Izrada pletenih objekata iz 3D modela
Tehnička dokumentacija
Verzija 1.0

Student: Ana Marija Devčić

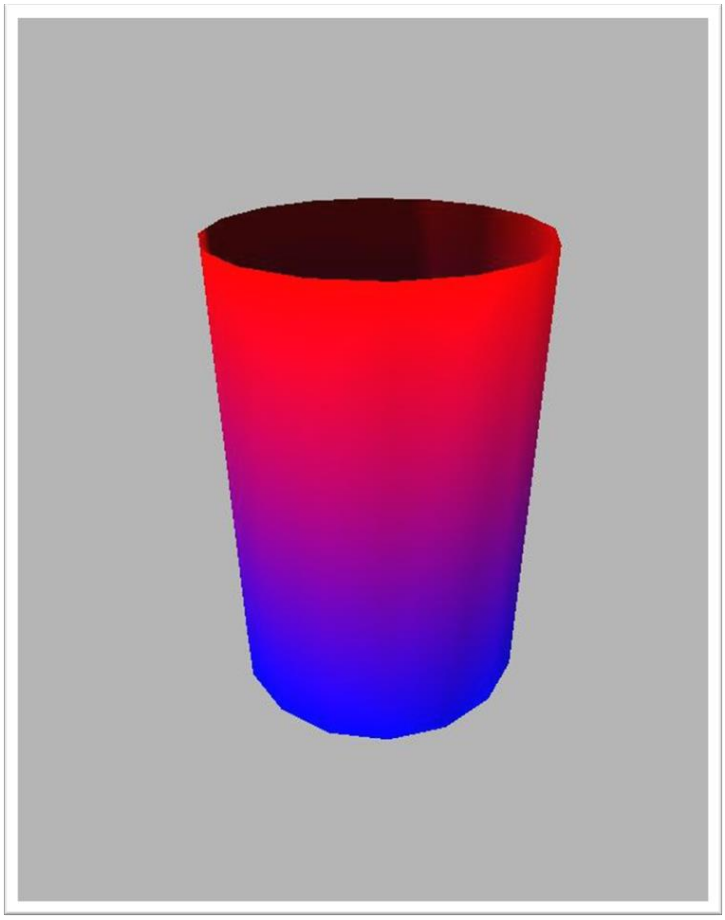
Nastavnik: Željka Mihajlović

Sadržaj

Tehnička dokumentacija

1. Opis razvijenog proizvoda

Razvijeni proizvod je aplikacija koja omogućuje korisniku da iz bilo kojeg 3D modela izradi uzorak za pletenje. Aplikacija kao ulaz prima .obj datoteku s željenim 3D modelom te .txt datoteku koja sadrži željeni početak i kraj pletiva. Prije provođenja algoritma korisnik može vidjeti prikaz 3D modela i njegovu funkciju vremena. Aplikacija je napisana u C++-u te se može *buildati* i pokrenuti kao samostalni program.



Slika 1: Prikaz modela u programu

2. Tehničke značajke

2.1 Korišteni alati i tehnologije

2.1.1 OpenGL

OpenGL je biblioteka za grafičko programiranje. Omogućava prikaz trodimenzionalnih modela i podataka na ekranu uz korisnički definirane *shadere.*
GLAD je alat koji dodaje ekstenzije za OpenGL.
GLFW je biblioteka koja implementira razne funkcije za rad u OpenGL-u.
GLM je biblioteka koja implementira matematičke operacije i strukture za brzo i lako obrađivanje točaka u 3D prostoru. Ovdje je korištena za obrađivanje modela i njegovo pretvaranje u drugi model.

2.1.2 TinyOBJLoader

TinyOBJLoader je biblioteka za jednostavno učitavanje .obj datoteka. Najprikladniji je za ovu svrhu za razliku od ostalih biblioteka za učitavanje modela zbog svoje male veličine – sastoji se od jednog *headera* – te zbog svoje jednostavnosti.

2.1.3 Blender

Blender je program za izradu 3D modela. Ima mogućnost *exportanja* modela u .obj format. Korišten je za izradu modela za testiranje programa.

2.2 Osnovno o pletenju

Pletivo se sastoji od redova i stupaca očica. Ovo je analogno 3D modelu koji se sastoji od točaka spojenih u *quadove* ili četverokute. Ovime možemo zaključiti da se bilo koje pletivo može interpretirati kao 3D model, no obrnuto nije nužno istina.

2.3 Opis algoritma

Prije provedbe algoritma model se pripremi za obradu. U svakoj točki se pohrane indeksi njenih susjeda. Definira se visina i širina očice, te širina konačnog objekta. Na osnovu toga se računaju udaljenosti između točaka. Na osnovu definiranog početka i kraja se u točkama izračuna funkcija vremena, iliti redoslijed kojim se točke pletu od 0 do 1. Početne točke imaju vrijednost 0, krajnje točke vrijednost 1, te se sve točke između interpoliraju između te dvije vrijednosti ovisno o udaljenosti od početnih i krajnjih točaka.

Prva faza algoritma je da model podijeli po širini. Algoritam počinje spajanjem svih početnih točaka u jednu petlju. Za svaku točku se prođe kroz sve njene susjede. Oni susjedi koji su s njom u petlji se ignoriraju. Oni kojima je funkcija vremena manja od točke se također ignoriraju. Od ostalih susjeda se nađe onaj najbliži.
Ako je najbliži susjed unutar visine očice, u idućem redu se on uzima kao točka u petlji. Ako je izvan visine očice, onda se pronađe točka koja je za visinu očice udaljena od točke u smjeru susjeda. Ovo se ponovi za svaku točku u petlji, te se time izračuna nova petlja na kojoj se ponavlja algoritam. Algoritam se ponavlja sve dok se ne dođe do kraja modela, to jest do trenutka kad točke u petlji nemaju više validnih susjeda.

