Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva

Računalna igra „LABIRINT“

TEHNIČKA DOKUMENTACIJA

Mentorica: Prof.dr.sc. Željka Mihajlović

Voditelj projekta: Marko Vadlja

Projektni tim:

- Petar Dučić
- Matija Forko
- Davorin-Gordan Keserica
- Mirjana Ostojić
- Hrvoje Pađen
- Marina Tajić
- Marko Vadlja
- Stanislav Žužić

Sadržaj

Uvod

Računalna igra „Labirint“ zamišljena je u sklopu projekta predmeta Diplomski projekt na diplomskom studiju Fakulteta elektrotehnike i računarstva.

„Labirint“ je *multiplayer* igra čija se radnja odvija unutar generiranog labirinta. Glavni likovi su malena čudovišta koja imaju zadatak što prije doći do središta labirinta, uzeti kolač koji ih tamo čeka i donijeti ga do izlaza labirinta te predati malenim Čupavcima koji čuvaju izlaz labirinta. Tijekom putovanja labirintom likovi nailaze na *PowerUp* elemente koje mogu pokupiti ako žele te koji im mogu pomoći ili odmoći.

Projekt obuhvaća većinu komponenti koje su vezane uz izradu računalne igre. Uz rješavanje problema iz područja računalne grafike primjenjeni su i algoritmi iz umjetne inteligencije te osnovni principi interakcije čovjeka i računala.

Igra je izrađena za Microsoft Windows platformu, za jednog ili dva igrača, upotrebom tehnologija .NET 4.0 FRAMEWORK, Visual Studio 2010, XNA 4.0, Autodesk Maya 3D te je pisana u programskom jeziku C#.

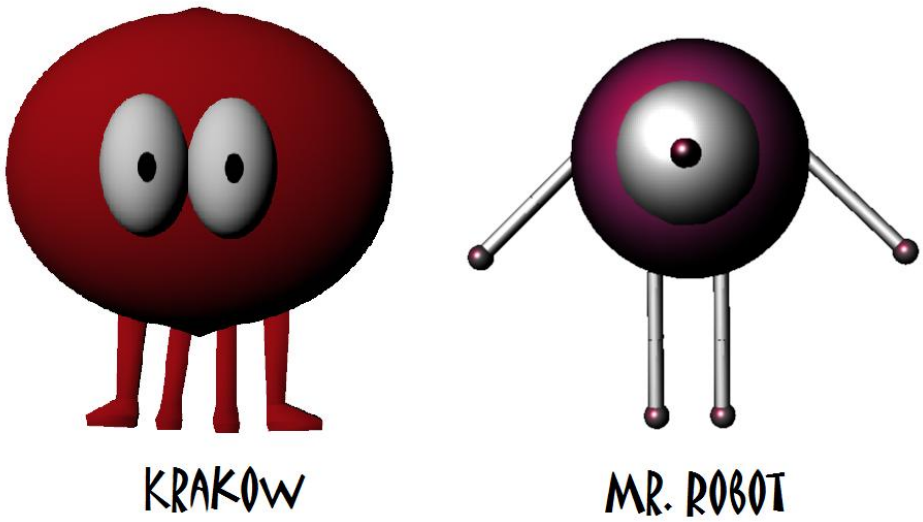
Izrada računalne igre „Labirint“

Za izradu računalne igre bilo je potrebno pobrinuti se za ostvarenje nekoliko komponenti: generiranje labirinta i pripadajućih tekstura, upravljanje kamerom, modeliranje i animacija likova, pokretač fizike i grafičko sučelje. Svaka komponenta opisana je u nastavku teksta.

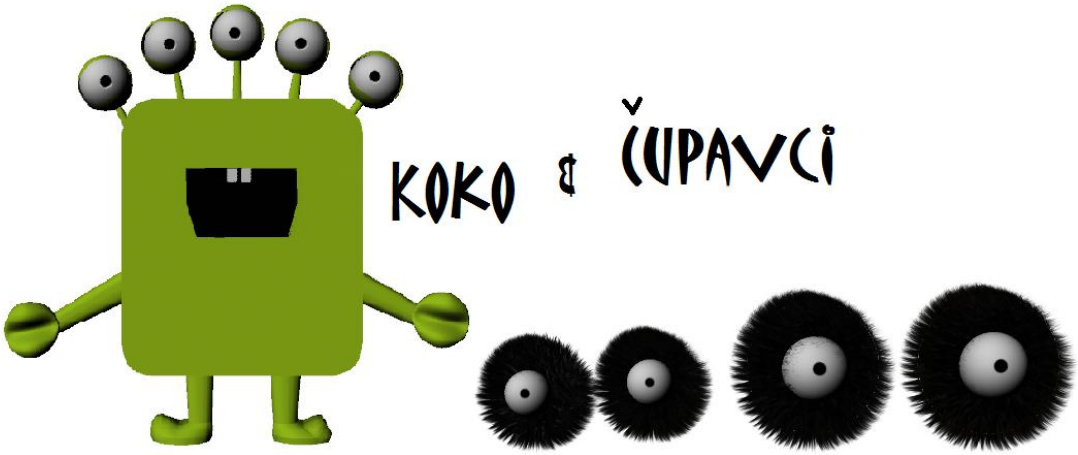
Modeliranje

Svi likovi korišteni u sklopu projekta *Labirint* izrađeni su pomoću popularnog softvera za 3D računalnu grafiku Autodesk Maya 3D. Tijekom dogovaranja detalja igre nisu bile zadane nikakve osnovne karakteristike koje likovi moraju posjedovati stoga je svaki lik neobičan sam po sebi i drugačiji od ostalih likova.

Likovi su izrađeni korištenjem osnovnih geometrijskih objekata poput kvadra, valjka i kugle, osnovnih transformacijskih funkcija translacije, rotacije i skaliranja te ugrađenih pomoćnih funkcija koji omogućuju rad sa stranama, bridovima i vrhovima objekta.



Slika 1. Likovi Krakow i Mr.Robot



Slika 2. Likovi Koko i Čupavci

Slika 3. Kolač koji igrač treba donijeti Čupavcima

Skeletna animacija

Za potrebe animiranja 3D likova projekt koristi tehniku skeletalne animacije. Kod tehnike skeletalne animacije model virtualnog lika sastoji od dva dijela:

1. Kože – površinske reprezentacije izgleda lika (geometrija lika)
2. Kostura – hijerarhijskog skupa kostiju koje se koriste isključivo za animaciju

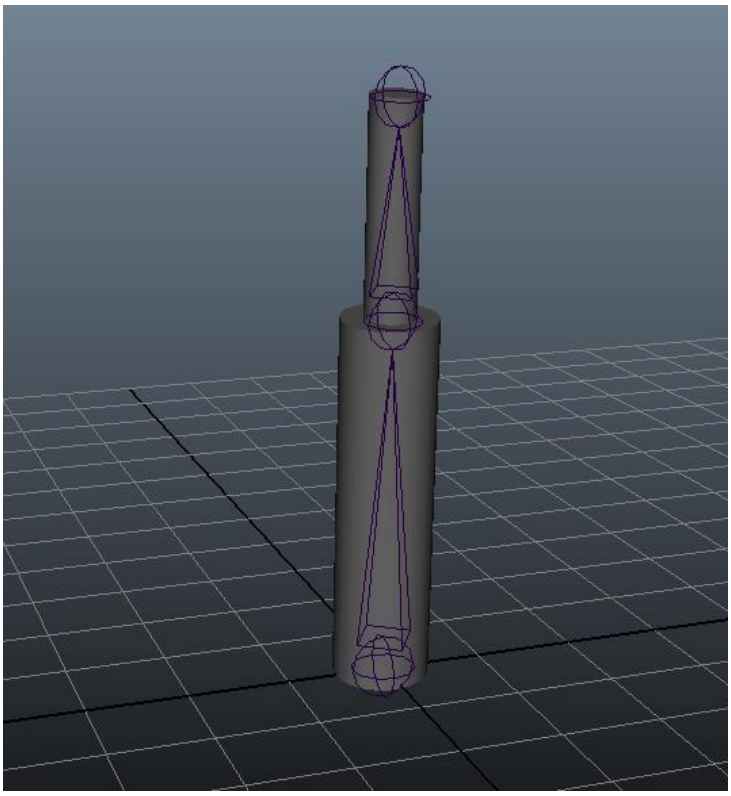
Kost je nevidljivi manipulator geometrije lika. Definirana je trodimenzionalnom transformacijom koja uključuje položaj, rotaciju i skalu, te kost roditelja. Rezultantna transformacija pojedine kosti je produkt transformacijskih matrica kosti roditelja i nje same. Na taj način pomicanje kosti (npr. bedrene) pomaknut će sve kosti koje se u hijerarhiji kostura nalaze ispod nje (npr. potkoljenica).

