()-
>IsChildOfByName(currentType, wantedType))
	{
		AddDebugMessage("No need to convert passed value.");
		ArgumentIndex++;
		continue;
	}

	UAST_FunctionDefinition* converter = GetCodeRunner()->GetSymbolTable()-
>GetImplicitConverterByName(currentType, wantedType);
	AddDebugMessage("Conversion required. Converting " + currentType + " to " +
wantedType + " using " + converter->FunctionData.FunctionName + " converter.");
```

```
        ConvertingValue = true;

        URTS_ChainedFunctionCall* converterRTS =
NewObject<URTS_ChainedFunctionCall>(GetCodeRunner());
        TArray<UValue*> ValueToConvert;
        ValueToConvert.Add(Values[ArgumentIndex]);
        converterRTS->Init(GetCodeRunner());
        converterRTS->SetData(converter, Owner, ValueToConvert);
        GetCodeRunner()->AddRTS(converterRTS);
        StepCompleted();
        return;
}
```

Kod 7.12 Isječak koda iz klase URTS_ChaineDFunctionCall, koji konvertira sve argumente na željene tipove. Izvor: vlastita izrada

Ovaj dio koda je sličan kodu u URTS_FunctionCall, pošto oboje iteriraju kroz listu parametara. Tu su važne tri varijable: bool ConvertingValue, koji je pokazuje da li je potrebno konvertirati neku vrijednost. Druga je vektor vrijednosti Values, koji sadrži sve argumente za pozivanje funkcije. Te vrijednosti je možda potrebno konvertirati. Zadnja varijabla je ArgumentIndex, te predstavlja indeks argumenta koji se trenutno provjerava.

Prilikom prvog poziva, varijabla ConvertingValue će imati vrijednost laž, te će se prvi dio koda preskočit. Nakon toga započinje iteracija kroz argumente. Tip prve vrijednosti se uspoređuje sa tipom prvog argumenta definicije funkcije. Ukoliko su isti, ili vrijednost nasljeđuje od tipa argumenta, tada se ArgumentIndex povećava, te se petlja ponavlja.

Ukoliko nisu kompatibilni, potrebno je konvertirati vrijednost u zadani tip. Preko SymbolTable-a se dohvaća funkcija konvertiranja koja je ili već unaprijed definirana za određene tipove, ili ju je korisnik definirao u kodu. Zatim ConvertingValue varijabla poprima vrijednost istine, kreira se nova instanca URTS_ChainedFunctionCall klase, pridoda joj se definicija funkcije konvertera, vrijednost koja se mora konvertirati, te se nadoda na vrh stoga. Tada korak završava.

Nakon što se funkcija konvertiranja izvršila, ovaj dio koda se ponovo pokreće. U ovom slučaju je ConvertingValue varijabla istinita, te joj se vrijednost vraća na laž. Zatim se dohvaća konvertirana vrijednost spremljena u CodeRunner-u, te se originalna vrijednost u Values vektoru zamjenjuje s njom. Zatim se nastavlja iteracija kroz parametre dok se svi ne provjere.

Nakon toga se izvršava sljedeći isječak koda.

```
AddDebugMessage("All type conversions for function " + FunctionDefinition-
>FunctionData.FunctionName + " completed.");
GetCodeRunner()->AddStackFrame(Owner);
for (int i = 0; i < Values.Num(); ++i)
{
	// Duplicate value before passing it, if it is not passed by reference
	UValue* ValueToPass;
	if (FunctionDefinition->FunctionData.ParameterData[i].IsPassByReference)
	{
		AddDebugMessage("Argument " + FunctionDefinition-
>FunctionData.ParameterData[i].Name + " is passed by reference. Passing value.");
		ValueToPass = Values[i];
	}
	else
	{
		AddDebugMessage("Argument " + FunctionDefinition-
>FunctionData.ParameterData[i].Name
			+ " is not passed by reference. Duplicating value before passing
it to the function.");
		ValueToPass = Values[i]->Duplicate();
	}

	FString name = FunctionDefinition->FunctionData.ParameterData[i].Name;
	GetCodeRunner()->AddVariable(name, ValueToPass);
}

AddDebugMessage("Running function " + FunctionDefinition->FunctionData.FunctionName +
" with properly converted types.");
AddRTSToCodeRunner(FunctionDefinition);
Finished();
```

Kod 7.13 Isječak koda iz URTS_ChainedFunctionCall klase, koji pokreće funkciju sa konvertiranim parametrima. Izvor: vlastita izrada

Nakon konvertiranja parametara se može kreirati novi stack frame koji sadrži sve podatke potrebne da bio se funkcija uspješno izvršila. Zatim se iterira kroz sve argumente. Ukoliko je argument označen kao „pass by value“, tada se pripadajuća vrijednost duplicira, te se dodaje kao varijabla u novi stack frame. Ukoliko je „pass by reference“, tada se vrijednost direktno nadodaje kao varijabla. Nakon toga se sama definicija nadodaje na stog, te URTS_ChainedFunctionCall završava.

7.4.5 URTS_FunctionDefinition

URTS_FunctionDefinition je klasa koja pokreće definirano ponašanje funkcije. Važno je napomenuti da funkcije definirane u C++ kodu imaju svoju vlastitu implementaciju , te ne sadrže UAST_Scope kao dijete. Ukoliko definicija funkcije sadrži UAST_Scope, tada je funkcija napisana od strane korisnika, te se odvija isječak sljedećeg koda.

```
// Functions stack frame and initial scope with parameters as variables
// are already created by function call.
if (!ScopeCompleted)
{
	AddDebugMessage("Preparing to run scope for " + FunctionDefinition-
>FunctionData.FunctionName + ".");
	// Custom function definitions should override RunStep to add their own
behavior
	AddRTSToCodeRunner(FunctionDefinition->Scope);
	ScopeCompleted = true;
	StepCompleted();
	return;
}
```

Kod 7.14 Isječak koda iz klase URTS_FunctionDefinition, koji pokreće kod funkcije.

Izvor: vlastita izrada

Pošto URTS_ChainedFunctionCall kreira StackFrame i nadoda argumente kao lokalne varijable, URTS_FunctionDefinition može direktno krenuti na analizu UAST_Scope djeteta. U prvom koraku je bool varijabla ScopeCompleted laž, te joj se vrijednost namješta na istinu, URTS_Scope se kreira i nadoda na stog, te korak završava. Sav kod funkcije se nalazi u njemu.

Funkcija završava kada se pokrene URTS_Return. No ukoliko se radi o void funkciji nije nužno da se return nalazi unutar koda. U slučaju da je funkcija konstruktor, tada se return ne smije nalaziti u funkciji. Stoga je potreban sljedeći isječak koda.

```
// If we reach this code, then the function completed without reaching a return
statement.
// This means that this is a void function. Pop the stack frame and clear return
value.
AddDebugMessage("Function " + FunctionDefinition->FunctionData.FunctionName
	+ "completed.");
UValue* own = GetCodeRunner()->GetOwner();
GetCodeRunner()->PopStackFrame();
// Special case if this is a constructor
```

```
if (GetCodeRunner()->GetSymbolTable()->ClassNames.Contains(FunctionDefinition-
>FunctionData.FunctionName))
	GetCodeRunner()->SetValue(own);
else GetCodeRunner()->SetValue(nullptr);
Finished();
```

Kod 7.15 Isječak koda iz klase URTS_FunctionDefinition, koji se pokreće ukoliko je funkcije void, te ne završava sa return. Izvor: vlastita izrada

Nakon što se cijeli URTS_Scope pokrenuo i završio, te nije pokrenuta URTS_Return klasa, tada se pokreće ovaj kod. Prvo se nalazi vlasnik funkcije, te se cijeli stack frame briše sa stoga. Samim time nestaju i sve lokalne varijable. Vlasnik je koristan u slučaju da se radi o konstruktoru, pošto on vraća instancu koju je kreirao. Ukoliko nije konstruktor, tada se povratna vrijednost briše, te URTS_FunctionDefinition završava.

7.4.6 URTS_Return

URTS_Return je klasa koja predstavlja ponašanje UAST_Return čvora. To ponašanje je prikazano u sljedećem kodu.

```
void URTS_Return::RunStep()
{
	if (!CheckedReturnExpression && Return->ReturnExpression)
	{
		AddDebugMessage("Solving the return expression.");
		AddRTSToCodeRunner(Return->ReturnExpression);
		CheckedReturnExpression = true;
		StepCompleted();
		return;
	}

	// If there is return expression, get the value from it
	// and place it in the other stack frame
	UValue* returnValue = nullptr;
	if (Return->ReturnExpression) returnValue = GetCodeRunner()->GetValue();

	AddDebugMessage("Finishing function");
	GetCodeRunner()->PopStackFrame();
	GetCodeRunner()->SetValue(returnValue);
	Finished();
}
```

Kod 7.16 RunStep funkcija iz klase URTS_Return.

Izvor: vlastita izrada

Prvi korak ove funkcije započinje provjerom da li UAST_Return čvor ima dijete. Ukoliko ima, to znači da ova funkcija vraća neku vrijednost. To dijete se nadoda na stog, te se CheckedExpression varijabla stavlja na istinu, kako se taj dio koda ne bi ponovio sljedeći korak. Nakon što pokretanje djeteta završi, dohvaća se vraćena vrijednost, stack frame se vadi sa stoga, te se ta vrijednost ponovo vraća na ovu upravo dohvaćenu. Nakon toga URTS_Return završava. Ukoliko URTS_Return nema dijete, tada se prvi dio koda preskače, stack frame se vadi, vraća se nullptr kao vrijednost, pošto se radi o void funkciji, te URTS_Return završava.

Potrebno je naglasiti da cijela funkcija živi unutar stack frame-a. Samim time, kako se on izvadio iz stoga, sve lokalne varijable, te i sama funkcija su se izbrisale.

8 CodePlayerController

Prijašnji dio ovoga rada je pokazao na koji način se kod analizira i pokreće, dok će sljedeći dio pokazati na koji način taj kod utječe na igrin svijet, te kako se povezuje sa ostalim sustavima te igre. Ponajprije je važno pojasnit koncept kontrolera (controller) u Unreal Engine-u.

PlayerController je klasa zadužena za primanje naredbi sa strane igrača, koja ih procesira i šalje ostalim objektima u igri. Lako se može dohvatiti njena referenca unutar igre, neovisno od kuda se traži, te se stoga često koristi za spremanje podataka i objekta koji se moraju dohvatiti iz raznih dijelova igre.

U tu svrhu je kreirana klasa ACodePlayerControllerBase, koja nasljeđuje od kontrolera. Ona služi kao poveznica između pokretanja koda i ostatka igre. Sve potrebne klase i funkcije, te kod napisan sa strane korisnika joj se šalju, te ga ona zatim analizira i pokreće. Bilo kakve događaje ili greške koje se pojavljuju tijekom analize ili pokretanja prosljeđuje ostalim objektima kako bi oni mogli reagirati na odgovarajući način.

Prije samog kompajliranja koda potrebno je dohvatiti sve klase koje se nalaze u svijetu. Te klase se dodaju u stablo apstraktne sintakse pozivom funkcije AddClass unutar ACodePlayerControllerBase klase, koja je prikazana ispod.