Jednom kad su izračunati svi redovi se računaju stupci, što je druga faza algoritma. Kreće se opet od početnog retka i ide prema krajnjem. Za svaki prethodno izračunati redak se ide od početne točke u krug dok se ne dođe opet do iste točke. Princip je isti kao i u prethodnom algoritmu, ako iduća točka nije unutar definirane širine očice onda se pomičemo prema idućoj točki, inače uzimamo iduću točku kao točku u petlji.

3. Upute za korištenje

Izvorni kod se može preuzeti na *GitHub-u* [1]. Preuzeti kod se može *buildati* ako korisnik ima instaliran *CMake* i bilo koji C++ *compiler*. Biblioteke *glm, glad, glfw* i *TinyOBJLoader* su uključene u repozitorij, dok je očekivano da korisnik ima *OpenGL* instaliran, što većina modernih računala ima.
Program kao parametre prihvaća .obj datoteku, .txt datoteku s početnim i krajnjim točkama, visinom i širinom očice te dimenzijama konačnog modela. Zatim se u novom prozoru prikaže slika 3D modela te se konačni uzorak pohrani u datoteku.
Konačni program je testiran i može se pokrenuti na Windowsu i na Linuxu.

4. Literatura

[1] A. M. Devčić, »knit-generator,« [Mrežno]. Available: https://github.com/amdevcic/knit-generator.

[2] V. Narayanan, K. Wu, C. Yuksel i J. McCann, »Visual Knitting Machine Programming«.

[3] V. Narayanan, L. Albaugh, J. Hodgins, S. Coros i J. McCann, »Automatic Machine Knitting of 3D Meshes«.

[4] J. McCann, L. Albaugh, V. Narayanan, A. Grow, W. Matusik, J. Mankoff i J. Hodgins, »A Compiler for 3D Machine Knitting«.

[5] A. Kaspar, L. Makatura i W. Matusik, »Knitting Skeletons: Computer-Aided Design Tool for Shaping and Patterning of Knitted Garments«.

[6] A. Kaspar, K. Wu, Y. Luo, L. Makatura i W. Matusik, »Knit Sketching: from Cut & Sew Patterns to Machine-Knit Garments«.

[7] Y. Igarashi, T. Igarashi i H. Suzuki, »Knitting a 3D model,« 2008.

Korisničko sučelje za uređivač pokretača video igara u DirectX 12
Tehnička dokumentacija
Verzija 1.0

Studentski tim: David Čemeljić

Nastavnik: Željka Mihajlović

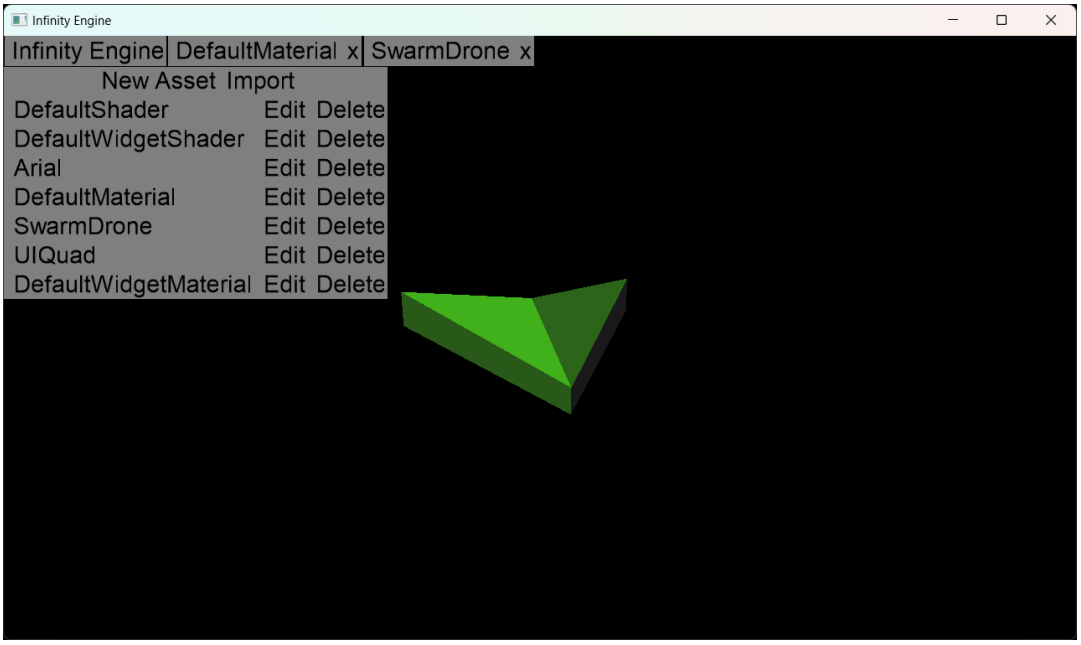
Sadržaj

1. Opis razvijenog proizvoda

U sklopu ovog projekta razvijen je razvojni okvir za korisničko sučelje unutar pokretača video igara. Pomoću tog okvira razvijeno je osnovno sučelje za uređivač projekata kako bi se olakšalo upravljanje resursima poput modela, materijala, fontova itd.