Velika prednost skeletalne animacije je što jedna kost može kontrolirati velik broj poligona, što animiranje čini mnogo jednostavnijim jer treba upravljati puno manjim skupom vrijednosti. Animator se može fokusirati na upravljanje velikim cjelinama lika (npr. noge, ruke, prsti, itd.) umjesto definiranjem promjene položaja za svaku točku posebno.

Postupak kreiranja kostura se na engleskom naziva *rigging*, a postupak spajanja kostiju s geometrijom lika *skinning*.

Animiranje lika

Kada je 3D lik modeliran i spojen sa svojim kosturom, spreman je za animiranje. U sljedećih nekoliko koraka opisuje se postupak skeletalne animacije u Mayi na vrlo jednostavnom primjeru robotske ruke, koja se sastoji od dva cilindrična dijela i dvije kosti.



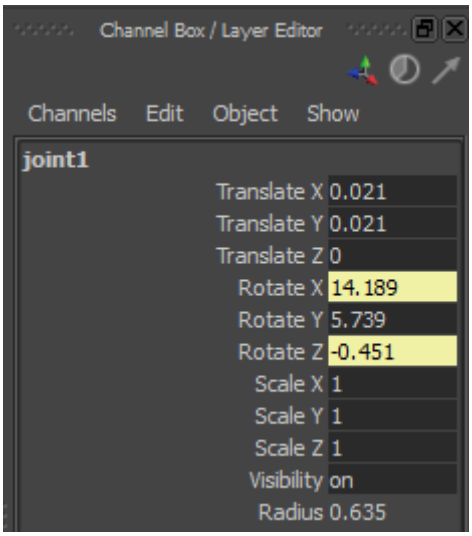
Slika 4. Robotska ruka

Priprema za animaciju

Najprije je potrebno odrediti koliko će animacija trajati, u broju sličica ili okvira (engl. *frames*). Standardni prikaz hodanja (engl. *walk cycle*) sastoji se od 33 sličica koje se ponavljaju, stoga se i za primjer robotske ruke koristi ta vrijednost, koja se definira na vremenskoj skali (engl. *timeline*).

Slika 5. Vremenska skala

Unaprijed je potrebno definirati skup atributa lika (engl. *character set*) koji će se mijenjati u tijeku animacije. Novi skup atributa (koji je na početku prazan) kreira se opcijom *Character -> Create Character Set*. Zatim je potrebno odabrati određenu kost, u prozoru opcija (engl. *channel box*) označiti sve atribute koji će se mijenjati te dodati označene atribute u skup (*Character -> Add to Character Set*). Ovaj postupak ponavlja se za sve kosti koje će se pomicati.



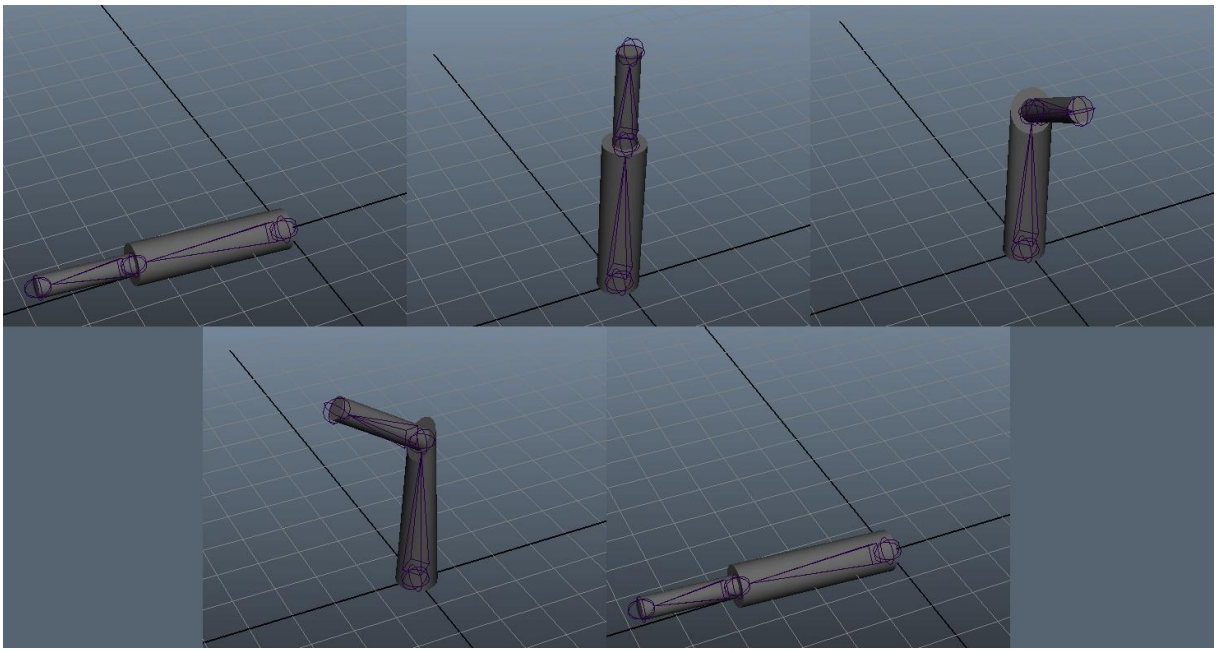
Slika 6. *Rotate X* i *Rotate Z* dodane su u skup atributa

Animacija

Postupak animiranja modela svodi se na definiranje položaja kostiju u ključnim vremenskim trenucima (engl. *keyframe*). Primjerice, ako na početku animacije robotska ruka treba biti u polegnutom položaju potrebno je označiti prvi trenutak u *timeline*-u (t = 1), promijeniti vrijednosti iz skupa atributa tako da se dobije željeni položaj i orijentacija robotske ruke, te postaviti zadane vrijednosti kao ključ (*Animate -> Set Key*).

Ako će se animacija ponavljati nakon što dođe do kraja (što je slučaj za prikaze hodanja) onda se mora namjestiti da posljednji trenutak u *timeline*-u (t = 33) bude jednak prvome. Ako se to ne podesi vjerojatno će se pojaviti nagli skokovi u položajima kostiju.

Zatim se kreiraju svi ostali *keyframe*-ovi potrebni za ostvarenje određene animacije. U ovom se primjeru robotska ruka podiže u uspravni položaj, nakon čega se pregiba u zglobu, rotira za 270° te vraća u početni položaj.



Slika 7. Prikaz svih *keyframe*-ova za animaciju robotske ruke

Izvoz animacije

Da bi dobivena animacija mogla biti prikazana u igri izrađenoj u XNA alatu potrebno ju je izvesti iz Maye u format FBX. Najprije treba učitati modul za FBX (engl. *plug-in*) u Mayu (*Window* -> *Settings/Preferences* -> *Plug-in Manager*) te izvesti čitavu scenu (*File* -> *Export All...*) u željenu datoteku.

Skeletna animacija i XNA

Osnove sustava XNA

XNA (engl. *XNA is Not an Acronym*) je Microsoftov paket alata namijenjen razvoju 2D i 3D igara. Prva verzija objavljena je 2006. godine a svake godine poslije predstavljena je sljedeća inačica. Verzija 4.0 objavljena je u rujnu 2010. godine te obuhvaća sve Microsoftove odjele vezane uz razvoj igara: XNA se koristi za izradu igara čija je ciljana platforma *Windows*, *Xbox 360*, *Zune* ili *Windows Phone 7*.

Struktura programskog rješenja u XNA sustavu

Programsko rješenje (engl. *solution*) sastoji se od tri glavna (*C#*) projekta:

1. Glavni projekt igre – sadrži opću logiku igre

2. Biblioteka igre (engl. *game library*) – sadrži sve potrebne klase i metode koje će se koristiti u ostala dva projekta
3. Biblioteka proširenog protočnog sustava sadržaja (engl. *content pipeline extension library*) – opcionalna komponenta; ako program zahtijeva posebnu obradu sadržaja (modeli, teksture, zvukovi, fontovi ili efekti) potrebno je stvoriti ovaj projekt s kojim se može odrediti specifični način obrade učitanog sadržaja

Struktura glavnog projekta

Pojednostavljeno, glavni projekt igre izvršava sljedeće: na početku izvršavanja programa inicijalizira se grafički sustav i vrše sve pripreme za daljnji rad. Sav sadržaj koji će program koristiti učitava se u funkciji *LoadContent*.