```
void ACodePlayerControllerBase::AddClass_Implementation(UAST_ClassDefinition*
ClassDefinition)
{
	if (!ClassDefinition) return;

	if (IsClassAlreadyDefined(ClassDefinition->Name))
	{
		// Class is already added, just increase the counter
		ClassCounter[ClassDefinition->Name]++;
		return;
	}

	// Add new class
	ExtraClasses.Add(ClassDefinition);
	ClassCounter.Add(ClassDefinition->Name, 1);
}
```

Kod 8.1 Funkcija AddClass iz klase ACodePlayerControllerBase.

Izvor: vlastita izrada

Pošto se često u svijetu nalazi više instanci iste klase, potrebno je paziti da se klasa doda u stablo samo jedanput. U protivnom će ista klasa biti definirana više puta, te će kod bacit grešku. Za brojanje klasa se koristi mapa ClassCounter, koja sadrži ime klase kao ključ, te int kao vrijednost, koja predstavlja broj koliko se puta ta klasa pokušala dodati u stablo apstraktne sintakse. Ukoliko se klasa dodaje prvi put, tada se ta klasa dodaje u vektor ExtraClasses, koje se dodaju u stablo prije kompajliranja. U protivnom se samo poveća brojač za tu određenu klasu.

Razlog zašto je potrebno pratiti broj pokušaja dodavanja klase je taj što je moguće i maknuti klasu iz koda. Na primjer ako imamo tri instance iste klase, svaka će pozvati AddClass funkciju. Samim time, brojač za tu klasi će imati vrijednost 3. Ukoliko se jedna od tih instanci uništi, ona će se pokušati „izvaditi“ iz koda, pozivajući funkciju RemoveClass. Ona će smanjiti brojač te klase za 1, no druge instance će još uvijek postojati. Tek kada se sve instance „izvade“ će se brojač klase spustiti na 0, te će se klasa izvaditi sa liste.

Potrebno je naglasit da se u igri ne koristi ACodePlayerControllerBase klasa direktno, nego klasa ACodePlayerController koja od nje nasljeđuje. Razlog tome je što se ona brine i o drugim sustavima u igri, poput grafičkog sučelja.

9 CodeComponent

U Unreal Engine-u 4 se svi objekti koji se nalaze u svijetu zovu Actor-i, tj. nasljeđuju od klase Actor. Pošto se u CodeBreaker igri upravlja okolinom preko pisanja koda, važno je da ti Actor-i mogu reagirati na promjene i tom istom kodu.

Prvi pokušaj implementacije tog sustava je bilo korištenjem nasljeđivanja. Kreirala se klasa koja nasljeđuje od Actor-a, te sadržava podatke o kodu. Međutim, taj pristup je vrlo nefleksibilan iz više razloga. Kao prvo, sve instance te klase imaju logiku za korištenje koda, iako to često nije potrebno, te u neki slučajevima nije ni poželjno. Također naknadno dodavanje logike za kod u druge klase je nepraktično, pošto ta izmjena utječe na sve druge instance u igri.

Stoga se koristi kompozicija kao rješenje tog problema. Actor-i mogu držat komponente u sebi, te se stoga kreira nova generična komponenta CodeComponent koja se brine o svim podatcima i svoj logici potrebnoj za modifikaciju preko koda.

Prednost komponente je u tome što se može pridodati na svaku instancu Actor-a u svijetu zasebno, te sam Actor ne mora imati nikakvu logiku kako bi komponenta pravilno radila. To dopušta da se nadodaje praktički bilo kojem objektu u igri bez potrebnih modifikacija. Samo dodavanje komponente na Actora i mijenjanje parametara je vrlo jednostavno, te je stoga sustav jednostavan za korištenje i vrlo fleksibilan.

9.1 Klase u CodeComponent-u

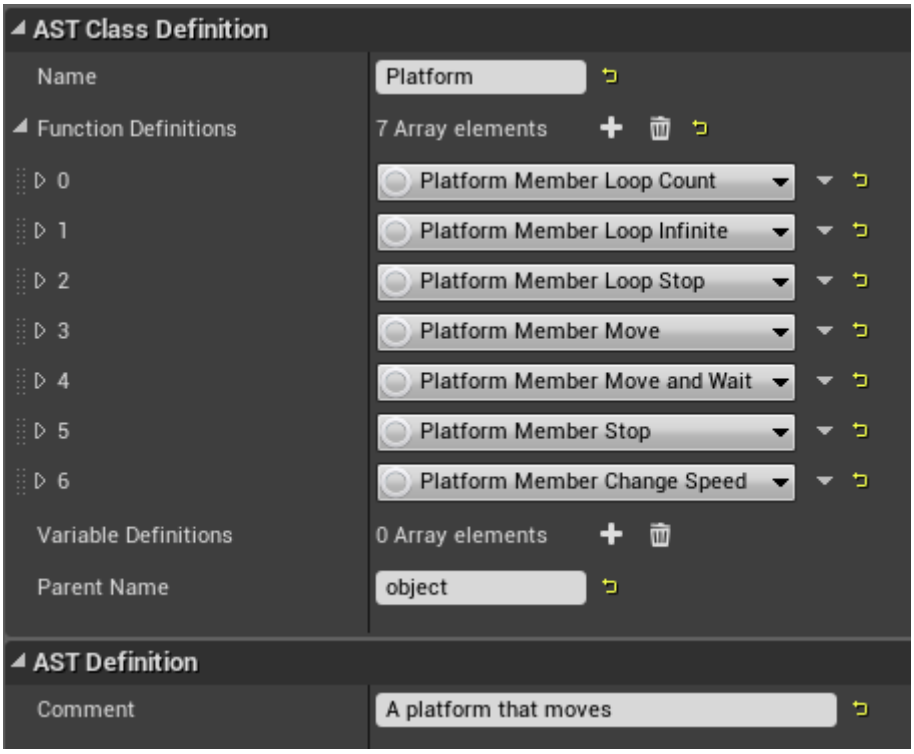
Svaka instanca CodeComponent-a mora imati definiranu klasu, tj. klasu koja predstavlja Actor-a kojem ona pripada. Važno je obratit pažnju na način na koji se klase dodaju u stablo apstraktne sintakse, što je bilo prikazano sa funkcijom AddClass u CodePlayerControllerBase klasi. Tijekom inicijalizacije svaki CodeComponent pokuša dodati svoju klasu, te se prva klasa dodaje u stablo, dok ostale samo povećavaju brojač. Stoga je vrlo važno da sve klase CodeComponent-a istog imena imaju istu definiciju. Ukoliko su definirane dvije različite klase istog naziva, tada će se prva inicijalizirana koristiti u stablu, pošto će se druga smatrati duplikatom, te samo povećati brojač. Te pošto se druga klasa neće dodati na stablo, analiza koda neće pronaći dvije različite klase istog naziva, neće baciti grešku. To može dovesti do

problema koje je teško otkriti, pogotovo zato jer redoslijed inicijalizacije komponenti nije definiran, te može varirat.

Kako bi se smanjila mogućnost takvih grešaka, postoje dva načina definiranja klase za CodeComponent.

9.1.1 UPredefinedClass

UPredefinedClass je klasa koja nasljeđuje od UAST_ClassDefinition klase, te se može kreirati koristeći Blueprint sustav. Samim time, svi parametri te klase se mogu definirati unutar editora. Primjer mijenjanja tih parametara je prikazan na sljedećoj slici.

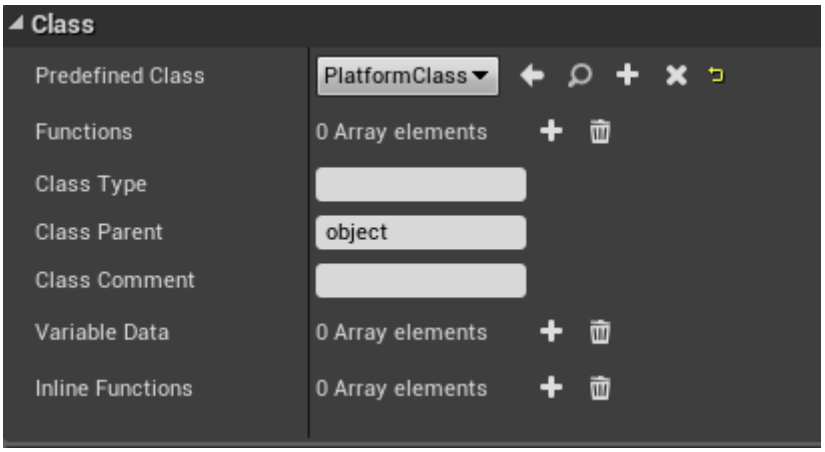


Slika 9. Kreiranje klase kao zaseban Blueprint.

Izvor: vlastita izrada

Na slici možemo vidjeti parametre potrebne za definiciju klase, u ovom slučaju klase koja predstavlja pomičnu platformu. Način na koji se funkcije dodavaju u klasu će se kasnije pojasnit.

Ovaj način kreiranja klase ima prednost u tome što kreira novu Blueprint klasu koja sadrži sve potrebne podatke. U slučaju više platformi u svijetu, svaki CodeComponent može referencirati ovu klasu. Samim time je garantirano da će svaki imati iste podatke o Platform klasi. Na taj se osiguralo da će sve komponente koje koriste ovu klasi biti iste. Dodatna prednost je u tome što je dovoljno izmijeniti ovu klasu, te će sve komponente koje je referenciraju automatski koristiti novu verziju bez ikakvih izmjena na njima. Primjer korištenja predefinirane klase je prikazan na sljedećoj slici.



Slika 10. Postavljanje unaprijed definirane klase u CodeComponent.

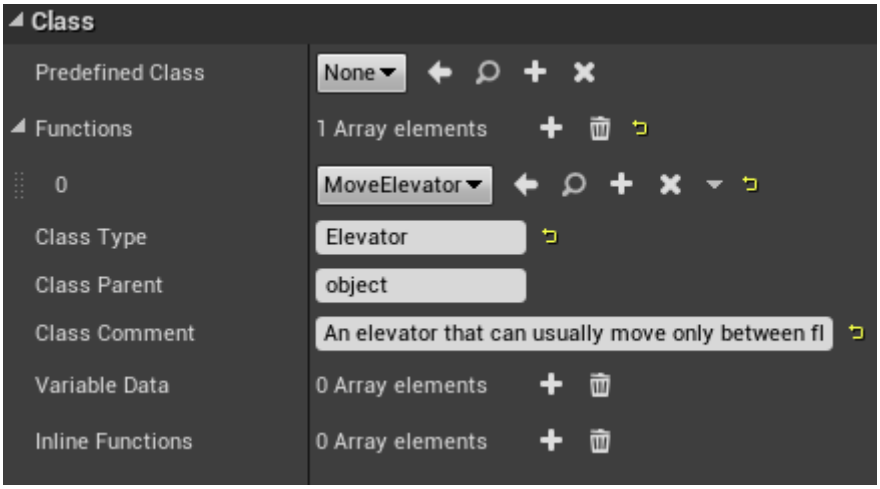
Izvor: vlastita izrada

Na ovoj slici su prikazani neki od parametara CodeComponent-a. Parametar Predefined Class ima vrijednost PlatfromClass, tj. ranije opisane klase. Ostali parametri se koriste za drugi način definiranja klasa, te će se pojasniti u sljedećem poglavlju.

9.1.2 Ugrađene klase

Kreiranje novih Blueprint-a za definiciju klase je korisno jer je te klase vrlo lako referencirati i ponovno upotrebljavati. No vrlo često se u igri pojavljuju slučajevi gdje je potrebno definirati klasu koja se koristi samo jedanput. U tom slučaju bi bilo nepraktično kreirati novi Blueprint za svaku takvu situaciju. Za takve slučajeve postoji mogućnost kreiranja ugrađene (inline) klase. Ugrađena klasa je klasa koja se definira

na samoj instanci CodeComponent-a, te je stoga savršena za slučajeve gdje se neka klasa pojavljuje samo jedanput. Primjer definiranja ugrađene klase je prikazan na sljedećoj slici.



Slika 11. Kreiranje ugrađene klase unutar CodeDComponent-a.

Izvor: vlastita izrada

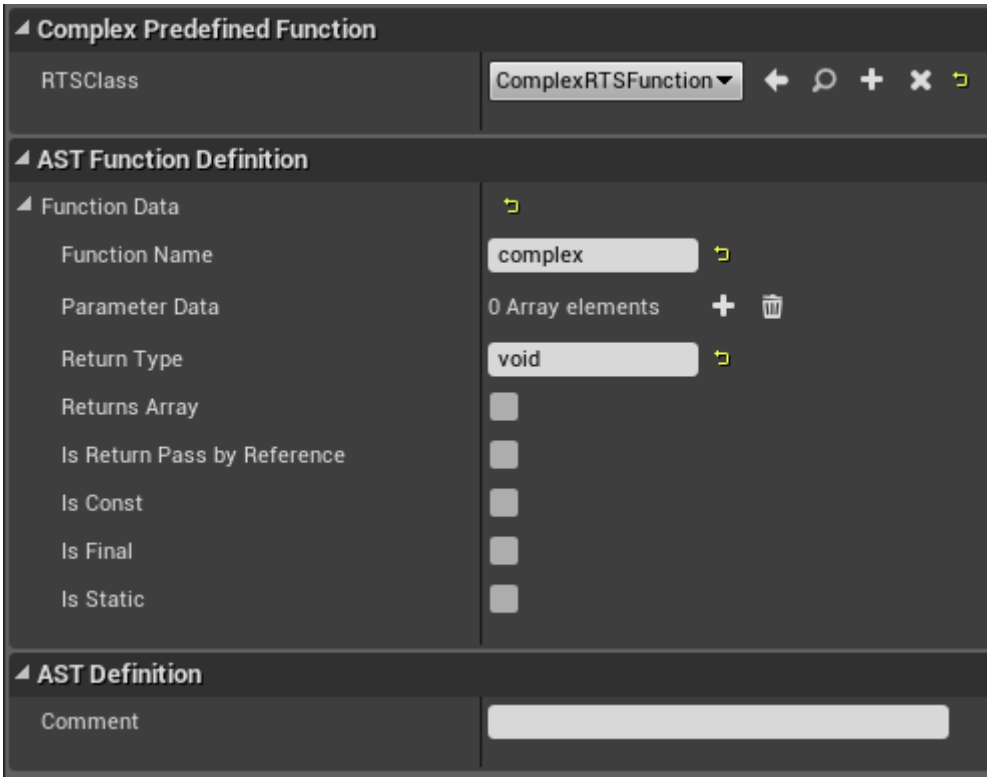
Važno je primijetiti da je vrijednost Predefined Class parametra None, tj. da ne koristi unaprijed definiranu klasu. Ukoliko ju komponenta koristi, tada se ostali parametri ignoriraju, slično kao i na prijašnjoj slici. Ostali parametri su samo podatci koji definiraju klasu, uz iznimku da postoje dva različita parametra za funkcije: Functions i Inline Functions. O njima, i općenito o funkcijama će se pojasniti u sljedećem poglavlju.

9.2 Funkcije