Razvijene su brojne komponente unutar okvira pomoću kojih je sastavljeno korisničko sučelje za uređivač projekata. Cijeli sustav povezan je sa sustavom refleksije, te je vrlo jednostavno kreirati meni za uređivanje resursa određenog tipa.

Slika 1 prikazuje početni ekran uređivača.



Slika 1 Glavni ekran korisničkog sučelja

2. Tehničke značajke

2.1 Opis

Razvojni okvir za korisničko sučelje napisan je u sklopu pokretača video igara *Infinity Engine*. Pokretač je pisan u programskom jeziku *C++*, namijenjen za platformu *Windows*. Za implementaciju grafičkih značajki koristi se *DirectX 12*.

Jedina vanjska biblioteka koja se koristi unutar projekta je *DirectXTK12*. Iz te biblioteke koristi se samo matematika (omotač oko *DirectXMath*) te implementacija *bitmap* fontova za iscrtavanje teksta. *Bitmap* fontovi koriste atlas tekstura u kojima su znakovi, te se tekst dobiva iscrtavanjem kvadrata popunjenih teksturama.

Grafički elementi su dizajnirani tako da budu responzivni i jednostavni za korištenje te kreiranje svojih novih kompozitnih elemenata. Cijeli sustav je u potpunosti modularan.

Veličina i pozicija elemenata je određena udjelom elementa u elementu roditelju. Ako u korijenski element prozora dodamo element i postavimo mu veličinu (1, 1), element će prekriti cijeli prozor.

Implementirana su i „sidra“ kako bi se lakše računale koordinate u odnosu na roditelja. Također, implementirana su i sidra u odnosu na vlastito ishodište. Na primjer, ako namjestimo oba sidra gore lijevo, element će biti pozicioniran u gornjem lijevom kutu roditelja te odmaknut tako da cijeli stane u kut, odnosno da ne izlazi iz okvira roditelja.

Osnovni sjenčar implementira SDF [1] za ostvarivanje zaobljenih rubova u slučaju da korisnik želi takav izgled.

Pokretač podržava neograničen broj prozora. Prozori se mogu povećati i smanjiti, no sadržaj u njima se osvježava samo kad je skaliranje dovršeno. Razlog tome je što skaliranje zahtjeva promjenu veličine (odnosno rekreiranje) spremnika dubine (engl. *depth buffer*) i spremnika boja (engl. *frame buffer*). Skaliranje se događa asinkrono, kao poruka procesu. Zbog toga je opasno pokušati rekreirati spremnike te je najjednostavnije rješenje odgoditi kreiranje novih spremnika do završetka skaliranja [2].

2.2 Implementirani elementi

Kreirane su brojne komponente uz pomoć kojih se mogu dizajnirati kompleksni UI element:

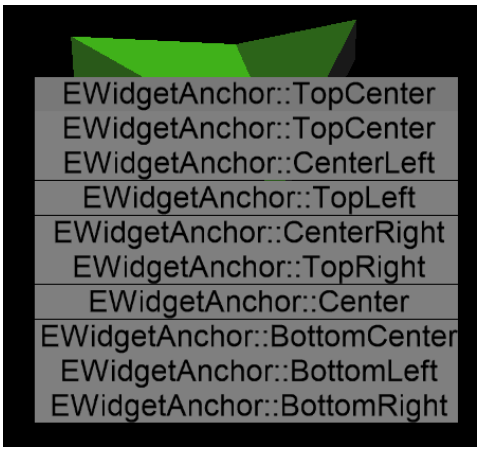
- *Widget*
    - Osnovna klasa, implementira transformacije te osnovni raspored za jedno dijete
- *CanvasPanel*
    - Veličina panela ne ovisi o veličini djece, a raspored djece je određen njihovom relativnom pozicijom
    - Podržava neograničen broj djece
- *FlowBox*
    - Podržava dvije orijentacije - vertikalnu i horizontalnu
    - Veličina ovisi o zbroju traženih veličina djece i njihovog popunjavanja (engl. *padding*)
    - Elementi mogu biti poravnati od početka, kraja ili u centru
- *TextBox*
    - Koristi se za iscrtavanje teksta
    - Veličina ovisi o veličini teksta, koja se računa iz fonta

  - Koristi *bitmap* fontove uvezene u projekt
  - Kad je kursor iznad ovog elementa te mu je uključena kolizija, promijenit će se ikona kursora u *IBeam*
- *EditableTextBox*
  - Omogućava uređivanje upisanog teksta
  - Veličina također ovisi o veličini upisanog teksta
  - Za vizualizaciju pozicije kursora implementirana je klasa *Caret*
  - Slika 2 prikazuje ovaj element, kursor je na kraju teksta

Slika 2 Prikaz elementa EditableTextBox kad je fokusiran

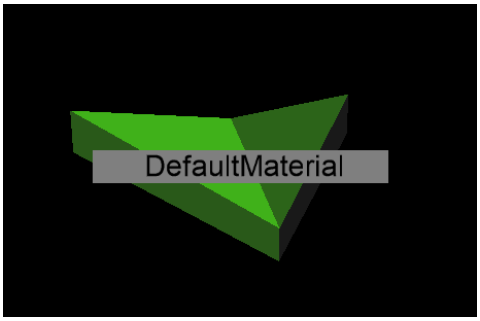
- *Caret*
  - Vizualizacija kursora unutar *EditableTextBox*
- *Button*
  - Pomoćna komponenta koja predstavlja gumb s tekstom
- *Checkbox*
  - Predstavlja potvrdni okvir
- *DropdownMenu*
  - Klikom na ovaj meni otvara se lista opcija
- *DropdownTextChoice*
  - Pomoćna klasa koja popunjava *DropdownMenu*, sadrži samo tekst
- *EnumDropdown*
  - DropdownMenu koji se popunjava nazivima elementa *C++* enumeracije za koju je uključena refleksija
  - Slika 3 i Slika 4 prikazuju ovaj element kreiran za enumeraciju *EWidgetAnchor*