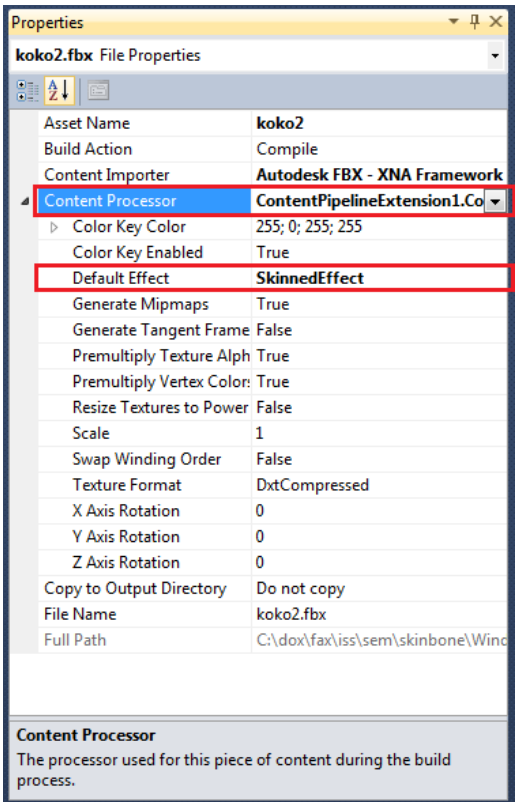
Zatim se pokreće glavna programska petlja koja se izvršava sve do završetka programa. Ona uključuje dvije glavne metode: *Update*, u kojoj se izvršava glavna logika igre (primjerice provjeravanje je li korisnik pritisnuo neku tipku te izvršavanje odgovarajućeg odgovora na tu akciju), i *Draw*, koja obuhvaća iscrtavanje virtualne scene i virtualnih likova na ekran.

Primjer skeletalne animacije za XNA

Na službenim internetskim stranicama za sustav XNA može se pronaći i skinuti projekt *Skinning Model* – demonstracija korištenja skeletalne animacije. U projektnom rješenju kreirani su projekti koji obavljaju zadaće biblioteke igre i biblioteke proširenog protočnog sustava sadržaja. Ti projekti su opće namjene, tj. mogu se iskoristiti za implementaciju bilo koje standardne skeletalne animacije.

Skeletalna animacija u sustavu XNA

Postupak implementacije skeletalne animacije u sustavu XNA započinje s uvozom animiranog modela u formatu FBX. Osim standardnog dodavanja novog sadržaja u projekt potrebno je za uvezeni model specificirati koji procesor sadržaja treba koristiti: bira se prošireni protočni sustav sadržaja, tj. novi projekt dodan u rješenje radi obrade animiranog modela. Također, potrebno je odrediti koji je podrazumijevani tip efekta za učitani model. Efekti u sustavu XNA su metode koje određuju na koji način će se neki objekt iscrtati te mogu uključivati razne opcije poput položaja i ostalih transformacija, sustava osvjetljenja, tekstura itd. Postoji više vrsta efekata od kojih se za ovu primjenu koristi *SkinnedEffect*.



Slika 8. Opcije modela

Skeletalna animacija zahtijeva posebnu obradu podataka prilikom učitavanja modela, koja je isprogramirana u projektu proširenog protočnog sustava sadržaja. Nadalje, taj projekt zahtijeva korištenje biblioteke koja opisuje osnovne principe skeletalne animacije, te je i nju potrebno dodati u programsko rješenje – ili u obliku C# projekta ili u formatu *DLL*.

Konačno, u glavni projekt igre potrebno je dodati podršku za skeletalnu animaciju. Navedeni koraci detaljnije su objašnjeni u sljedećim sekcijama.

Biblioteka za animaciju

Najvažnije metode koje opisuju rad skeletalne animacije zapisane su u biblioteci *SkinnedModel.dll*. Biblioteka je organizirana u četiri klase:

1. *Keyframe.cs*

   Definira položaj jedne kosti u jednom trenutku. Kost je određena cjelobrojnim indeksom, trenutak varijablom tipa *TimeSpan* (količina vremena protekla od početka animacije) a položaj kosti je definiran transformacijskom matricom.
2. *AnimationClip.cs*

   Sadrži sve *keyframe*-ove koje uključuje neka animacija, u obliku liste. Dodatno sadrži ukupno trajanje animacije.

3. *SkinningData.cs*

   Sadrži skup od jedne ili više animacija organiziranih u strukturi *dictionary<string, AnimationClip>* – drugim riječima, svaka animacija može se pozvati svojim imenom. Osim skupa animacija, klasa sadrži i neke informacije o kosturu – za svaku kost u modelu definirane su: matrica za resetiranje kosti u početnu pozu (engl. *bindpose*), inverz te matrice i hijerarhija kostura (indeks roditeljske kosti).

4. *AnimationPlayer.cs*

   Kontrolna klasa zadužena za upravljanjem animacije; sadrži informaciju koja je animacija (*AnimationClip*) trenutno odabrana, koje je trenutno vrijeme u animaciji i koji je trenutni *keyframe*. Također sadrži podatke o kosturu koji se preuzimaju iz *SkinningData* klase.

Klasa *AnimationPlayer* posjeduje i nekoliko metoda od kojih je najvažnija *UpdateBoneTransforms*: metoda dobije (u obliku parametra) podatak koliko je vremena proteklo od prošlog osvježavanja animacije i na temelju njega i trenutno odabranog *AnimationClip*-a određuje nove položaje kostiju (tj. nove transformacijske matrice).

Metoda najprije provjerava je li trenutno vrijeme premašilo ukupno trajanje animacije. Ako jest, animacija kreće ispočetka: trenutno vrijeme i trenutni *keyframe* se postavljaju na početak, a kosti se resetiraju na početni položaj (*bindpose*).

Ako animacija nije došla do kraja onda se pronalaze svi *keyframe*-ovi koji se pojavljuju u vremenu proteklom od prošlog osvježavanja animacije. *Keyframe*-ovi se upotrebljavaju tako da se njihova transformacijska matrica primijeni na kost na koju pokazuju. Ako se u proteklom vremenu nalazi više *keyframe*-ova koji se odnose na istu kost primjenjuje se samo najnoviji *keyframe*.

Nadalje, dobivene transformacijske matrice kostiju se zatim proslijeđuju funkciji *UpdateWorldTransforms*, koja svakoj kosti izračuna matricu *worldTransform* kao umnožak prethodno dobivene *boneTransform* matrice i *worldTransform* matrice roditeljske kosti, tako da se svaka kost pozicionira na ispravan položaj u odnosu na cjelokupni kostur.

Konačno, izračunate matrice se šalju funkciji *UpdateSkinTransforms*, koja ih množi s inverzom matrice *bindpose* i rezultat zapisuje u matricu *skinTransform* za svaku kost.

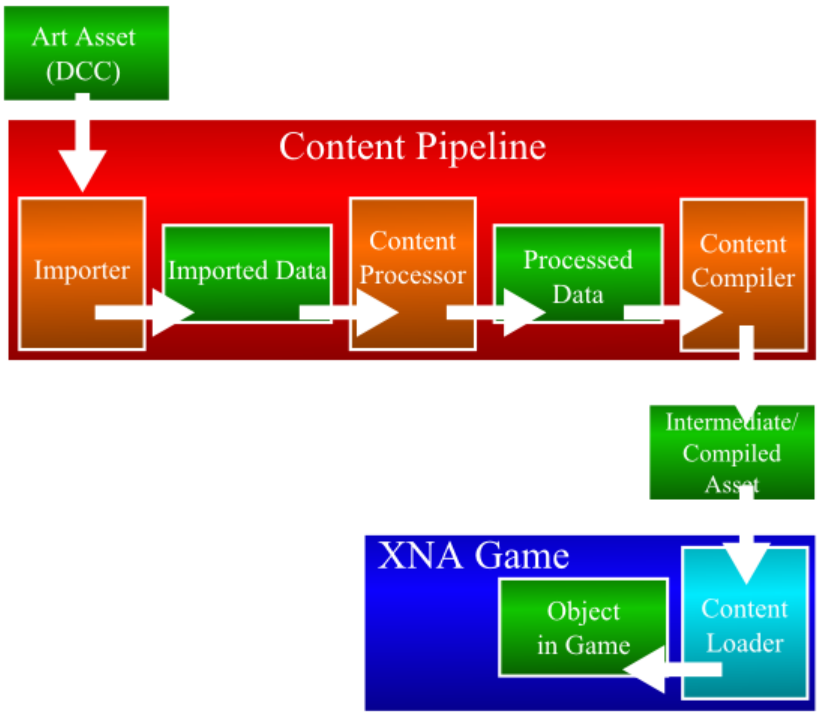
Prošireni protočni sustav sadržaja

Prilikom učitavanja animiranog modela u formatu FBX potrebno je prenijeti animaciju u oblik pogodan za upravljanjem putem *SkinningData* biblioteke. Obrada

sadržaja obavlja se prije samog pokretanja aplikacije, za vrijeme prevođenja programskog rješenja.