Funkcije su vrlo važne u CodeBreaker igri, pošto su one glavni način na koji korisnik modificira svijet oko sebe. Stoga je važno da sustav kreiranja tih funkcija bude jednostavan i fleksibilan. Postoje tri načina kreiranja funkcija, svaka ima svoje prednosti i mane, ovisno o situaciji u kojoj će se koristit.

9.2.1 Kompleksna funkcija

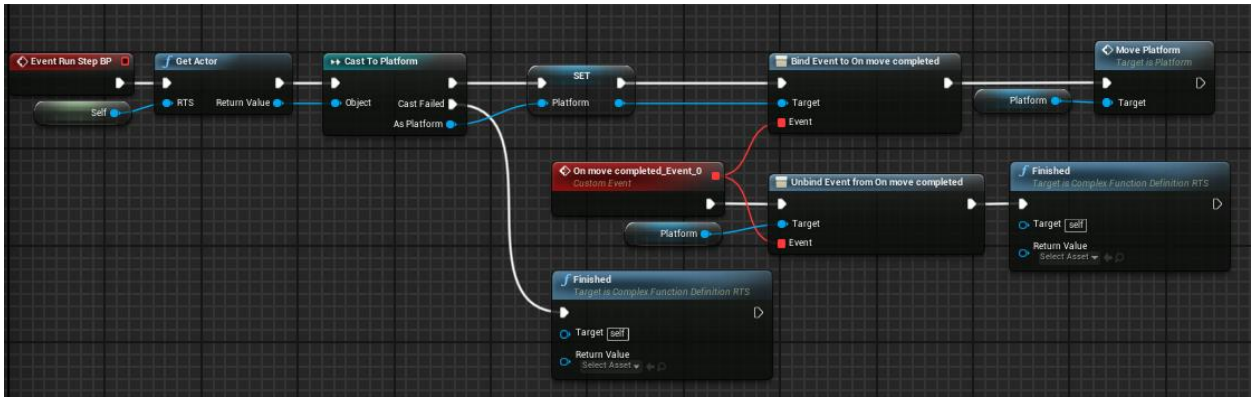
ComplexFunction je klasa koja nasljeđuje od UAST_FunctionDefinition klase, no ima dodatni parametar RTS Class, preko kojeg može odabrati klasu tipa URTS_FunctionDefinition koja će se pokrenuti. Parametri te klase su prikazani na sljedećoj slici.



Slika 12. Definiranje podataka kompleksne funkcije.

Izvor: vlastita izrada

Kao i normalna UAST_FunctionDefinition klasa, ona samo drži podatke o definiciji funkcije, a ne o samoj logici unutar nje. Logika koja se izvršava se nalazi u drugoj klasi tipa UComplexFunctionDefinitionRTS, koja se odabire kao vrijednost RTS Class parametra. Potrebno je samo kreirati tu klasi u Blueprint sustavi, dodati logiku funkcije unutar nje, te u ComplexFunction staviti vrijednost RTS Class parametra na tu klasu. Primjer logike u UComplexFunctionDefinitionRTS klasi je prikazan na sljedećoj slici.



Slika 13. Definiranje logike kompleksne funkcije.

Izvor: vlastita izrada

Kada se funkcija poziva, tada se vrijednost pozivatelja (UValue) spremi u StackFrame. Međutim UValue vrijednosti kreirane sa strane CodeComponenta su tipa CodeComponentInstance, koje sadrže i referencu na sam CodeComponent koji ih je kreirao. Preko njega se može dohvati referenca na samog Actor-a, te samim time se mogu pozvati funkcije na njemu.

U ovom slučaju se prilikom pozivanja funkcije dohvaća referenca na Actor-a koji je pozvao ovu funkciju, te se zatim Cast-a na Actor-a tipa Platform. Zatim se preko Observer-a počinje slušat naredba da je platforma stala, te se zatim platforma pokreće. Kada platforma stane, tada se poziva Finished funkcija, te funkcija završava. Na ovaj način je kreirana funkcija koja pokreće platformu, te zaustavlja egzekucija koda dok platforma ne stane. Također je vidljivo da sama platforma ne mora imati nikakvu logiku specifičnu za CodeComponent ili pozivanje funkcija.

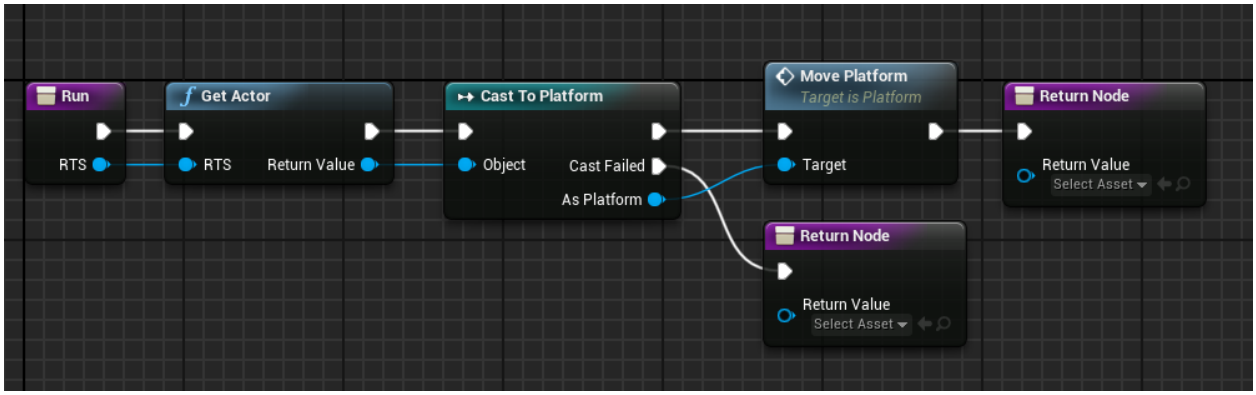
Ovaj način kreiranja funkcija je vrlo fleksibilan, te ima sve mogućnosti kao i URTS_FunctionDefinition, uključujući pokretanje funkcije u više koraka i pozivanje drugi funkcija. No često takva kompleksnost nije potrebna, što dovodi do drugog načina kreiranja funkcija.

9.2.2 Jednostavna funkcija

Iako je sustav kompleksne funkcije fleksibilan, ta fleksibilnost često nije potrebna, te samo dodaje dodatni sloj kompleksnosti. Potrebno je kreirati dvije klase, jednu koja

predstavlja definiciju funkcije i drugu koja predstavlja njenu logiku, te ih međusobno povezati. Stoga postoji i jednostavniji način kreiranja funkcija, koristeći klasu SimplePredefinedFunction.

SimplePredefinedFunction je klasa koja nasljeđuje od UAST_FunctionDefinition klase. No za razliku od drugih klasa koje predstavljaju samo podatke funkcije, SimplePredefinedFunction sama izvršava svoju logiku. To čini preko posebnog RuntimeState-a, koji kad se pokrene poziva funkciju Run u njoj. Samim time potrebno je kreirati samo jednu klasu za kreirati i koristiti funkciju. Primjer jednostavne funkcije je prikazan na slici ispod.



Slika 14. Definiranje logike jednostavne funkcije.

Izvor: vlastita izrada

Ova funkcije dohvaća referencu od Actor-a koji ju je pozvao, Cast-a na platformu, te ju pomiče. Za razliku od prijašnje funkcije, ova funkcije ne zaustavlja egzekuciju koda.

Važno je napomenuti da jednostavna funkcija dolazi sa nekoliko ograničenja. Jedno od njih je da funkcija mora odmah završit (ne može zaustavit egzekuciju koda). Isto tako ne može se odvijati u nekoliko koraka. Također je važno zapamtit da UAST_FunctionDefinition predstavlja podatke o funkciji, te stoga postoji samo jedna instanca, neovisno o tome koliko se funkcija poziva. Pošto SimpleFunctionDefiniton ne koristi RuntimeState za spremanje stanja svake pozvane funkcije, važno je obratit pozornost da ova klasa ne smije imati stanja. Ukoliko, na primjer ona sadrži neku varijablu koja se promijeni, tada će ta varijabla zadržati vrijednost i sljedeći put kad se

pozove. Lokalne varijable unutar Run funkcije se mogu koristiti, pošto će se one izbrisati kad funkcija završi.

Iako ima dosta ograničenja, ovaj način kreiranja funkcija se često koristi u igri, pošto se u većini slučajeva radi o jednostavnim funkcijama.

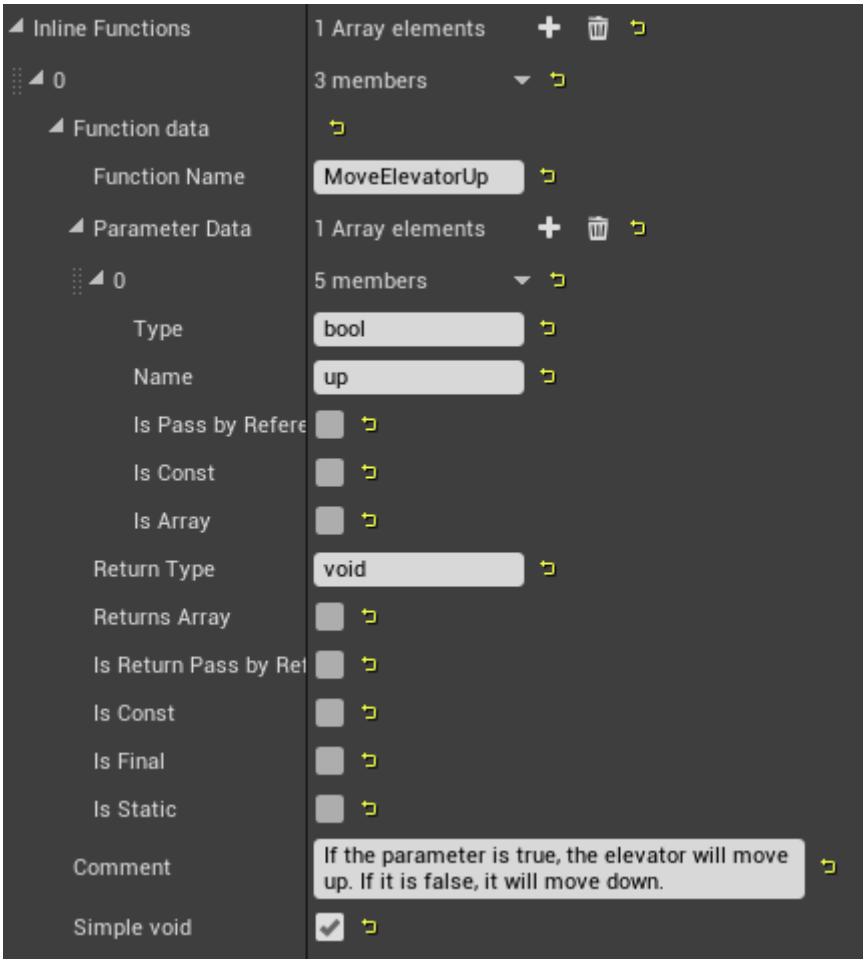
9.2.3 Ugrađene funkcije

Iako su kompleksne i jednostavne funkcije moćan alat, kao i kod kreiranja klasa često se javljaju situacije gdje se neka funkcija koristi samo jedanput. Drugi problem kod jednostavnih i kompleksnih funkcija je taj što su izolirane od ostatka igrinog svijeta. Jedina poveznica funkcije sa svijetom je preko Actor-a kojem CodeComponent pripada. Ukoliko funkcija mora utjecati na neki drugi Actor tada pozivatelj mora imati neki način da ga referencira. To predstavlja problem, pošto se sad javlja potreba za modifikacijom Actor-a kako bi funkcija mogla raditi.

Jedno od mogućih rješenja je da se Actor-u kojeg je potrebno modificirati doda CodeComponent, te da sa zatim on spremi kao varijabla. To dopušta funkciji pristup bez modifikacije samih Actora, no to rješenje je nepraktično, te daje mogućnost korisniku da direktno upravlja varijablom, što nije poželjno.

Taj problem se rješava koristeći Level Blueprint. Level Blueprint je klasa koja predstavlja razinu igre. Pošto se svi Actor-i nalaze u njoj, ona ima brzi pristup svakom od njih. Dodatna prednost te klase je što ona postoji samo u svojoj razini koju predstavlja, te je stoga savršena za jedinstvene događaje gdje neka funkcija utječe na neki drugi Actor.

Na sljedećoj slici je prikazan način definiranja ugrađene funkcije unutar CodeComponent-a.

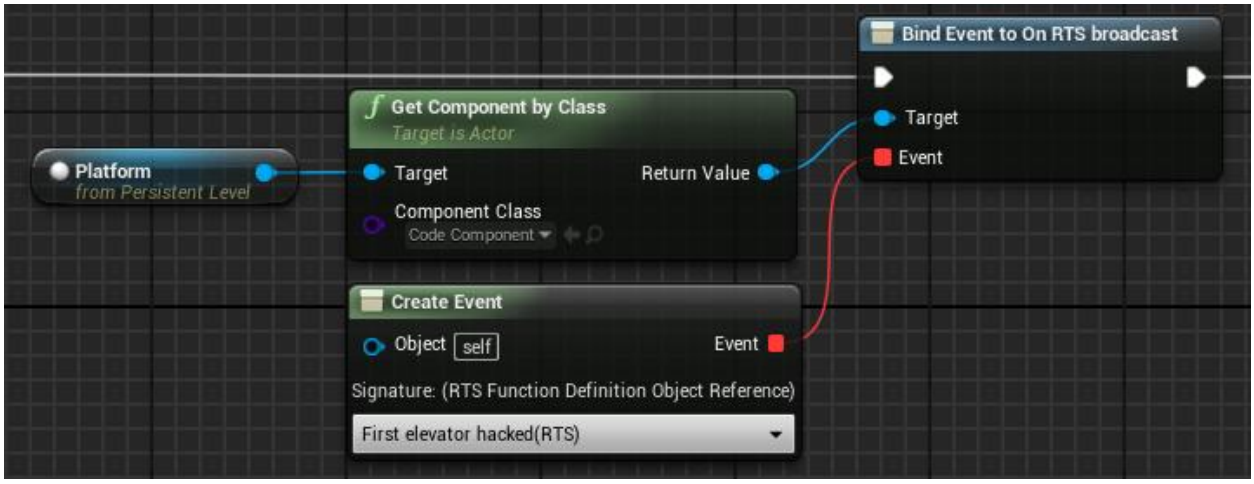


Slika 15. Postavljanje podataka ugrađene funkcije.

Izvor: vlastita izrada

Sa slike se može primijetiti da je definirana jedna funkcija sa nazivom MoveElevatorUp koja vraća void. Sadrži jedan parametar tipa bool koji se zove up. Također sadrži komentar te je označena kao Simple void. Ukoliko funkcija nije Simple void, tada se ponaša kao kompleksna funkcija, tj. može imati više koraka, može pauzirat egzekuciju koda, pozivat druge funkcije i sl. Ukoliko je funkcija Simple void, tada ne vraća vrijednost i ponaša se kao jednostavna funkcija.

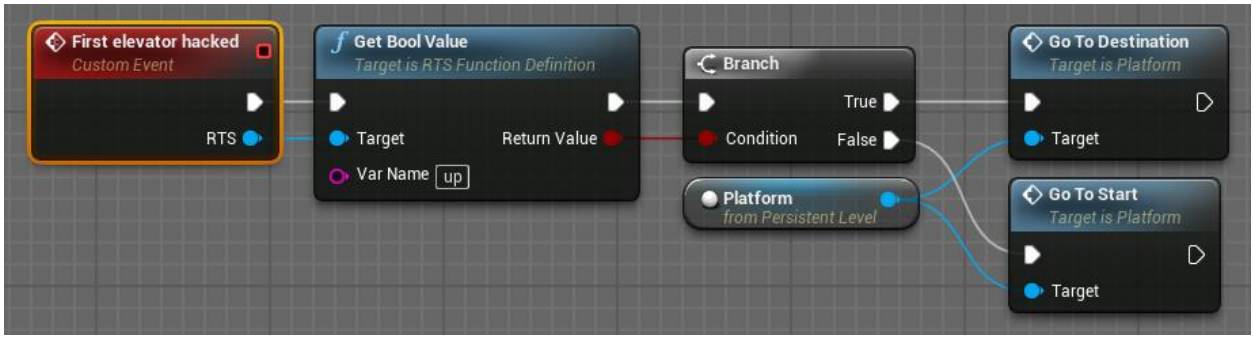
Pošto se koristi Observer za implementaciju funkcije, potrebno je slušati kada se funkcija pozove. Slušanje počinje na samom početku razine, tijekom inicijalizacije, kao što je prikazano na sljedećoj slici.



Slika 16. Povezivanje ugrađene funkcije i Level Blueprint-a preko Observer-a.

Izvor: vlastita izrada

Tijekom inicijalizacije razine se dohvaća referenca na platformu. To je jednostavno pošto Level Blueprint sadrži reference na sve Actor-e unutar sebe. Sa te platforme se zatim dohvaća CodeComponent, te se OnRTSBroadcast Observer povezuje sa FirtsElevatorHacked funkcijom. OnRTSBroadcast je Observer koji se aktivira kada se funkcija pozove. Sama implementacija funkcije MoveElevatorUp se nalazi unutar FirstElevatorHacked funckije unutar Level Blueprint-a, kao što je prikazano na sljedećoj slici.