Slika 3 Prikaz elementa EnumDropdown za enumeraciju EWidgetAnchor

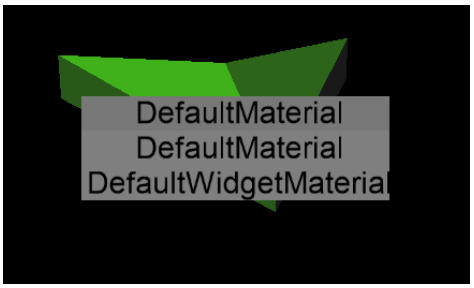


Slika 4 Prikaz elementa EnumDropdown za enumeraciju EWidgetAnchor s otvorenim opcijama

- *TypePicker*
    - *DropdownMenu* popunjen nazivima podtipova određenog tipa
- *AssetPicker*
    - *DropdownMenu* popunjen resursima određenog tipa, npr. ako je kao tip postavljen *Material::StaticType()*, meni će biti popunjen samo materijalima
    - Slika 5 i Slika 6 prikazuju ovaj element za tip Material::StaticType()



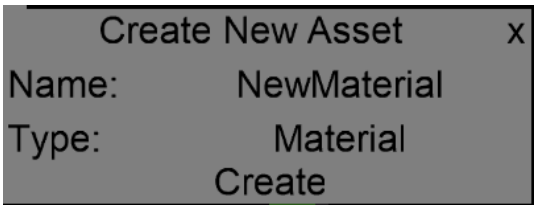
Slika 5 Prikaz menija za odabir resursa tipa Material



Slika 6 Prikaz otvorenog menija za odabir resursa tipa Material

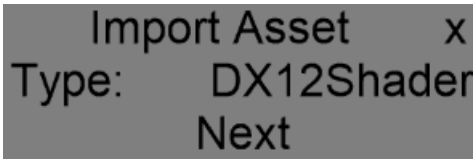
- *TableWidget*
    - Predstavlja tablicu
    - Veličina stupaca je automatski izračunata

  - Podržava različit broj stupaca po retku
- *WidgetSwitcher*
  - U jednom trenutku prikazuje samo jedno dijete
  - Indeks prikazanog djeteta postavlja se ručno
- *TabSwitcher*
  - Koristi *WidgetSwitcher* za prikaz *tab*-ova
  - *Tab*-ovi se mogu dodati i zatvoriti
  - *Tab*-ovi imaju gumb za zatvaranje, ali zatvaraju se i na klik srednjeg gumba miša
  - *Tab*-ovi se mogu kreirati tako da ih se ne može zatvoriti
- *AssetBrowser*
  - Koristi tablicu za prikaz resursa
  - Automatski se osvježava
  - Omogućava kreiranje novih resursa, uvoz resursa, uređivanje i brisanje
  - Klikom na gumb za uređivanje otvara se novi *tab*
- *EditorWidget*
  - Glavni element uređivača, koristi *TabSwitcher*
  - Sadrži *AssetBrowser* na prvom *tab*-u
  - Dodatni *tab*-ovi se dodaju po potrebi
- *AssetCreatorMenu*
  - Služi za kreiranje novih resursa
  - Potrebno je odabrati tip i naziv novog resursa
  - Prikazan je na slici Slika 7



Slika 7 AssetCreatorMenu

- *AssetImportMenu*
  - Služi za uvoz resursa
  - Klikom na *Next* otvara se novi element, specifičan za odabrani tip resursa
  - Prikazan je slikom Slika 8



Slika 8 AssetImportMenu

2.3 Elementi za reflektirane klase

Uz pomoć prethodno navedenih klasa implementirani su elementi za reflektirane klase.

Kako bi klasa bila reflektirana, treba je označiti s *REFLECTED()* i *GENERATED()*.

Pojedine članove klase koje želimo reflektirati potrebno je označiti s *PROPERTY()*. Ta oznaka može primiti parametre poput *Edit* i *DisplayName*. Edit označava da će se član moći uređivati u korisničkom sučelju. Ako nema atributa *Edit*, član će biti prikazan u sučelju, ali se vrijednost neće moći mijenjati.

DisplayName određuje naziv člana koji će biti prikazan u korisničkom sučelju. Ako nije naveden, preuzima se naziv varijable.

Slika 9 prikazuje zaglavlje elementa FlowBox s oznakama za refleksiju. Na slici Slika 10 prikazane su dvije reflektirane enumeracije.

```
REFLECTED()
class FlowBox : public Widget
{
    GENERATED()

public:
    FlowBox() = default;
    FlowBox(const FlowBox& other) = default;
    FlowBox& operator=(const FlowBox& other) = default;

    void SetDirection(EFlowBoxDirection direction);
    EFlowBoxDirection GetDirection() const;

    void SetAlignment(EFlowBoxAlignment alignment);
    EFlowBoxAlignment GetAlignment() const;

    // Widget
protected:
    void RebuildLayoutInternal() override;
    void UpdateDesiredSizeInternal() override;

    void OnChildAdded(const std::shared_ptr<Widget>& child) override;

private:
    PROPERTY(Edit, DisplayName = "Direction")
    EFlowBoxDirection _direction = EFlowBoxDirection::Vertical;

    PROPERTY(Edit, DisplayName = "Alignment")
    EFlowBoxAlignment _alignment = EFlowBoxAlignment::Center;
};
```