Protočni sustav sadržaja čine četiri faze:

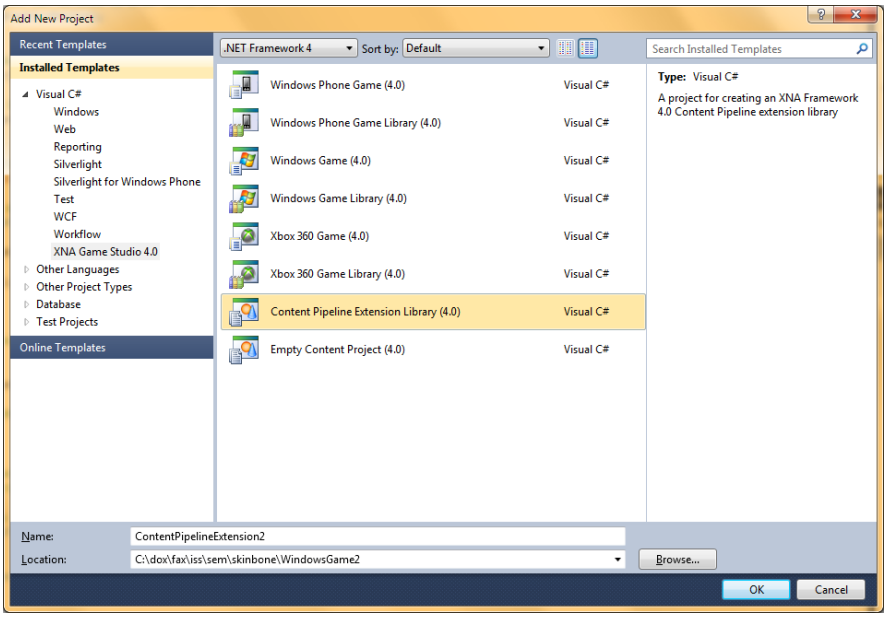
1. Učitavanje (engl. *importer*) – datoteka sa sadržajem (koji se ponekad naziva *DCC* – engl. *Digital-Content Creation*) se učitava i rezultat se proslijeđuje sljedećoj fazi.
2. Procesor sadržaja (engl. *content processor*) – preuzima učitani sadržaj u sirovom obliku i obrađuje ga u smisleni format koji igra može prepoznati.
3. Prevoditelj sadržaja (engl. *content compiler*) – obrađeni podaci spremaju se u prevedene datoteke koje služe kao izvor sadržaja za igru. Te su datoteke binarnog formata i najčešće imaju ekstenziju *XNB*.
4. Učitavanje sadržaja (engl. *content loader*) – ova faza nije dio samog protočnog sustava, već se događa kad pokrenuta igra počne učitavati sadržaj. Originalne datoteke sadržaja više nisu potrebne; sav se sadržaj učitava iz prethodno obrađenog posredničkog formata (XNB).



Slika 9. Faze protočnog sustava sadržaja

Iako XNA dozvoljava modificiranje bilo koje faze protočnog sustava sadržaja, za potrebe skeletalne animacije nužno je proširiti samo drugu fazu: procesor sadržaja. To se obavlja dodavanjem novog projekta tog tipa (engl. *template*) i proširivanjem određenih dijelova, od kojih je najvažnija funkcija *Process*: ona putem parametra dobiva tek učitani sadržaj (u objektu tipa *NodeContent* – bazna klasa za grafičke

podatke koji definiraju lokalni koordinatni sustav) i nakon obrade vraća gotov model (*ModelContent*).



Slika 10. Dodavanje projekta za proširivanje protočnog sustava sadržaja

Funkcija *Process* najprije pronađe podatke o kosturu iz učitanog sadržaja. Kostur je hijerarhijski organiziran pa ga je potrebno „spljoštiti“ (engl. *flatten*) tako da su kosti spremljene u običnoj listi. Za svaku se kost određuju standardne informacije: matrica za resetiranje kosti u početnu, inverz te matrice i indeks roditeljske kosti. Zatim se izgrađuje mapa kostiju u obliku rječnika koji povezuje ime neke kosti s njenim indeksom.

Nakon što je kostur obrađen treba konvertirati animacije u pogodan oblik. Animacije u formatu FBX sastoje se od više kanala; jedan kanal upravlja jednom kosti. Proces obrade eliminira kanale tako da izdvoji sve *keyframe*-ove iz svih kanala i spremi ih u običnu listu. Lista se zatim sortira po vremenu.

Dobiveni podaci (liste *keyframe*-ova, *bindpose* matrica, njen inverz i hijerarhija kostura) spremaju se u novu instancu *SkinningData* klase, koja se pohranjuje kao oznaka (engl. *tag*) u rezultantnom modelu.

Modifikacija glavnog projekta igre

Skeletalna animacija postiže se proširivanjem glavnog projekta na tri mjesta. Najprije, neposredno nakon učitavanja modela u *LoadContent* funkciji, podaci o animaciji se izdvajaju iz oznake modela. Na temelju tih podataka kreira se nova instanca klase *AnimationPlayer* (definirane u biblioteci za animaciju), bira se jedna od animacija i pokreće se: ona se postavlja kao odabrana animacija u *AnimationPlayer*-u, trenutno vrijeme i trenutni *keyframe* se postavljaju na vrijednost nula, a položaj kostiju se transformira u početni položaj na temelju *bindpose* matrica.

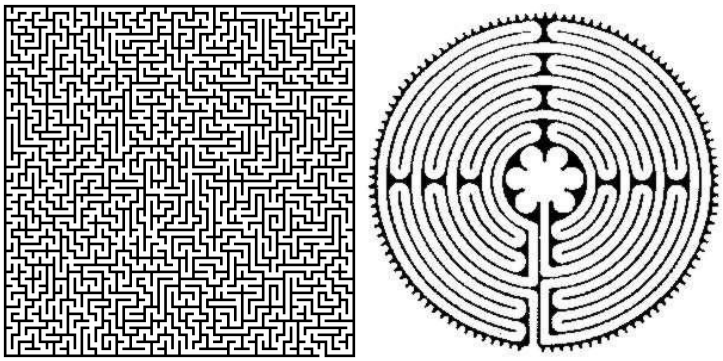
Zatim je potrebno osigurati redovito osvježavanje animacije. U funkciji *Update* (koja se stalno poziva za vrijeme trajanje igre) treba pozvati funkciju za osvježavanje animacije u *AnimationPlayer*-u i predati joj u obliku parametra količinu proteklog vremena od prošlog osvježavanja. Metoda je detaljno objašnjena u sekciji koja opisuje biblioteku za animaciju.

Konačno, u funkciji *Draw* potrebno je dohvatiti trenutno aktivne matrice *skinTransform* iz *AnimationPlayer*-a te, za svaki *SkinnedEffect* u modelu koji se iscrtava, primijeniti *skinTransform* pozivom funkcije *SetBoneTransforms*.

Labirint

Razlikujemo dvije vrste labirinta:

1. labirint s pravokutnim zidovima (engl. *maze*) i
2. labirint sa zakrivljenim zidovima (engl. *labyrinth*)



Slika 11. Vrste labirinta: *maze* (lijevo) I *labyrinth* (desno)

Za potrebe projekta odabran je labirint s pravokutnim zidovima. Rad s takvom vrstom labirinta jednostavniji je u nekoliko pogleda: jednostavnija je navigacija unutar labirinta i jednostavniji su algoritmi za generiranje i iscrtavanje labirinta.

Generiranje labirinta

Za generiranje labirinta potrebljen je algoritam pretraživanja u dubinu (engl. *Depth First Search*, DFS). Osnovna klasa koja se koristi je klasa *Cell,* ćelija.

```
Stvori stog StogCelija koji ima popis svih ćelija
Postavi BrojCelija = broj ćelija u StogCelija
Postavi TrenutnaCelija = nasumična ćelija iz StogCelija
Postavi BrojPosjecenihCelija = 1

Dok BrojPosjecenihCelija < BrojCelija
   Pronađi sve susjede od  TrenutnaCelija sa netaknutim zidovima
               Ako barem jedan nađena
                     Izaberi nasumično jednu
                     Sruši zid između nje i TrenutnaCelija
                     Stavi TrenutnaCelija  na StogCelija
                     Izabranu ćeliju upiši u TrenutnaCelija
                     Poveća BrojPosjecenihCelija za 1
```

```
Inače
        Makni ćeliju sa vrha stoga stoga  StogCelija
        Upiši ju u TrenutnaCelija
```

Algoritam DFS

Popis svih ćelija inicijaliziranih na taj način tvori labirint. Slika labirinta koja se može primjeniti za iscrtavanje labirinta kreira se implementiranom metodom *GenerateTexture* koja koristi navedeni popis ćelija.

Struktura labirinta

Generirani labirint predstavljen je binarnim zapisom te pohranjen u sliku. Slikovni elementi crne boje predstavljaju element zida, a bijeli predstavljaju praznine. Kako bi se omogućila 3D reprezentacija labirinta potrebno je kreirati odgovarajuću podatkovnu strukturu labirinta.

Glavna potreba strukture labirinta jest vizualna reprezentacija stoga se struktura labirinta kreira iz 2D slike labirinta. Za kreiranje 3D modela labirinta potrebno je definirati zidove i njihovu debljinu te pritom paziti kako bi konačna reprezentacija bila što realnija.