Slika 17. Definiranje logike ugrađene funkcije.

Izvor: vlastita izrada

Kad se MoveElevatorUp funkcija pozove automatski se dobiva RuntimeState te funkcije, te se preko njega dobiva vrijednost parametra up. Ovisno o njegovoj vrijednosti platforma se podiže ili spušta.

Ovo je jednostavan slučaj pošto se radi o jednoj funkciji, te platforma utječe samo na sebe. No na isti način je moguće utjecati na bilo koje druge Actor-e u razini. U slučaju da platforma ima više funkcija, tada se RTS koristi kako bi se dohvatili podatci o funkciji, te se preko nje može otkriti o kojoj funkciji se radi i pokrenuti odgovarajući kod.

10 Spremanje podataka

Spremanje podataka u CodeBreaker igri se može podijeliti u dvije skupine: trajno i privremeno. Iako će se kasnije oba dvije skupine detaljnije pojasnit, prvo je potrebno objasnit GameInstance klasu.

U Unreal Engine-u 4 GameInstance je objekt koji se kreira na samom početku pokretanja igre, te se uništava tek kada se igra ugasi. To znači da se podatci koji su spremljeni u taj objekt neće izgubiti ukoliko se razina mijenja. To ga čini vrlo moćnim alatom za spremanje i dohvaćanje podataka, te se koristi u obje skupine spremanja podataka.

10.1 Trajno spremanje podataka

Kako korisnik ne bi morao ponavljati igru od početka svaki put kad ju pokrene, potrebno je određene podatke serijalizirati i spremiti na disk. To se čini korištenjem SaveGame objekta u Unreal Engine-u 4.

SaveGame objekt je objekt koji služi za serijaliziranje i deserijaliziranje podataka. Dovoljno je kreirati novi objekt koji nasljeđuje od SaveGame objekta, te u njega spremiti se potrebne podatke. Pozivom funkcije SaveGame se svi podatci spremljeni u tom objektu serijaliziraju i spremaju u posebnu datoteku. Pozivanjem funkcije LoadGame se svi podatci sa te datoteke automatski deserijaliziraju, te se kreira novi SaveGame objekt koji ima te podatke spremljene u sebi.

Koristeći GameInstance na početku igre se učitaju podatci i kreira se novi SaveGame objekt. Pošto je on spremljen u GameInstance objektu, moguće je dohvatiti te podatke

preko njega, neovisno u kojoj razini se korisnik nalazi. To olakšava dohvaćanje podataka pošto je garantirano da isti SaveGame objekt mora postojati u bilo kojoj razini, što pojednostavljuje kod.

Spremanje podataka tijekom igranja je isto jednostavno. Svi podatci se spremaju u SaveGame objekt preko GameInstance objekta. Kada se bilo koji podatak spremi, tada GameInstance serijalizira cijeli objekt. Iako u pravilu to nije poželjno, radi se o vrlo malom broju podataka, pogotovo zato jer nije potrebno spremati imena svih riješenih razina, nego samo posljednje riješene, pošto se razine rješavaju linearno. S obzirom da se radi o vrlo malom broju podataka, sama serijalizacija je neprimjetna tijekom igranja, te ima dodatnu prednost da se najnoviji podatci odmah spreme na disk. Samim time korisnik ne mora brinuti o tome kada će spremiti podatke pošto se oni uvijek sami automatski serijaliziraju.

10.2 Privremeno spremanje podataka i resetiranje razina

Prije svega potrebno je naglasit da se pod privremeno spremanje misli na podatke koji moraju ostati spremljeni tijekom mijenjanja razine, no ne spremaju se na disk. Međutim, kako bi pojasnili njihovu svrhu potrebno je prvo objasnit razlog zašto bi se neka razina ponovo pokrenula.

U CodeBreaker igri postoje određene opasnosti za korisnika koje mogu uništiti lika kojim upravlja, poput vode i sličnih opasnosti u svijetu. No postoje i drugi razlozi zašto bi se razina ponovo pokrenula, na primjer, korisnik ju može svojevoljno ponovo pokrenuti pozivajući ReloadLevel funkciju preko koda, ili koristeći opciju za preskakanje zagonetki.

U svakom slučaju dolazi do resetiranja razine. U pravilu postoje dva načina kako resetirati razinu. Prvi način je da se razina nanovo učita. To će garantirati da će svi objekti biti u početnom stanju. Tu je zatim potrebno vratiti relevantne objekte u stanje prije samog učitavanja. Podatke o tim stanjima je potrebno sačuvati prije samog učitavanja, u protivnom će se izgubiti.

Drugi način je da se razina ne učita nanovo, nego da se određenim objektima vrati prvobitno stanje. Ova verzija ima prednost u tome što je lakše vratiti određene objekte

u prvobitno stanje nego pamtiti sva stanja na koja je korisnik mogao utjecati prije ponovnog učitavanja.

Problem kod druge verzije resetiranja razina je u tome što je sama mogućnost korisnika da upravlja svijetom oko sebe preko pisanja koda vrlo fleksibilna, zbog čega je vrlo teško predvidjeti sve načine na koje može kreirati situaciju iz koje ne postoji izlaz. Stoga se koristi prva verzija resetiranja razina, pošto ona garantira da će se svi objekti vratit u prvobitno stanje, te da preko podataka modificiramo samo ono što je potrebno. Samim time i garantira da u slučaju da korisnik kreira situaciju iz koje ne može izaći, resetiranje razine će riješit taj problem.

Pošto resetiranje razine vraća sva stanja u prvobitno, svi podatci koji opisuju stanja objekta koje treba modificirati bi se izgubili. Stoga se ti podatci spremaju u GameInstance objekt, pošto se on ne uništava niti modificira tijekom mijenjanja razina. Međutim, važno je napomenuti da GameInstance ne sprema te podatke direktno, nego sadrži objekt tipa UPersistentLevelData koji sprema te podatke u sebe. Razlog tome je što to omogućava da se nasljeđivanjem od tog objekta mogu spremati dodatni podatci koji su potrebni za neke specifične razine.

10.2.1 UPersistentLevelData

UPersistentLevelData (PLD) je objekt koji se koristi za spremanje podataka tijekom resetiranja razine. Potrebno je napomenuti da Persistent (trajan) dio naziva je u kontekstu razine. Tijekom resetiranja razine se ti podatci ne brisu te su stoga „Persistent“. Ukoliko se učita nova razina, tada se ti podatci gube. Pošto mu je glavna svrha spremanje podataka, on ne sadrži puno koda. Iako sprema određene podatke o korisniku, npr. koliko dugo rješava trenutnu razinu, koliko puta je zatražio pomoć za rješavanje zagonetki i sl. najvažniji podatci su sljedeći:

- LevelName: Varijabla koja sprema naziv razine.
- CheckpointIndex: Varijabla koja sprema indeks Checkpoint-a, tj. lokacije sa koje će korisnik počet igrat kada se razina resetira.
- PuzzleIndex: Jedna od najvažnijih varijabli koje PLD sprema. Predstavlja indeks zagonetke koju korisnik rješava. Pošto su zagonetke u razinama

linearne, tada je dovoljno spremati samo indeks zadnje neriješene zagonetke. Sve zagonetke prije nje su stoga riješene.

10.2.2 Sustav zagonetki

Vrlo je važno da se tijekom resetiranja razina održe određena stanja, ponajviše je važno da sve zagonetke koje je korisnik riješio ostanu riješene. Neriješene zagonetke se vraćaju u prvotno stanje. Ta stanja je potrebno promijeniti tijekom inicijalizacije razine, te se stoga tijekom učitavanja te razine prvo pokrene sljedeći isječak koda iz Level Blueprint klase.