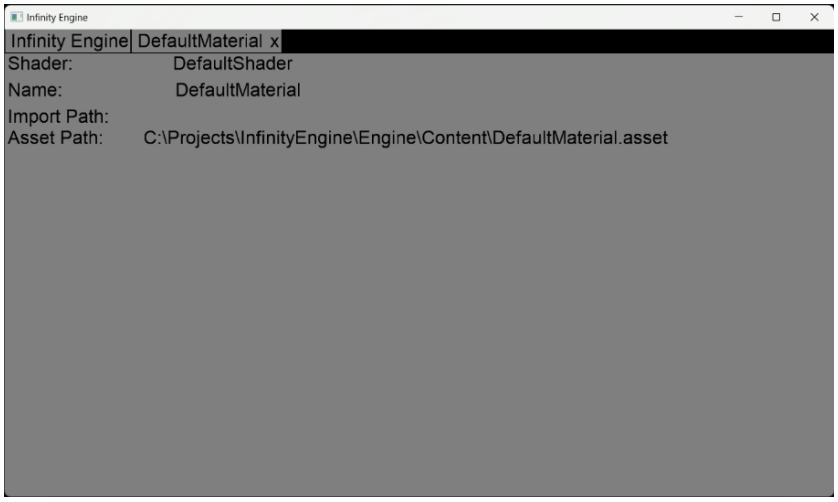
Slika 9 Kod reflektirane klase FlowBox

Za refleksiju enumeracija, potrebno ju je samo označiti s *REFLECTED()*. Ako ta oznaka sadrži atribut *BitField*, enumeracija će biti prikazana listom potvrdnih okvira umjesto listom opcija.

```
REFLECTED()
enum class EFlowBoxDirection : uint8
{
    Horizontal,
    Vertical
};

REFLECTED()
enum class EFlowBoxAlignment : uint8
{
    Start,
    Center,
    Justify,
    End
};
```

Slika 10 Primjer reflektirane enumeracije

Za bilo koju reflektiranu klasu moguće je kreirati element za uređivanje. Element je automatski generiran za svaki reflektirani član. Ako taj član nije reflektiranog tipa, potrebno je napisati dvije funkcije kojima se vizualizira tip. Za primitivne tipove i najčešće korištene tipove već postoji implementacija. Na slici Slika 11 prikazan je takav element za materijal.



Slika 11 Automatski generirani element za uređivanje objekta klase Material

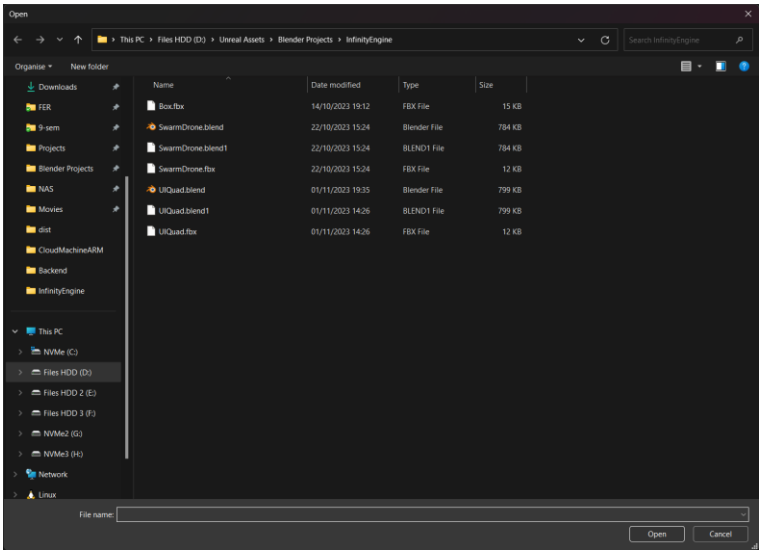
Za uvoz resursa također se koristi refleksija. Primjer objekta za uvoz prikazan je na slikama Slika 12 i Slika 13. Ovaj objekt definira da za uvoz modela trebamo putanju.

```
REFLECTED()
class StaticMeshImporter : public Importer
{
    GENERATED()

public:
    PROPERTY(Edit)
    std::filesystem::path Path;

public:
    StaticMeshImporter() = default;
};
```

Slika 12 StaticMeshImporter

Slika 13 StaticMeshImporter UI element

Za tip putanje koji se može mijenjati u korisničkom sučelju definiran je UI element – element za upisivanje teksta i gumb. Gumb asinkrono otvara meni za odabir datoteke koji je dio operacijskog sustava (prikazan na slici Slika 14). Kad korisnik odabere putanju, objekt se popuni s tom vrijednošću.



Slika 14 Windows File Explorer

3. Upute za korištenje

Za pokretanje programa potreban je *Visual Studio 2022*, *CMake* 3.26 i *Conan* 1.60.

Nakon kloniranja projekta s *GitHub*-a, potrebno je pokrenuti *GenerateProjectFiles.bat*. Skripta pokreće *Conan* i *CMake*, instalira vanjske biblioteke i alate te generira *Visual Studio Solution* u *Build* direktoriju. Otvoriti generirani *InfinityEngine.sln* kroz *Visual Studio* ili *Rider* te pokrenuti izgradnju i pokretanje programa.