Kako bi se poboljšala realnost samih zidova implementirana je metoda *normal mapping* koja naglašava izbočine na zidovima uz pomoć *normal* mape, a riješena je uz pomoć osjenčivača piksela (engl. *Pixel shader*). Ova metoda ne opterećuje geometrijsku fazu te je broj poligona isti, ali imamo dojam da je objekt puno složeniji te samim time djeluje realističnije.

Uvođenje umjetne inteligencije te omogućavanje prolaska računala kroz labirint zahtjeva logičku reprezentaciju labirinta. Računalo mora moći prolaziti kroz labirint, birati smjerove, putanje i imati svrhu svojeg gibanja.

Grafička reprezentacija labirinta

Labirint se sastoji od zidova i poda. Na početku grafičke reprezentacije potrebno je iz 2D slike labirinta odrediti segmente zidova. Zidovi u labirintu orijentirani su horizontalno i vertikalno te imaju svoju konačnu duljinu.

Kako bi se dobila vektorska reprezentacija zidova potrebno je izdvojiti horizontalne i vertikalne segmente zidova. Minimalna duljina segmenta zida je jedan slikovni element koji se u 3D reprezentaciji može preslikati u bilo koje mjerilo. Pretvorba iz slike u vektorsku strukturu gotova je kada se iterativnim postupkom prođe kroz sve slikovne elemente te odrede svi horizontalni i vertikalni segmenti zidova.

Nakon uspješno dobivene liste segmenata zidova, potrebo je svakom segmentu zida implementirati način njegovog iscrtavanja.

Iscrtavanje zidova

Segment zida reprezentiran je svojim početnim i krajnjim položajem iz kojih se izvode duljina i smjer zida, a vrijednosti poput debljine i visine zida potrebno je definirati. Za svaku stranicu zida potrebno je definirati 5 bridova. Brid se definira pomoću 2 trokuta pa se konačna struktura jednog segmenta zida sastoji od 10 trokuta.

Unutar klase *Quad*, koja se definira za svaki brid, definirane su vektorske pozicije vrhova brida te spremnik vrhova sa svim informacijama potrebnim za iscrtavanje brida. Prilikom poziva iscrtavanja labirinta prolazi se kroz sve segmente zidova i poziva se metoda *Draw* koja iscrtava sve bridove kojima je opisan pojedini segment. Implementirana je i metoda *back face culling* kako se ne bi iscrtavali stražnji poligoni čime se ubrzava cijeli protočni sustav jer mu se šalje manji broj poligona.

Reprezentacija zidova poboljšava se i dodavanjem odgovarajuće teksture na svaki brid zida. Dodatno poboljšanje same reprezentacije dobiva se implementiranjem metode *normal mapping* koja realistično prikazuje izbočenja i deformacije na plohama zida.

Iscrtavanje poda

Pod labirinta definiran je kao pravokutnik, na isti način kao i pojedini brid segmenta zida. Na njega se „lijepi“ tekstura poda i pristupa metodi *normal mapping* kako bi pod labirinta izgledao što realističnije.

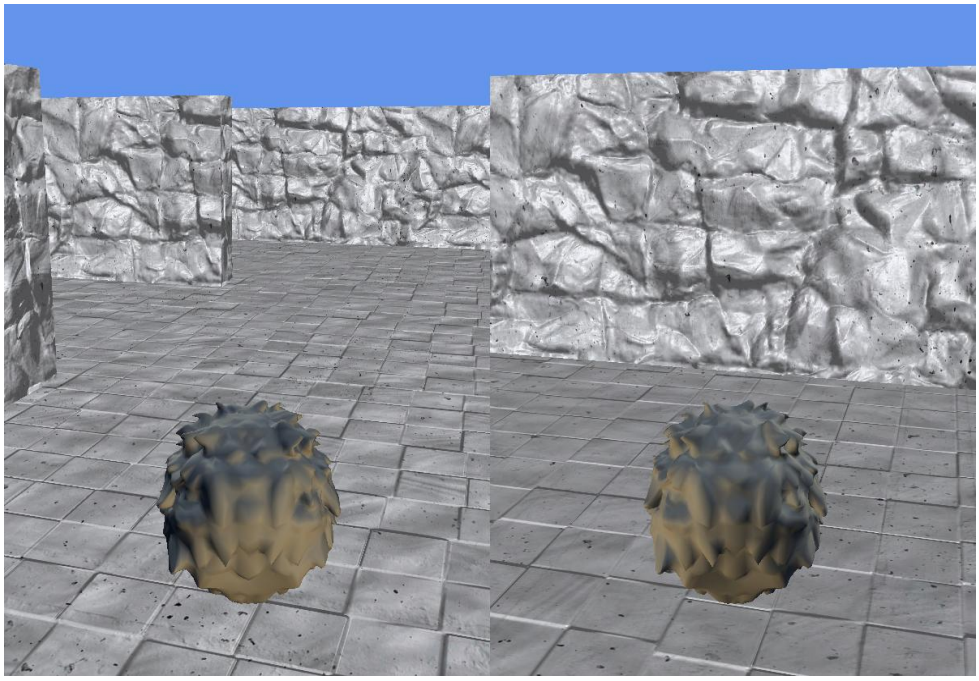
Iscrtavanje izobličenja pomoću metode *normal mapping*

Metoda *normal mapping* koristi se za poboljšanje reprezentacije labirinta. Njome se omogućava jednostavna implementacija i prikaz izbočina ravnog dijela površine pomoću *normal* mape koja se dodaje i koristi u fazi rasterizacije.

Normal mapa je tekstura koja sadrži informacije o deformaciji normale na pojedinom slikovnom elementu teksture korištenjem komponente boje za prikaz vektora deformacije normale. Prilikom rasterizacijske faze unutar osjenčivača piksela, informacije o deformaciji normale se kombiniraju sa normalom površine i s izvorom svjetla te se određuje boja pojedine točke na poligonu. Na taj način difuzna komponenta svjetla stvara dojam da ravni objekt koji vidimo ima izobličenja koja odgovaraju zapisu u *normal* mapi.

Dakle, na konačni izgled zida djeluje tekstura zida, globalno osvjetljenje i *normal* mapa u kombinaciji sa difuznim osvjetljenjem. Na taj način dobivamo iluziju da je objekt zakrivljen iako u stvarnosti nije.

Metoda *normal mapping* je jedan od trikova kojima se može zaobići kompliciranost i složenost kreiranja izobličene geometrije. Za većinu slučajeva, gdje su izobličenja dovoljno mala, konačan ishod je isti.

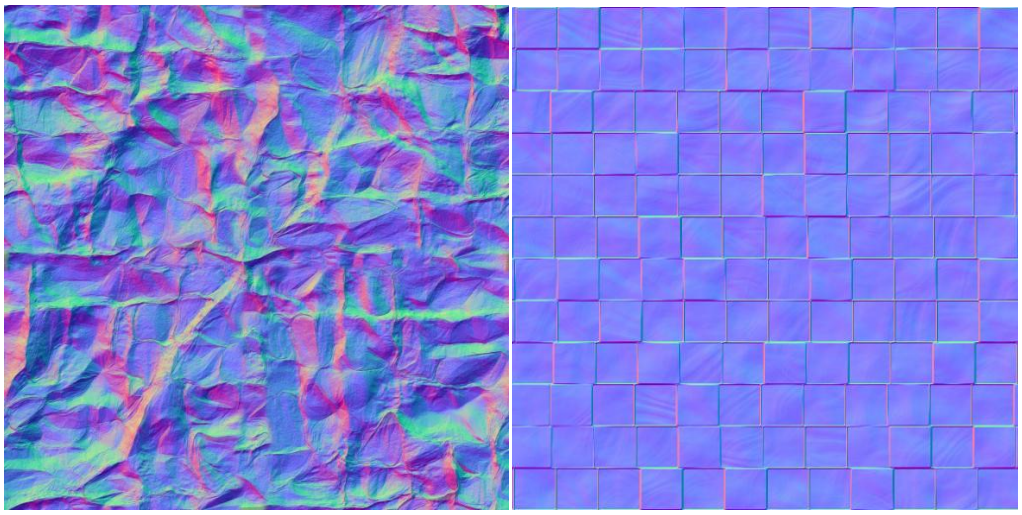


Slika 12. Prikaz efekta koji se dobiva metodom *normal mapping* (zidovi i pod su obične plohe).

Nedostatak ove metode javlja se prilikom gledanja u plohu na kojoj je implementirana *normal* mapa pod malim kutom jer se tada neuspješno prikazuju izobličenja i efekt 3D reljefa se gubi, odnosno, vidi se samo ravna ploha.

Osjenčivač piksela koji je korišten u ovom dijelu je preuzet sa Interneta te modificiran kako bi se prilagodio programu – prilagoditi opis vrhova i definirati ulazne vrijednosti za osjenčivač. Ulazni podaci se jednostavnim algoritmom obrađuju u fazi rasterizacije te se dobiva konačna boja pojedinog slikovnog elementa.