```
CodeGameInstance = Cast<UCodeGameInstanceBase>(GetGameInstance());
if (CodeGameInstance)
{
	// No stored data, or data that came from another level,
	// meaning that this level loaded for the first time.
	if (!CodeGameInstance->GetPLD()
		|| CodeGameInstance->GetPLD()->LevelName != GetWorld()->GetMapName())
	{
		FirstLoadSetup(CodeGameInstance);
	}
	else
	{
		OnLoad(CodeGameInstance->GetPLD());
	}

	// Bind on puzzle index changed
	CodeGameInstance->GetPLD()->OnNewPuzzleIndex.AddDynamic(this,
		&ACodeLevelScript::NewPuzzleStarted);
}
```

Kod 10.1 Isječak koda unutar Level Blueprint-a koji se pokreće tijekom pokretanja razine. Izvor: vlastita izrada

Tijekom inicijalizacije razine se dohvaća GameInstance objekt, te se Cast-a na tip UCodeGameInstanceBase. To je tip GameInstance objekta koji se koristi u igri, te sadrži PLD. Zatim se gleda da li PLD postoji, te ako postoji da li je naziv razine spremljen u njemu jednak nazivu trenutne razine. Ukoliko ne postoji tada je ovo vjerojatno prva pokrenuta razina, te ukoliko postoji ali imena naziva nisu jednaka, tada je prijašnja razina bila neka druga. U oba slučaja se poziva funkcija FirstLoadSetup. U protivnom PLD postoji, te sadrži ime ove razine, što znači da se razina resetirala, te

se stoga poziva OnLoad funkcija. Obje funkcija će se prikazati, počevši od FirstLoadSetup funkcije koja je prikazana ispod.