Projekt je dostupan na *GitHub*-u: https://github.com/WingmanD/InfinityEngine

4. Literatura

[1] Akenine-Möller, T; Haines, E; Hoffman, N; Pesce, A; Iwanicki, A; Sébastien, Hillaire, S. Real-Time Rendering. 4. izdanje. Boca Raton, FL, USA: A K Peters/CRC Press, 2018.

[2] Luna, F.D. Introduction to 3D Game Programming with DirectX 12. 1. izdanje. Mercury Learning and Information, 2016

Generiranje cestovne mreže s pomoću OpenDRIVE i OpenStreetMap formata

Antonija Engler*
*Fakultet elektrotehnike i računarstva/Sveučilište u Zagrebu, Zagreb, Hrvatska

***Sažetak* – Samovozeći automobili su sve više prisutni u današnjem prometu. Kako bi njihovo treniranje i testiranje bilo sigurnije, novčano isplativije i prilagođeno svim uvjetima, koriste se simulacije autonomne vožnje. Osnova takvih simulacija su cestovne mreže koje se sastoje od kolnika, semafora i prometnih znakova. Postoji mnogo formata u kojima se takvi podaci mogu pohraniti, no zbog popularnosti su uzeti OpenDRIVE i OpenStreetMap koji imaju svoje prednosti i nedostatke. U ovom radu će se detaljnije proučiti ova dva alata, usporediti ih i opisati kako se mogu iskoristiti unutar alata CARLA. Na kraju će se prikazati rezultati generiranja jednostavne ceste.**

Ključne riječi – OpenDRIVE; OpenStreetMap; CARLA; cestovne mreže; simulacija; generiranje mape

I. UVOD

Kontrola samovozećih automobila ovisi ponajviše o informacijama koje vozilo dobiva iz senzora, kao na primjer radar, ultrazvuk, radio, infracrveni senzor i kamera [2]. Digitalne mape predstavljaju upravo takve podatke te su zbog toga važan dio razvoja automatizirane vožnje. Mape i formati u kojima su spremljene nameću ograničenja navigacije i omogućuju sustavu povezivanje informacije o poziciji sa stvarnom lokacijom automobila. Kako bi razvoj bio što točniji, potrebni su robusni podaci visoke preciznosti i dosljedan krajolik. Takvi zahtjevi mogu se postići ako se kod generiranja mape fokusira na tri elemenata: geometriju, implementaciju i upotrebljivost [3]. Svaka cestovna mreža mora biti čitava i točna, od globalnog izgleda do svake pojedine trake. Podaci moraju biti spremljeni u učinkovite i kompaktne strukture koje omogućuju preuzimanje, osvježavanje i obradu putem bežične mreže. Na kraju bi se mape i cestovne mreže spremljene u njima trebale moći procesuirati u stvarnom vremenu. [3]

Kako bi se svi navedeni zahtjevi ispunili, potrebno je detaljno poručiti svaki format korišten za generiranje digitalnih mapa i shvatiti njegove prednosti i nedostatke.

II. FORMATI CESTOVNIH MREŽA

Postoji mnogo alata otvorenog koda koji su besplatni za korištenje, a u ovom poglavlju ćemo obraditi dva: OpenDRIVE i OpenStreetMap.

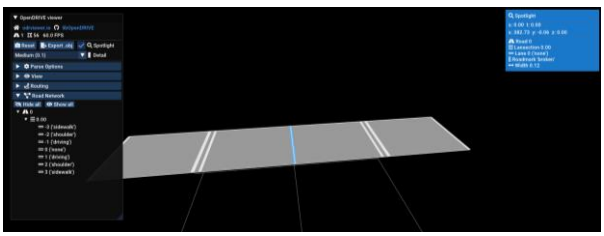
A. OpenDRIVE

ASAM (engl. *Association for Standardization of Automation and Measuring Systems)* OpenDRIVE je format koji pruža mogućnost opisivanja cestovnih mreža s pomoću XML (engl. *extensible markup language*) sintakse u obliku *xodr* ili *xordz* datotečnog nastavka. Koristi se za opis prometnih mreža, u simulacijama vožnje i u simulacijama prometa, a cilj mu je pružiti standardizirani format koji se može prosljeđivati između različitih simulacija i sustava bez potrebe prebacivanja u druge formate. [1]

Podaci koji su spremljeni u ASAM OpenDRIVE datoteci opisuju geometriju cesta, traka i objekata na cesti (kao na primjer putokazi) ili uz cestu (kao na primjer semafori). Mogu biti temeljeni na stvarnim podacima, stvoreni u za to predviđenim aplikacijama ili izvučeni iz drugih oblika podataka.

Cijeli format organiziran je u čvorove (engl. *nodes*). Svaki čvor ima sintaksu <*podatak*></*podatak*> unutar koje se može definirati bilo kakav podatak koji se želi prikazati. Time se omogućuje iznimna specijalizacija korisnikovim potrebama, no i dalje se zadržava kompatibilnost s drugim aplikacijama.

Temeljna komponenta svake cestovne mreže je referentna linija [1] (engl. *reference line*, plava linija na slici 1). Svi ostali cestovni objekti: ceste, trake i njihove elevacije, spojeni su na referentnu liniju. S druge strane, objekti koji predstavljaju obilježja uz cestu, kao na primjer prometni znakovi ili semafori, mogu biti postavljeni uz pomoć referentne linije ili uz pomoć globalnog koordinatnog sustava u kojem se cestovna mreža nalazi.



Slika 1. Prikaz jednostavnog dijela ceste u online alatu „OpenDRIVE viewer“.