Struktura koja se šalje osjenčivaču piksela sadrži položaj točke, koordinate teksture, normalu i tangentu. Također, u osjenčivač ulaze sve informacije o svjetlu, tekstura zida, te *normal* mapa koja sadrži deformacije normale na pojedinoj lokaciji teksture tj. njegovom slikovnom elementu.



Slika 13. Primjer izgleda *normal* mape, lijeva se koristi za zidove, a desna za pod.

Umjetno inteligentni protivnik

Kreiranje umjetno inteligentnog protivnika

Dobro definirana struktura labirinta nužna je prilikom kreiranja umjetno inteligentnog protivnika. Umjetno inteligentni protivnik ne koristi vid kako bi se kretao po labirintu, odnosno, nije mu dovoljna samo 2D ili 3D reprezentacija labirinta već struktura koju računalo može pročitati i omogućiti umjetno inteligentnom protivniku kretanje kroz labirint, pamćenje putanje i dolazak do konačne lokacije unutar labirinta.

Kreiranje odgovarajuće strukture

Najprije je potrebno kreirati moguće putanje unutar labirinta. Putanje se kreiraju jednako kao i kreiranje segmenata zidova unutar labirinta osim što se kao elemente koji grade segmente puta ne promatraju zidovi već praznine. Nakon što su izrađeni horizontalni i vertikalni segmenti putanje izrađuje se logička reprezentacija labirinta, odnosno, graf labirinta koji umjetno inteligentni protivnik može iskoristiti u svojim izračunima.

Graf labirinta sastoji se od čvorova i poveznica čvorova gdje svaki čvor ima informacije o susjednim čvorovima i poveznicama tih čvorova. Čvorovi predstavljaju raskrižja unutar labirinta a za njihovo određivanje potrebno je pronaći sva presjecišta horizontalnih i vertikalnih putanja. Postoje i dodatni čvorovi (rubni čvorovi), tj. početni i završni čvorovi svakog segmenta puta koji nisu morali biti u presjeku s drugim

segmentom puta. Svaki čvor može imati maksimalno četiri susjedna čvora (lijevi, desni, gornji i donji) za koje se bilježi da su susjedi pojedinog čvora.

Kreiranje strukture labirinta je završeno kada struktura labirinta sadrži listu zidova, puteva i čvorova.

Implementacija algoritma umjetno inteligentnog protivnika

Umjetno inteligentni protivnik se inicijalno nalazi na početnom čvoru koji je slučajno odabran. Iz početnog čvora računalo rekurzivnim pretraživanjem u širinu bira najbolji smjer kojim će se umjetno inteligentni protivnik kretati. Za svaki put poznata je dobrota dobivena nekom kriterijskom funkcijom te se za najbolji put bira onaj koji ima najveću dobrotu. Ulaskom u novi čvor ponavlja se algoritam pretraživanja u širinu i bira se put s najvećom dobrotom, odnosno sljedeći čvor u koji će doći umjetno inteligentni protivnik.

Algoritam rekurzivnog pretraživanja u širinu pretražuje sva moguća rješenja do određene dubine. Inicijalno je dubina postavljena na vrijednost 10 što znači da algoritam pregledava do maksimalno deset raskrižja ispred trenutnog te na temelju poznatih informacija odredi najbolji smjer izlaska iz trenutnog čvora. Algoritam reducira moguća rješenja time što pamti već poznate (viđene) čvorove u svakoj rekurzivnoj iteraciji i ne ponavlja ih ako ih se već obišlo u prethodnim iteracijama. Takvim reduciranjem mogućih puteva smanjuje se složenost pretraživanja labirinta i omogućava dolazak do većih dubina u manjem vremenskom roku.

Funkcija cilja osigurava umjetno inteligentnom protivniku prolazak cijelog labirinta tako da prije prijeđe sve neviđene dijelove labirinta nego da opet posjeti dijelove koji su već viđeni. Funkcija cilja zbraja udaljenosti i dobrote pojedinih čvorova na putanji te se kao najbolja putanja, odnosno, smjer iz trenutnog čvora, bira najmanja vrijednost puta. Pri izlasku iz poznatog čvora i odabira sljedećeg, poznatom čvoru se povećava vrijednost čime osiguravamo da će algoritam uvijek izabrati putanju koja ima više nepoznatih ili manje posjećenih čvorova (njihova vrijednost je manja pa će i konačna vrijednost putanje biti manja!).

Grafička reprezentacija umjetno inteligentnog protivnika

Umjetno inteligentni protivnik je predstavljen transparentnim modelom duha koji ne vrši koliziju s objektima na sceni. Kada umjetno inteligentni protivnik prođe kroz igrača dolazi do negativnih efekata na igrača kroz koji je duh prošao, poput smanjenja brzine kretanja igrača.

Prilikom prolaska umjetno inteligentnog protivnika iz trenutnog čvora u novi čvor potrebno je izvršiti interpolaciju kretanja od čvora do čvora određenom brzinom kretanja u definiranom smjeru prema sljedećem čvoru. Nakon interpolacije kretanja

potrebno je izvršiti interpolaciju rotacije umjetno inteligentnog protivnika unutar čvora s obzirom na sljedeću lokaciju.

Postoje dvije vrste interpolacije skretanja: skretanje za 90° i skretanje za 180°.

Interpolacija skretanja za 90° se vrši linearno s gibanjem pa se tako umjetno inteligentni protivnik kreće i rotira po imaginarnoj kružnici unutar raskrižja sve dok se ne orijentira na novi smjer. Kod interpolacije skretanja za 180° (vraćanje putem od kojeg je došao) umjetno inteligentni protivnik se rotira oko svoje osi određenom brzinom rotacije.

PowerUp objekti

Namjena *PowerUp* objekata u računalnim igrama je davanje nepravedne kratkotrajne prednosti jednom od igrača (ili računalu).

U igri “Labirint” *PowerUp* objekti implementirani su na način da će igrač imati 50% šanse dobiti kratkotrajnu prednost ili kratkotrajnu štetu ako pokupi *PowerUp* objekt. Primjerice, ako igrač pokupi određeni *PowerUp* objekt jednaka je vjerojatnost da će prednost dobiti on ili njegov suparnik.

Tipovi *PowerUp* objekata ostvarenih u igri:

- Čizma – nemogućnost kretanja naprijed ili nazad
- Skok – nemogućnost skoka
- Upitnik – kontrole kretanja se izmiješaju
- Žarulja – zamračenje svjetla
- Kompas – prikaz mini mape u uglu ekrana

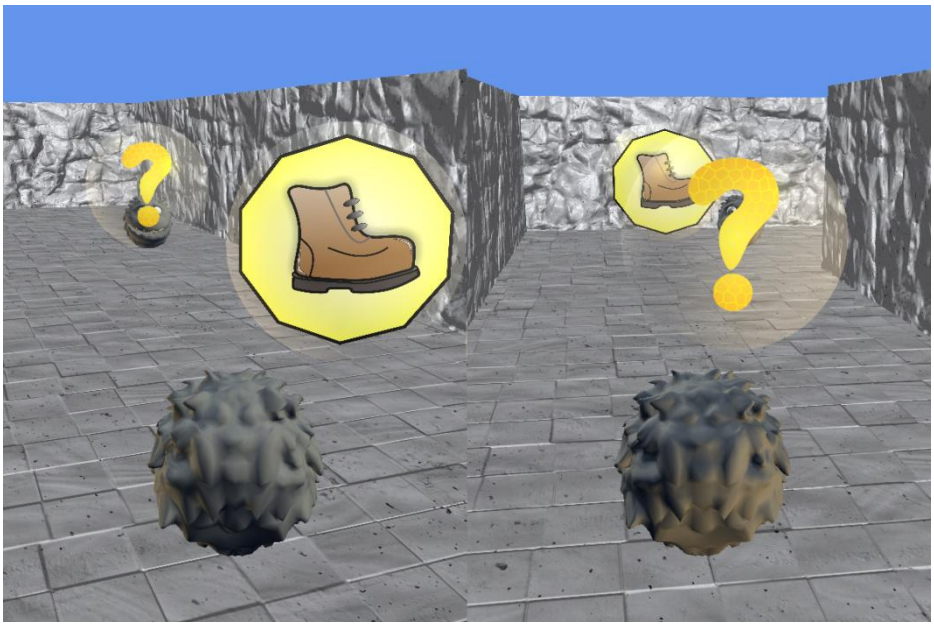
Stvaranje slučajnog događaja prikupljanjem *PowerUp* objekta započinje detekcijom sudara sa *PowerUp* objektom. U klasi *Player* postoji metoda *AddPowerUp* koja se pokreće detekcijom sudara i predaje joj se instanca klase *PowerUp* s kojom se igrač sudario.