```
void ACodeLevelScript::FirstLoadSetup(UCodeGameInstanceBase* CodeGameInstance)
{
	IsFirstLoad = true;
	if (PldClass) CodeGameInstance->CreatePLD(PldClass);
	else
	{
		// Backup in case the class is null
		CodeGameInstance->CreatePLD(UPersistentLevelData::StaticClass());
	}

	CodeGameInstance->GetPLD()->LevelName = GetWorld()->GetMapName();
	CodeGameInstance->GetPLD()->LevelStartTime = FDateTime::Now();
}
```

Kod 10.2 Funkcija FirstLoadSetup unutar ACodeLevelScript klase.

Izvor: vlastita izrada

Ova funkcija se pokreće ukoliko se razina pokreće prvi put (tj. razina se ne resetira). Kreira se novi PLD, te se u njega spremaju naziv razine i trenutno vrijeme. Vrijeme se koristi kako bi se prikazalo korisniku koliko vremena je prošlo da završi razinu, te nije važno za sam sustav zagonetki. Varijabla IsFrstLoad se koristi kako bi se unutar razine mogla pokrenuti neka logika koja se ne bi trebala ponavljati svaki put kada se razina resetira, poput posebne animacije kada korisnik prvi puta dođe do te razine.

Funkcija OnLoad je prikazana ispod.