Jednom kada su ceste definirane, one se mogu međusobno povezati. Svakoj cesti se može definirati prethodnik (engl. *predecessor*) i sljedbenik (engl. *successor*) koji može biti druga cesta ili raskrižje. Kako bi se bolje simulirala vožnja, OpenDRIVE daje mogućnost povezivanja i traka unutar dvije ceste.

B. OpenStreetMap

Alat OpenStreetMap je infrastruktura otvorenog kôda koju mogu uređivati volonteri, tvrtke i državne organizacije u cijelom svijetu [3].

OSM podaci preuzimaju se u topološki strukturiranom XML formatu. Svaka tako generirana datoteka sastoji se od tri vrste primitiva: čvorovi (engl. *nodes*), putevi (engl. *ways*) i odnosi (engl. *relations*). Čvor je zapisan u obliku para zemljopisne širine i dužine, a predstavlja točku u prostoru. Put je lista najmanje dvije reference čvora koje su linearno povezane. Ako su prvi i zadnji čvor isti, onda je put zatvoren. Odnosi predstavljaju grupe podatkovnih primitiva koje su povezane određenom ulogom, npr. tramvajska linija, prometna ograničenja itd. Odnosi također mogu biti nedefinirani ili se sastojati od nula primitiva. [4]

Osim podatkovnih primitiva, ne postoje druga strukturirana pravila. Svaki čvor, put i odnos može imati neograničeni broj oznaka (engl. *tags*) koji služe kao atributi. Oznaka se sastoji od para ključ i vrijednost, npr. <tag k="surface" v="asphalt"/>. Korisnik može definirati vlastite ključeve, no preporuča se slijediti već predefinirane i često korištene vrijednosti sa službenih stranica [5].

III. POSTUPAK GENERIRANJA JEDNOSTAVNE CESTE

Simulator CARLA (*Car Learning to Act*) je softverska platforma razvijena kako bi oponašala gradsku vožnju. Cilj CARLA-e je pružiti alat kojim će se trenirati, stvarati prototipovi i testirati modeli autonomne vožnje. Implementiran je kao server-klijent arhitektura, gdje server vrti simulaciju i prikazuje scenu s pomoću alata Unreal Engine 4, a klijent upravlja dinamičnim dijelom svijeta (vremenski uvjeti, gustoća prometa i gustoća pješaka) s pomoću programskog jezika Python preko mrežne utičnice (engl. *socket*). [6]

Budući da je dio CARLA-e za stvaranje novog virtualnog svijeta i manipuliranje njegovim komponentama napisan u Pythonu, odabran je taj jezik za generiranje mapa. Neovisno o formatu u kojem će se pisati cesta, za pojedini čvor napisana je klasa unutar koje su definirana određena svojstva čvora te funkcije za ispis sintakse. Neki čvorovi, koji nisu osnovna gradivna jedinica formata, nisu razmatrani u ovom članku kako bi se dao bolji pregled osnovne ideje.

A. *OpenDRIVE*

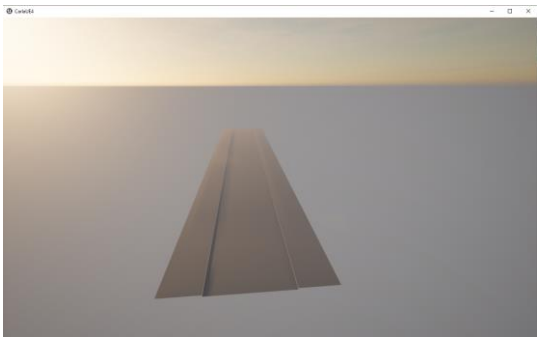
S obzirom na to da OpenDRIVE format pruža maksimalnu personalizaciju podataka, za ovaj članak implementirane su samo klase koje predstavljaju osnovne gradivne elemente cesta.

Klasa *Road* je struktura koja odražava jednu cestu unutar cijele prometne mreže. Ima atribute *name* (proizvoljno ime ceste), *length* (ukupnu duljinu ceste), *id* (jedinstveni identifikator) i *junction* (vrijednost koja pokazuje pripada li cesta raskrižju, -1 je pretpostavljena vrijednost, a označava da cesta nije spojena na raskrižje). Svaka cesta mora imati jedan *view* koji detaljnije opisuje geometriju ceste i bez kojeg se ne može prikazati ni u jednom alatu. Cesta se u stvarnom svijetu sastoji od traka pa se tako u atributu *lanes* popisuju sve trake koje se nalaze oko referentne linije.

Klasa *PlanView* sastoji se od pojedinih geometrija. Klasa *Geometry* opisuje te geometrije s pomoću atributa *s* (s koordinata referentne linije koja pokazuje lokaciju početne pozicije), *x* (x koordinata početne pozicije), *y* (y koordinata početne pozicije), *hgd* (početnu orijentaciju inercijskog navigacijskog sustava), *length* (duljinu referentne linije). Instanca te klase mora imati točno jedan dodatan element koji opisuje njen izgled: linija, spirala, luk, kubni polinom ili parametarski kubni polinom [7]. Za svaki od tih opcija može se definirati jedan razred, no zbog jednostavnost implementirani su samo linija i luk. Klasa *Line* jednostavna je reprezentacija linije koja ispisuje jedan redak *<line/>*. Klasa *Arc* ima atribut *curvature* koji definira konstantnu zakrivljenost segmenta ceste.