Klasa *Player* sadrži listu svih *PowerUp* objekata koji mogu djelovati na igrača. Temeljem tipa *PowerUp* objekta s kojim se sudario igrač, u listi svih *PowerUp* objekata se aktivira odgovarajući *PowerUp* i postavi se vrijeme njegova trajanja. Efekt koji označava određeni aktivirani *PowerUp* zatim se primjenjuje na igrača.

Grafička reprezentacija *PowerUp* objekata

Za grafički prikaz *PowerUp* objekata koristi se metoda *billboard*-a. *Billboard* je ploča koja je uvijek okrenuta prema promatraču tako da promatrač u svakom trenutku vidi teksturu *PowerUp* objekta.

Svaki *PowerUp* objekt ima različitu teksturu koja se dinamički mijenja ovisno o tipu *PowerUp* objekta koji se generira. Model *PowerUp* objekta kreiran je tako da se *billboard* s odgovarajućom teksturom nalazi unutar prozirne sfere. Tekstura *PowerUp* objekta je također transparentna.

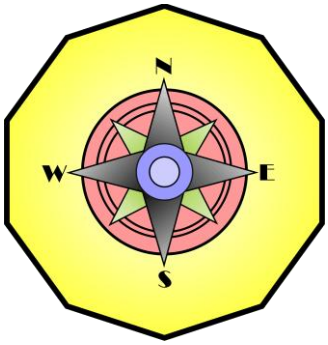


Slika 14. Prikaz modela *PowerUp* objekta

Zbog transparentnosti *PowerUp* objekata potrebno je izvršiti alfa korekciju prilikom iscrtavanja. Alfa korekcijom određuje se redoslijed iscrtavanja tako da se najprije iscrtaju objekti koji su udaljeniji od kamere te zatim objekti koji su bliže kameri. Kada se alfa korekcija ne bi koristila, neispravno bi se iscrtavala prozirnost. Na taj način svaki igrač prilikom iscrtavanja svog dijela prozora igre mora posložiti reodslijed iscrtavanja *PowerUp* objekata ovisno o svojoj lokaciji, odnosno položaju svog očišta.

Teksturiranje

Teksture upotrijebljene u izradi projekta izrađene su pomoću CorelDRAW Graphich Suite X5 programa za grafički dizajn. Izrađene su teksture za zidove labirinta, podove labirinta, *skydome* i *PowerUp* elemente.



Slika 15. Primjer izrađene teksture za *PowerUp* element

CorelDRAW Graphich Suite X5

CorelDRAW je vektorski grafički editor kojeg je razvila tvrtka *Corel Corporation.* Razvijen je primarno za *Windows* operativne sustave. CorelDRAW nudi korisnicima razne mogućnosti kao npr. korištenje CMYK formata boja umjesto RGB ili alat za uređivanje čvorova koji radi drugačije na drugačiji objektima. Svaki od CorelDRAW grafičkih paketa sadrži nekoliko različitih programa, ovisno o verziji. Korišteni CorelDRAW Graphics Suite X5 sadrži sljedeće:

- **CorelDRAW:** Editor za vektorsku grafiku
- **Corel PHOTO-PAINT:** Program za stvaranje i izmjenu rasterske grafike
- **Corel CONNECT:** Organizator zadržaja
- **Corel CAPTURE**: Program koji podržava različite metode hvatanja slika
- **Corel PowerTRACE**: Koverter između rasterske i vektorske grafike
- **Bitstream Font Navigator**
- **SB Profiler**

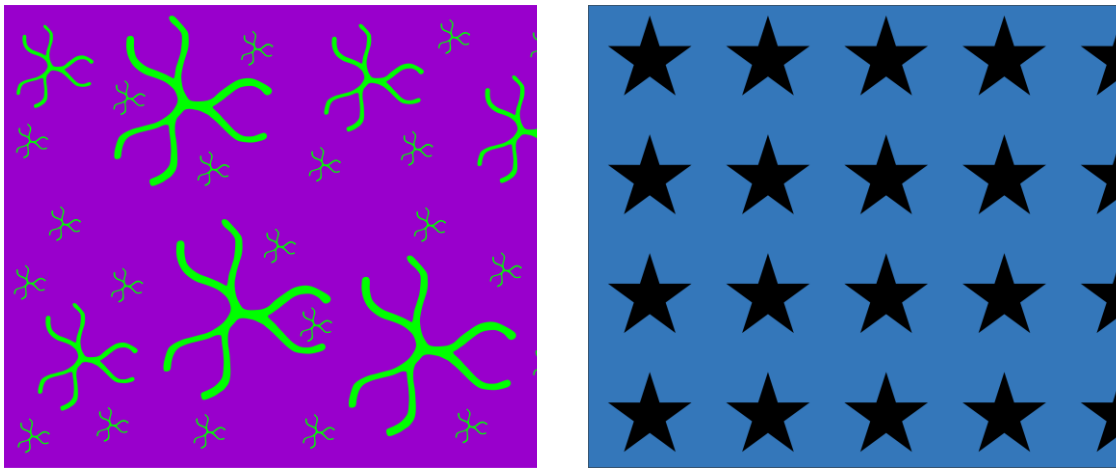
Od navedenih alata iz CorelDRAW grafičkog paketa korišteni su CorelDRAW i Corel PHOTO-PAINT.

Izrada tekstura

Korištene teksture rađene su na dva načina:

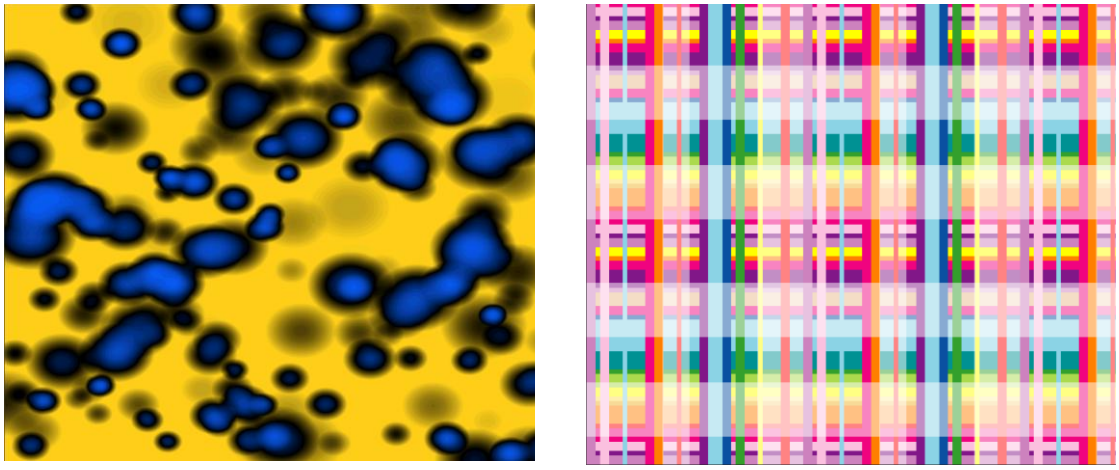
- ručnim crtanjem slike i
- korištenjem ugrađenih generatora uzoraka

Prilikom ručnog crtanja slike korišteni su ugrađeni alati za crtanje kao npr. alati za crtanje osnovnih oblika, alati za bojanje oblika, B-spline alat i sl. Primjeri ručno crtane teksture prikazani su na slici 16.



Slika 16. Primjeri ručno crtanih tekstura

Osim ručno crtanja teksture su rađene i pomoću ugrađenih alata za generiranje uzoraka. Kod ovakve izrade tekstura potrebno je izabrati uzorak (već ponuđeni ili definiran od strane korisnika), podesiti njegove parametre na željene vrijednosti (npr. broj specifičnih objekata, gustoća) i odabrati boje (dvije ili više).



Slika 17. Primjeri generiranih tekstura

Upravljanje kamerom

Kamerom se simulira pogled na igrača iz trećeg lica. Prilikom stvaranje nove kamere potrebno je zadati igrača kojeg kamera prati, položaj kamere i točku u koju kamera gleda (engl. *look at*) u lokalnom koordinatnom sustavu igrača, udaljenost bliske i daleke odrezujuće plohe, te dio prozora (engl. *viewport*) na koji kamera projicira sliku.

Da bi gibanje kamere bilo glatko, kamera se ne postavlja automatski na željenu poziciju, odnosno nije čvrsto vezana za igrača kojeg prati. Umjesto toga simuliramo situaciju kao da je kamera za igrača vezana oprugom. Udaljenost trenutne pozicije kamere od željene pozicije (engl. *stretch*), zajedno s parametrima koji određuju fizikalna svojstva opruge, određuje silu i u konačnici brzinu kojom će se kamera primicati željenoj poziciji. Na ovaj način je dobiven efekt glatkog gibanja kamere.