```
void ACodeLevelScript::OnLoad_Implementation(UPersistentLevelData* PLD)
{
	IsFirstLoad = false;
	PLD->ReloadCount++;
}
```

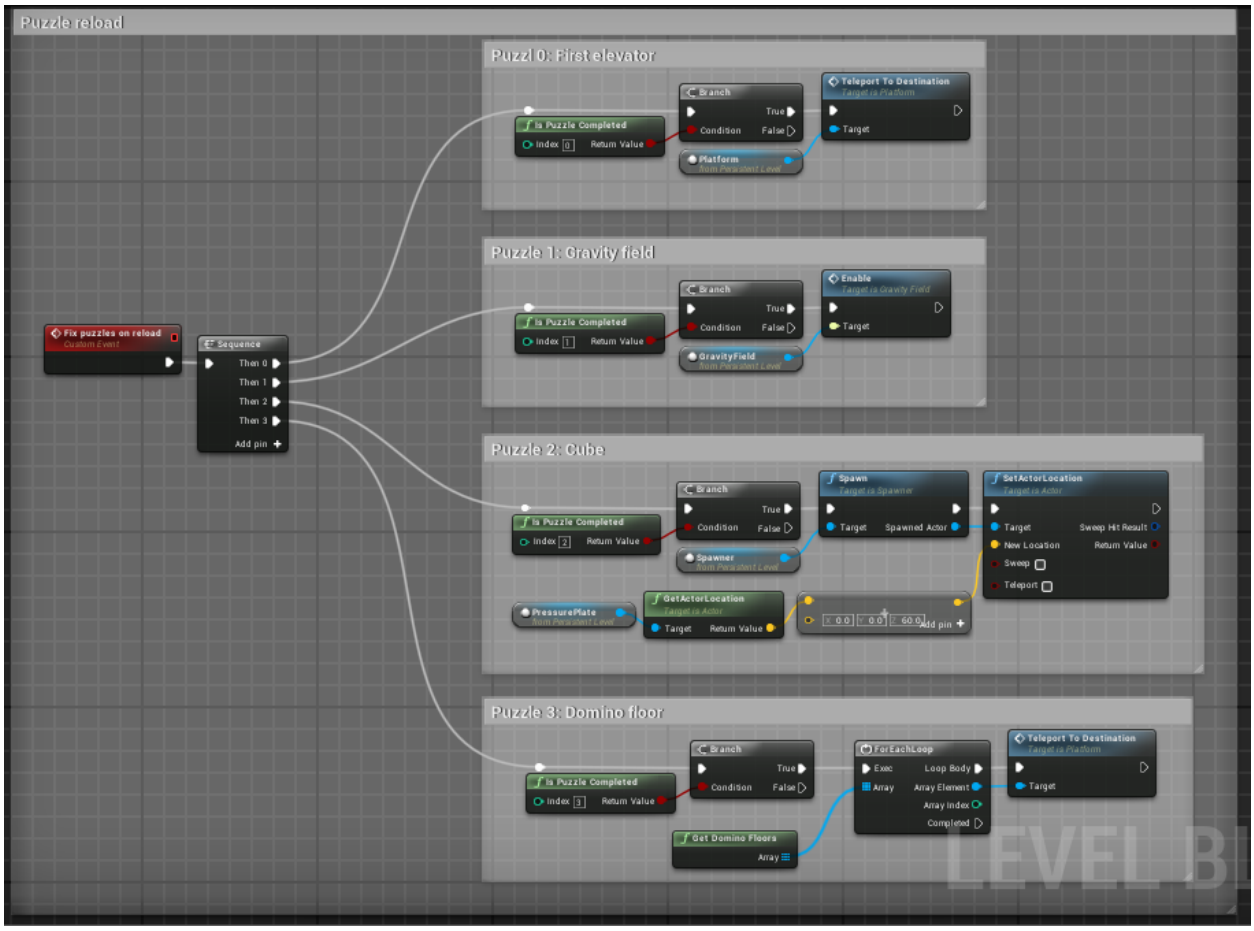
Kod 10.3 Funkcija OnLoad unutar ACodeLevelScript klase.

Izvor: vlastita izrada

Funkcija je vrlo jednostavna. IsFirstLoad se postavlja na laž, što označuje da se razina resetirala. Te se povećava brojač koji broji koliko puta se razina resetirala. Taj brojač

se koristi samo kako bi se prikazala statistika korisniku na samom kraju razine, te nije bitan za sustav zagonetki.

Sada se tijekom inicijalizacije znaju svi potrebni podatci. Poznato je da li se prvi put pristupilo razini ili je resetirana, te koja je zadnja riješena zagonetka. Sa tim podatcima svaka razina tijekom inicijalizacije može promijeniti stanja objekata tih zagonetki. Primjer je prikazan na sljedećoj slici.



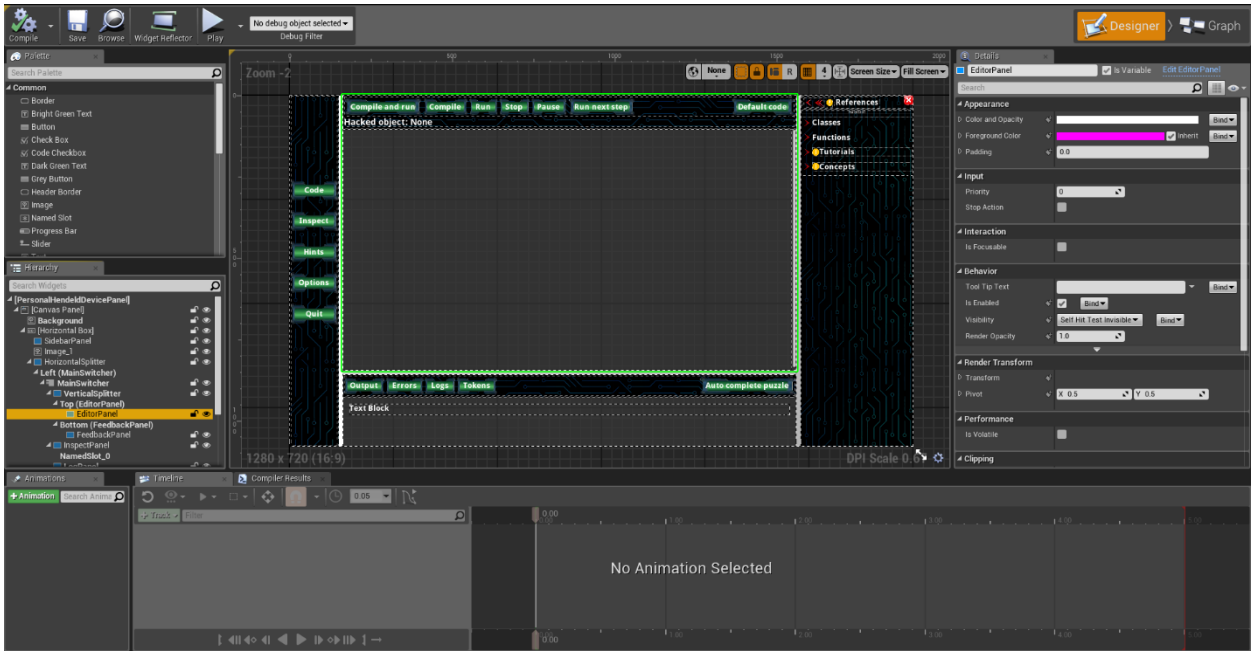
Slika 18. Primjer mijenjanja stanja zagonetki nakon resetiranja razine.

Izvor: vlastita izrada

Ova funkcija se pokreće tijekom inicijalizacije razine, te pozivanjem IsPuzzleCompleted funkcije provjerava da li je zagonetka sa zadanim indeksom riješena. Ukoliko je tada se zagonetka rješava. Samo rješavanje ovisi o zagonetki.

11 Grafičko sučelje

Za kreiranje grafičkog sučelja unutar Unreal Engine-a 4 se koristi grafički sustav zvan Unreal Motion Graphics (UMG). Sustav radi na način da se kreiraju takozvani Widget-i koji predstavljaju određene dijelove sučelja. Widget-i također mogu imati druge Widget-e kao djecu, što omogućuje da se svaki dio sučelja zasebno kreira te na kraju spoji u jednu cjelinu. To olakšava i činjenica da oni mogu sadržati i logiku, što omogućuje da svaki Widget bude samostalan, te da može komunicirati sa ostalima koristeći Observer.



Slika 19. Primjer UMG editora unutar Unreal Engine-a 4.

Izvor: vlastita izrada

Na slici je prikazan UMG editor koji trenutno editira Widget koji predstavlja cijelu konzolu koju korisnik koristi tijekom igranja igre. Sa lijeve strane je prikazana hijerarhija drugih Widget-a od kojih je sačinjena ova konzola.

Iako je potrebno prikazati puno podataka korisniku tijekom igranja igre, sam sustav grafičkog sučelja je relativno jednostavan. Koristi se standardna Model-View-Controller arhitektura, te samim time se cijeli sustav može podijeliti na ta tri dijela.

PlayerController predstavlja Model. Kako se prije prikazalo, svi podatci potrebni za kompajliranje i pokretanje koda se prvo šalju PlayerController-u. Ti podatci se tada koriste i za grafičko sučelje. Ostali potrebni podatci, poput Actor-a u kojeg igrač gleda i sl. su također dostupni preko PlayerController-a.

Widget-i predstavljaju View. Iako Widgeti sadrže logiku, ona ne modificira nikakve podatke, te služi samo za njihov prikaz.

Za sam Controller se koristi posebna klasa tipa UI_Controller. Ona je komponenta koja pripada PlayerController-u, te samim time ima brz pristup svim potrebnim podatcima. Jedina iznimka između standardne MVC arhitekture i ove implementacije je u tome što korisnik ne može direktno slati naredbe UI_Controller-u, pošto se sve naredbe registriraju unutar PlayerController-a. U tom slučaju PlayerController prosljeđuje sve relevantne naredbe direktno prema UI_Controlleru, te se od tamo pokreće potrebna logika kao bi se korisniku prikazali željeni podatci.

12 Zaključak

U radu je opisana motivacija iza izrade CodeBreaker igre, te na koji način ona pokušava olakšati učenje programiranja. Također je ukratko opisana usporedba između klasičnog pisanja koda i vizualnog programiranja, te su spomenute prednosti i mane oba načina. Prikazala se sama implementacija koda od konvertiranja teksta u tokene, preko kreiranja stabla apstraktne sintakse, sve do semantičke analize. Također je prikazan način pokretanja samog koda. Uz to su pojašnjeni i drugi sustavi u igru, poput sustava spremanja podataka, sustava zagonetki i grafičkog sučelja.

Iako su svi opisani sustavi dovoljni za završavanje igre, postoje neka ograničenja. Kod ne podržava skupove ni mape, nego isključivo polja kao apstraktne tipove podataka. Također ne podržava sučelja, što otežava implementaciju nekih zagonetki, te samim time i učenje određenih koncepata. Još jedno veliko ograničenje je nedostatak mogućnosti kreiranja Observer-a ili neke slične mehanike.

Sva ta ograničenja su mogućnosti da se CodeBreaker poboljša u budućnosti. Međutim kako bi se dodali ti sustavi potrebno je refaktorirati većinu koda. Tijekom refaktoriranja bi se mogli popraviti neki manje kritični problemi, koji bi olakšali daljnji napredak igre. Jedan od najvećih je način na koji se definiraju čvorovi stabla apstraktne sintakse. Trenutno su čvorovi zasebni objekti koji sadrže odgovarajuće podatke za dio koda koji predstavljaju. Problem nastaje u tome što su ti objekti jedinstveni, te više različitih objekata mogu predstavljati istu definiciju klase ili funkcije. Zbog toga se unutar koda nekad klase referenciraju preko objekta koji sadrži podatke, a nekad preko imena klase. Ta nekonzistentnost je postala problematična kada su vanjski sustavi, poput CodeComponent-a, počeli kreirati klase.

Još jedan veliki problem sa trenutnom arhitekturom koda je u činjenici da za semantičku analizu se koristi Double Dispatch. Ta metoda pozivanja funkcija je nepoželjna pošto kreiranjem novog tipa čvora stabla apstraktne sintakse je potrebno modificirati sve faze semantičke sintakse. Iz tog razloga je teško dodavati nove funkcionalnosti, pošto svaka izmjena može kreirati greške u semantičkoj analizi.

Parser je također još jedan dio koda koji bi se mogao poboljšati. Sam rad parsera je dobar, no problem se javlja kada je potrebno javiti grešku korisniku. Pošto se svaki dio koda zasebno parsira, ponekad je teško dobiti kontekst gdje se greška nalazi, što

dovodi do kriptičnih poruka. Činjenica da parser nema informacije o tome na kojoj liniji se nalazi određeni dio koda također dovodi do grešaka koje je ponekad teško pronaći.

Drugi sustavi igre se također mogu poboljšati. Među njima je sustav zagonetki. Trenutno svaka razina sadrži zagonetke, te se ona brine o njihovim stanjima tijekom resetiranja. Unreal Engine 4 sadrži sustav pod-razina, tj. sustav u kojem jedna razina može sadržavati druge razine. Koristeći taj sustav može se kreirati pod-razina za svaku zagonetku, te je samim time svaka zagonetka pravilno ukomponirana. Tada je dovoljno dodati te pod-razine u glavnu razinu. Ovaj sustav bi također uvelike pomogao tijekom iteracije razina, pošto bi se zagonetke mogle jednostavno prebacivati iz jedne razine u drugu.

Još jedan dio igre koji bi se mogao poboljšat je dodavanje novog načina kreiranja funkcija u CodeComponent-u. Trenutno se funkcije kreiraju definiranjem preko Blueprint klasa (kompleksne i jednostavne funkcije), ili definiranjem unutar Level Blueprint-a. Treći način definiranja funkcija bi bila mogućnost da dizajner upiše kod direktno u definiciju funkcije. Taj kod bi se zatim kompajlirao, te ponašao kao da ga je korisnik kreirao. Na taj način dizajner ne bi morao koristiti Blueprint za definiranje logike. Dodatna prednost je ta što bi se tada kod te funkcije mogao pokazati korisniku

Sa druge strane, postoji i mnogo načina kako bi se moglo poboljšati iskustvo korisnika tijekom igranja. Trenutno korisnik mora ući u neku razinu kako bi pročitao tutorijale i primjere svih koncepata koje je naučio tijekom igranja. Dodavanjem mogućnosti da se ti podatci prikazuju unutar početnog menija bi eliminirale potrebu da se učita cijela razina kako bi se oni pročitali.

Neke razine bi se mogle izmijeniti na način da se nadodaju dodatne zagonetke. Na njima se pojavljuju novi tutorijali odmah nakon završetka zagonetke. Postoji mogućnost da korisnik nije u potpunosti upoznat sa konceptom koji je koristio, a već mora naučit novi. Idealno bi za svaki koncept imao nekoliko zagonetki za riješit.

Međutim trenutno stanje CodeBreaker igre je dovoljno izrađeno da korisniku prikaže sve osnovne koncepte programiranja, te sadrži dovoljno razina kako bi se ti koncepti uspješno koristili. U slučaju da korisnik zaboravi neki koncept, ima opciju da ponovo pročita o njemu, te da pokrene kod koji dolazi sa tim podatcima. Ukoliko ni to nije dovoljno, uvijek postoji opcija da se razina ponovo odigra.

13 Literatura

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software.* 1st Edition. United States: Addison-Wesley.

Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D., (2006). *Compilers: Principles, Techniques, and Tools.* 2nd Edition. Pearson Education, Inc.

Bileta, T. (2019). *Dizajn igre za učenje koncepata programiranja u razvojnom okruženju Unreal Engine 4*. Fakultet informatike u Puli.

Maloney, J., Resnick M., Rusk., Silverman., Eastmond., (2010). *The Scratch Programming Language and Enviroment*, Massachusetts Institute of Technology.