Da kamera ne bi prolazila kroz zidove, potrebno je detektirati koliziju kamere sa zidovima i napraviti odgovarajući pomak kamere (oporavak od kolizije). Kolizija se detektira na način da tražimo sjecište zrake sa zidom. Zraka polazi od točke u koju kamera gleda i prolazi kroz točku koja određuje poziciju kamere. Ako sjecište postoji i udaljenost na kojoj je sjecište nađeno je manja od udaljenosti na kojoj bi trebala biti kamera, detektirana je kolizija. Oporavak od kolizije radi se tako da iz udaljenosti na kojoj je detektirana kolizija računamo pomak kamere koji je potrebno učiniti kako kamera više ne bi bila u koliziji sa zidom i pomičemo kameru po pravcu.

Pokretač fizike

Kolizija sa zidovima

Kolizija sa zidovima je obrađena tako da se ispituje odnos igrač – zid za svaki postojeći zid u igri. S obzirom na relativno mali broj zidova (< 100), ovakav način kolizije ne predstavlja probleme, odnosno, ne dovodi do usporenja rada igre.

Programski igrač je pri koliziji predstavljen pomoću programske strukture *BoundingSphere* koja je svojstvena *XNA Game Studio 4.0* programskom alatu. Ona predstavlja imaginarnu sferu oko objekta pomoću koje se može lako detektirati kolizija s drugim sličnim programskim strukturama. S obzirom da su zidovi u igri predstavljeni sa sličnom programskom strukturom *BoundingBox*, detekcija kolizije se svodi na određivanje presjecanja nekog od zidova sa sferom oko igrača pomoću članske metode *Intersects* strukture *BoundingSphere*. Samo određivanje je programski riješeno preko statičke klase *CollisionDetection*. U nju se na početku izvođenja programa predaju sve *BoundingBox* strukture koje opisuju zidove te se u svakom novom ciklusu (metodi *Update*) u statičku metodu *CheckCollision* predaje objekt koji predstavlja igrača. Igraču se prije toga izračunava nova pozicija i ostale pripadne vrijednosti. Nakon toga se određuje sfera koja omeđuje igrača te se ispituju kolizije. Ukoliko dođe do kolizije, metoda vraća vrstu kolizije (npr. kolizija sa zidom) te se potom izvršava odgovor na koliziju.

Odgovor na koliziju (engl. *Collision detection*) je jednostavno odbijanje igrača od zida. Izvršeno je na način da se pozicija i pripadne vrijednosti izračunate prije detekcije kolizije vraćaju na vrijednosti iz ciklusa prije trenutnog (zato su definirane varijable koje pamte staro stanje igrača). Također, vektor igračeve brzine se množi s -1. Na taj način se igrač odbije u istom smjeru iz kojeg je došao. Zbog realističnosti prikaza, vektor brzine se također množi faktorom 0.5. Zbog toga je odbijanje manje primjetno i više realistično.

Problemi su se pojavili u određivanju centra sfere koja omeđuje igrača. Naime, početno je korištena pozicija igrača na mapi. Kako pozicija igrača nije bila u središtu njegova modela, sfera nije dobro opisivala igrača. Promjenom pozicije igrača, taj je problem riješen.

Skok

Implementacija skoka definirana je pomoću enumeracijskog tipa *Action*. U njemu su definirana 3 stanja: *Normal*, *Jumping* i *Falling*.

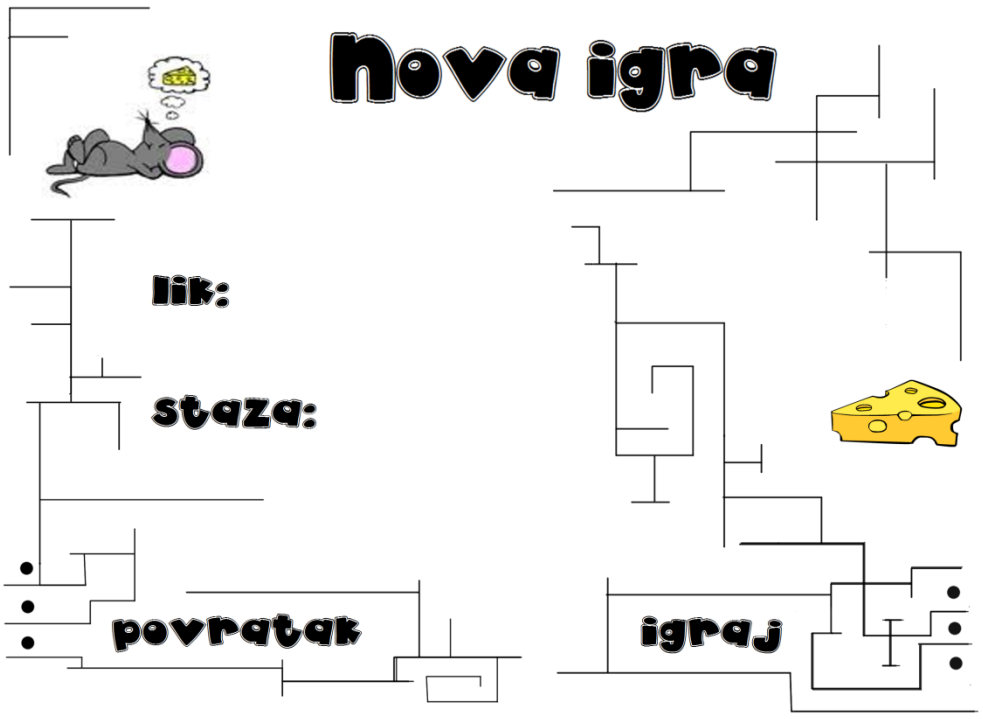
Ukoliko je pritisnuta tipka za skok, igrač prelazi iz akcije *Normal* u akciju *Jumping.* Izvođenje skoka se programski izvodi tako da se ukupnoj sili koja djeluje na igrača pridoda nova sila čiji je vektor jednak nuli na x i z-osi, a ima neku pozitivnu vrijednost na y-osi. Igrač ostaje u akciji *Jumping* sve dok ne dosegne određenu granicu skoka. Nakon akcije *Jumping* prelazi u stanje *Falling*. U tom stanju na igrača djeluje sila gravitacije tako da se ukupnoj sili dodaje sila čiji je vektor na x i z-osi jednak nuli, a ima neku negativnu vrijednost na y-osi. Igrač u akciji *Falling* ne može prijeći u drugu akciju sve dok njegova pozicija ne dosegne donju granicu (tj.pod). Tada prelazi ponovo u stanje *Normal*.

Grafičko sučelje

Grafičko sučelje implementirano je pomoću XNA biblioteka i Nuclex *framework*-a a sastoji se od sljedećih stranica:

- Glavni izbornik (engl. *Main)*,
- Nova igra (engl. *New Game*),
- nama (engl. *Abou*t),
- Postavke (engl. *Options*) i
- Pomoć (engl.*Help*).

Svaka stranica grafičkog sučelja podklasa je klase *Screen* kojom manipulira statička klasa *ScreenManager*. Klasa *Button* zadužena je za tipke koje se nalaze na svakoj stranici a klasa *MouseHandler* za poziciju i teksturu miša.



Slika 18. Prikaz jedne stranice grafičkog sučelja

Upute za korištenje

Igra se pokreće putem izvršne datoteke *Labirint.exe*. Grafičko sučelje nudi na izbor nekoliko opcija. Klikom na naslov “Nova igra” pokreće se nova igra te se nudi mogućnost odabira likova i tipa labirinta. Klikom na jednu od točkica koja void labirintom do naslova “Igraj” započinje nova igra.

Igrač broj 1 navigira lika pomoću direkcijskih strelica i desne tipke *Ctrl* dok igrač broj 2 igra pomoću kombinacija tipki W, S, A i D te lijeve tipke *Ctrl*.

Cilj igre je doći što prije do sredine labirinta, pokupiti kolač “Čupavac” i isti pokušati dostaviti do “Čupavaca” koji se nalaze negdje u labirintu.

Igrač koji kolač preda “Čupavcima” pobjeđuje.

Zaključak

Izrada ovog projekta bila je veliki izazov. Mišljenje je voditelja da je završni izgled projekta u potpunosti ispunio sva očekivanja koja su postavljenja pred cijeli tim. U računalnoj igri “Labirint” prikazani su mnogi od elemenata potrebnih za ostvarivanje proizvoda ovog tipa, a svaki od članova tima pridonio je svojim radom u ostvarivanju kvalitetne igre.

Projekt je u potpunosti završen i ispunio svoj cilj.

Literatura

[1] APP Hub, Chase camera, 26. svibanj 2010., *Chase Camera*, http://create.msdn.com/en-US/education/catalog/sample/chasecamera, 23. studeni 2010.

[2] APP Hub, http://forums.create.msdn.com/forums/, 19. studeni 2010.

[3] CG What, http://www.cgwhat.com/, 24. studeni 2010.

[4] MazeWorks, *How to build a maze, 20. studeni 2010., http://www.mazeworks.com/mazegen/mazetut/index.htm*