Cooper S., (2010). *The Design of Alice*, Stanford University.

Službena stranica Unreal Engine-a 4, komentari u Blueprint sustavu. Dostupno na https://docs.unrealengine.com/en-us/Engine/Blueprints/UserGuide/Comments (Datum zadnjeg pristupa: 21.04.2019)

Steam stranica igre Glitchspace, primjer koda Glitchspace igre. Dostupno na https://store.steampowered.com/app/290060/Glitchspace/ (Datum zadnjeg pristupa: 21.04.2019)

14 Popis koda

Kod 4.1 Funkcija Tokenize u klasi Lexer. Pretvara tekst koda u listu tokena. 13
Kod 4.2 Funkcija Lex u Lexer klasi, koja analizira sekvencu znakova i vraća odgovarajući token 14
Kod 5.1 Funkcija ParseScope u klasi UParser. 17
Kod 5.2 Funkcija IsTokenOfClass iz klase UParser 18
Kod 5.3 Funkcija IsTokenOfClassAtIndex iz klase UParser 19
Kod 5.4 Funkcija GetRPN iz klase UParser. 20
Kod 6.1 Funkcija AnalyseAST iz klase USemanticAnalysis 21
Kod 6.2 Funkcija Analyse iz klase UAST_Main. 23
Kod 6.3 Funkcija StartAnalysis iz klase USAP_ClassAnalysis. 23

Kod 6.4 Funkcija Analyse iz klase USAP_ClassDefinition, koja analizira čvor definicije klase 24
Kod 6.5 Funkcija StoreChildParentRelationship iz klase USAP_ClassAnalysis. 25
Kod 6.6 Funkcija CheckForCircularDependency iz klase USAP_ClassAnalysis. 26
Kod 6.7 Funkcija CreateType iz klase USAP_ConvertVariableCallsToType. 28
Kod 6.8 Funkcija Analyse iz klase USAP_ConvertVariableCallsToType, koja analizira čvor tipa USAT_VariableCall 28
Kod 6.9 Funkcija Analyse iz klase USAP_ConvertVariableCallsToType, koja analizira čvor tipa USAT_Scope 29
Kod 6.10 Funkcija Analyse iz klase USAP_FunctionAnalysis, koja analizira čvor UAST_ClassDefinition 30
Kod 6.11 Funkcija Analyse iz klase USAP_FunctionAnalysis, koja analizira čvor tipa USAT_FunctionDefinition 31
Kod 6.12 Funkcija CheckTypeValidity iz klase USAP_FunctionAnalysis. 31
Kod 6.13 Funkcija CheckDuplicateFunctions iz klase USAP_FunctionAnalysis. 32
Kod 6.14 Funkcija CheckDuplicateArgumentName iz klase USAP_FunctionAnalysis. 33
Kod 6.15 Isječak koda iz klase USAP_OperatorAnalysis koji provjerava broj argumenata funkcije 35
Kod 6.16 Isječak koda iiz USAP_OperatorAnalysis klase koji provjerava da li je preopterećenje operatora definirano dvaput 36
Kod 6.17 Funkcija Analyse iz klase USAP_StoreStatics, koja analizira čvor tipa UAST_Main 37
Kod 6.18 Funkcija Analyse iz klase USAP_StoreStatics, koja analizira čvor tipa UAST_FunctionDefinition 38
Kod 6.19 Funkcjia Analyse iz klase USAP_StoreStatics, koja analizira čvor tipa UAST_VariableDefinition 39
Kod 6.20 Funkcija AsString iz strukture FExpressionReturn. 40
Kod 6.21 Funkcija Analyse iz klase USAP_ExpressionAnalysis, koja analizira čvor tipa UAST_If 42
Kod 6.22 Funkcija Analyse iz klase USAP_ExpressionAnalysis, koja analizira čvor tipa UAST_FloatLiteral 42
Kod 6.23 Funkcija Analyse iz klase USAP_ExpressionAnalysis, koja analizira čvor tipa UAST_MemberAccess 43

Kod 6.24 Isječak koda iz klase USAP_ExpressionAnalysis, koji prikazuje analizu čvora UAST_VariableCall, ukoliko je pozivatelj tip podatka. 44
Kod 6.25 Isječak koda iz klase USAP_ExpressionAnalysis, koji prikazuje analizu čvora UAST_VariableCall, ukoliko je pozivatelj implicitan. 46
Kod 6.26 Isječak koda iz klase USAP_ExpressionAnalysis, koji prikazuje analizu čvora UAST_VariableCall, ukoliko je pozivatelj instanca neke klase. 47
Kod 6.27 Funkcija StartAnalysis iz klase USAP_StaticAnalysis. 49
Kod 6.28 Isječak koda iz klase USAP_StaticAnalysis, koji analizira da li se statična varijabla nalazi unutar klase 49
Kod 6.29 Isječak koda iz klase USAP_StaticAnalysis, koji analizira da li se varijabla nalazi unutar globalnog bloka ili je lokalna 50
Kod 6.30 Isječak koda iz klase USAP_StaticAnalysis, koji analizira da li se varijabla nalazi u TopOwner-u 51
Kod 6.31 Funkcija Analyse iz klase USAP_FlowControlAnalysis, koja analizira čvor tipa UAST_For 51
Kod 6.32 Funkcija Analyse iz klase USAP_FlowControlAnalysis, koja analizira čvor UAST_Break 52
Kod 6.33 Funkcija StartAnalysis iz klase USAP_ReturnAnalysis. 53
Kod 6.34 Funkcija Analyse iz klase USAP_ReturnAnalysis, koja analizira čvor tipa UAST_If..... 54
Kod 6.35 Isječak koda iz klase USAP_ReturnAnalysis koji provjerava da li void funkcija vraća vrijednost, ili da li funkcija ne vraća vrijednost kada bi trebala..... 55
Kod 6.36 Isječak koda iz klase USAP_ReturnAnalysis koji provjerava da li funkcija vraća korektnu vrijednost 56
Kod 6.37 Isječak koda iz klase USAP_Final, koji provjerava da li postoji nadjačana funkcija koji je označena kao final 57
Kod 7.1 Funkcija Duplicate iz klase UValue, koja duplicira vrijednost. 58
Kod 7.2 Funckija RunNextRTSStep iz klase UCodeRunner. 60
Kod 7.3 Funkcija RTSStepComplete iz klase UCodeRunner. 61
Kod 7.4 Funkcija AddRTSToCodeRunner iz klase URuntimeState. 62
Kod 7.5 Funkcija RunStep iz klase URTS_VariableCall. 63
Kod 7.6 Funkcija RunStep iz klase URTS_MemberAccess. 64
Kod 7.7 Isječak koda iz klase URTS_FunctionCall koji traži vlasnika funkcije. 65
Kod 7.8 Funkcija iz klase URTS_FunctionCall koja pokreće argumente funkcije..... 66

Kod 7.9 Isječak koda iz klase URTS_FunctionCall koji poziva default konstruktor. . 67
Kod 7.10 Isječak koda iz klase URTS_FunctionCall koji poziva konstruktor definiran sa strane korisnika 68
Kod 7.11 Isječak koda iz klase URTS_FunctionCall koji pokreće instance URTS_ChainedFunctionCall klase 68
Kod 7.12 Isječak koda iz klase URTS_ChaineDFunctionCall, koji konvertira sve argumente na željene tipove 70
Kod 7.13 Isječak koda iz URTS_ChainedFunctionCall klase, koji pokreće funkciju sa konvertiranim parametrima..... 71
Kod 7.14 Isječak koda iz klase URTS_FunctionDefinition, koji pokreće kod funkcije. 72
Kod 7.15 Isječak koda iz klase URTS_FunctionDefinition, koji se pokreće ukoliko je funkcije void, te ne završava sa return 73
Kod 7.16 RunStep funkcija iz klase URTS_Return. 73
Kod 8.1 Funkcija AddClass iz klase ACodePlayerControllerBase..... 75
Kod 10.1 Isječak koda unutar Level Blueprint-a koji se pokreće tijekom pokretanja razine 89
Kod 10.2 Funkcija FirstLoadSetup unutar ACodeLevelScript klase. 90
Kod 10.3 Funkcija OnLoad unutar ACodeLevelScript klase. 90

15 Popis slika

Slika 1. Primjer dijela igrinog svijeta unutar korumpirane simulacije. 3
Slika 2. Primjer dijela simulacije unutar igre koja nije korumpirana. 4
Slika 3. Primjer hakiranja unutar CodeBreaker igre. 5
Slika 4. Primjer Alice vizualnog programskog jezika 6
Slika 6. Primjer petlji i grananja vizualnog programskog jezika Scratch. 7
Slika 7. Primjer komentara koji obuhvaća nekoliko blokova u programskom jeziku Blueprints 8
Slika 8. Programski kod igre Glitchspace. Simboli na blokovima predstavljaju različite tipove podataka 9
Slika 9. Primjer grananja u programskom jeziku Blueprints. 10
Slika 10. Kreiranje klase kao zaseban Blueprint. 77
Slika 11. Postavljanje unaprijed definirane klase u CodeComponent. 78

Slika 12. Kreiranje ugrađene klase unutar CodeDComponent-a. 79
Slika 13. Definiranje podataka kompleksne funkcije. 80
Slika 14. Definiranje logike kompleksne funkcije. 81
Slika 15. Definiranje logike jednostavne funkcije. 82
Slika 16. Postavljanje podataka ugrađene funkcije. 84
Slika 17. Povezivanje ugrađene funkcije i Level Blueprint-a preko Observer-a. 85
Slika 18. Definiranje logike ugrađene funkcije. 85
Slika 19. Primjer mijenjanja stanja zagonetki nakon resetiranja razine. 91
Slika 20. Primjer UMG editora unutar Unreal Engine-a 4. 92

16 Sažetak

Ovaj rad objašnjava razlog kreiranja CodeBreaker igre, te na koji način ona pomaže u učenju osnovnih koncepata programiranja. Uspoređuju se prednosti i mane između klasičnog pisanja koda i vizualnog programiranja, te se prikazuju razlozi odabira klasičnog pisanja koda za igru. Ostatak rada se fokusira na implementaciju same igre, pogotovo na implementaciju koda. U to spadaju Lexer, Parser, semantička analiza te samo pokretanje koda. Nakon toga se pokazuje na kako kod utječe igrin svijet, te kako rade ostali sustavi unutar igre.

17 Summary

This paper explains the reason behind the creation of the CodeBreaker game, and how it can help in learning basic programming concepts. It also compares advantages and disadvantages between classic written code and visual scripting, and explains why the classic written code was chosen for the game. The rest of the paper focuses on the implementation of the game, especially on the code implementation. This contains Lexer, Parser, Semantic analysis and the actual code running. After that it explains how the code can affect the game world, as well as explaining the rest of the game systems.

18 Ključne riječi

CodeBreaker, učenje uz igru, kompajler, Unreal Engine 4, programiranje, edukacija, video igre, podučavanje.

19 Keywords

CodeBreaker, educational games, compiler, Unreal Engine 4, programming, education, video games, teaching