Svaka cesta mora imati definiranu barem središnju traku s dodatnom jednom trakom postavljene širine. Sve trake i općenite informacije o trakama zapisane su u klasi *Lanes*. Klasa *Lanes* ima atribut *lanesections* koji je lista klasa *LaneSection* i *LaneOffset*. Ponekad se u cesti središnja traka mora maknuti od referentne linije, a zato služi klasa *Offset*. Ona ima definirane atribute: *s* (s koordinata početnog položaja na referentnoj liniji), *a*, *b*, *c* i *d* (parametri polinoma trećeg stupnja s pomoću kojeg se računa pomak). Klasa *LaneSection* uzima atribute *s* (s koordinata početnog položaja) i *lanes* (lista traka). Vrste traka se dijele na: lijevu, srednju i desnu, s korespondentnim klasama *LeftLane*, *CenterLane* i *RightLane*. Svaka traka ima kao atribut *lanes* koji je lista gradivne jedinice *Lane*, a razlikuju se po ispisu. Klasa *Lane* se sastoji od atributa *id* (univerzalni identifikator), *type* (tip trake, npr. kolnik, pločnik, rub, zaustavni trak) i *level* (primjenjuje li se nadvišenje, engl. *superelevation*, pretpostavljeno „false“) [7]. Osim tih atributa, dodatno je definiran *width* koji opisuje širinu trake. Klasa *Width* sastoji se od atributa *sOffset* (s koordinata početne pozicije ovisna o prethodnom *laneSection* elementu) i četiri atributa *a*, *b*, *c* i *d* (parametri polinoma trećeg stupnja čiji rezultat je širina u pojedinoj točci) [7].

Klasa *OpenDRIVE* je početna klasa u koju se umeću sve željene ceste te se rekurzivno pozivaju svi podelementi cesta. Rezultat se sprema u datoteku *test.xodr* koja se prvo pročita u varijablu *xodr_xml*, a zatim naredbom *client.generate_opendrive_world(xodr_xml, ...)* prikazuje u alatu CARLA (slika 2.).



Slika 2. Generiran primjer ceste pomoću formata .xodr unutar alata CARLA

B. *OpenStreetMap*

OSM format sastoji se, kako je prije navedeno, od tri osnovna bloka: čvor, put i odnos pa su tako prvo implementirane klase *Node*, *Way* i *Relation*.

Klasa *Node* ima atribute *id* (daje jedinstvenu oznaku čvora koja se poslije može referencirati), *visible* (određuje hoće li čvor biti vidljiv, može biti točno ili netočno), *changeset* (interna oznaka koja identificira u kojoj skupini promjena je unesen podatak), *timestamp* (oznaka kada je unesen podatak), *lat* (zemljopisna širina na kojoj se nalazi čvor), *lon* (zemljopisna duljina na kojoj se nalazi čvor) i *tags* (lista oznaka koje detaljnije opisuju taj čvor). Uz te atribute, definiranu su tri funkcije ispisa: prva vraća početni redak u kojem se zapisuju atributi, druga vraća krajnji ispis koji označuje završetak tog čvora te treća koja objedinjuje prethodna dva ispisa i rekurzivno traži ispis svih oznaka.

Klasa *Way* se sastoji od sličnih atributa *id*, *visible*, *version*, *changeset*, *timestamp*, *tags* i dodatno *nds* (označuju reference čvorova koji čine taj put). Pri ispisu prvo se definiraju atributi, zatim se u sljedećoj razini ispisuju reference čvorova, nakon njih slijede oznake i na kraju završetak puta.

Klasa *Relation* ima ponovno iste atribute *id*, *visible*, *version*, *changeset*, *timestamp*, *tags* uz novi atribut *members* (opisuje vrstu elementa koji se nalazi u tom odnosu i njegovu ulogu). Kada se ispisuju odnosi, na početku se postave atributi, onda se u novoj razini ispišu članovi ako postoje, ispišu se određene oznake i završi se sa znakom kraja odnosa.

Klasa *Tag* definira oznaku koja detaljnije opisuje pojedini element, a obuhvaća atribute *k* (ključ) i *v* (vrijednost). Cijela oznaka ispisuje se u samo jednom redu.

Klasa *Member* definira član unutar odnosa i ima atribute *type* (tip elementa, može biti čvor, put ili odnos), *ref* (serijski broj koji određuje definirani element) i *role* (uloga, može se ostaviti prazna). Član se također ispisuje u samo jednom retku.

Klasa *Bounds* opisuje veličinu cijele cestovne mreže, sastojala se ona od jedne ceste ili bogate prometne infrastrukture. Najčešće je pravokutnog oblika te se svi elementi moraju nalazi unutar nje. Sastoji se od četiri atributa: *minlat* (manja zemljopisna širina), *minlon* (manja zemljopisna duljina), *maxlat* (veća zemljopisna širina) i *maxlon* (veća zemljopisna duljina). Ograničavajući pravokutnik mora se postaviti prije bilo kakve definicije čvorova, puteva i odnosa.

Na kraju je definirana klasa *OSM* koja postavlja metapodatke *version*, *generator*, *copyright*, *attribution* i *licence* kako bi CARLA znala o kojem formatu se radi te *bounds* (ograničavajući pravokutnik), *nodes* (lista čvorova), *ways* (lista puteva) i *realtions* (lista odnosa).

Unutar datoteke *main.py* postavljaju se određeni čvorovi, putevi i odnosi koji se žele prikazati te se poziva ispis *OSM* klase koja rekurzivno ispisuje sve elemente u datoteku *test.osm*. Jednom kada se dobije OSM format, pokreće se CARLA i naredbom *python config.py --osm-path=putanja\do\datoteke\test.osm* iscrtala se scena na slici 3.

Slika 3. Generiran primjer ceste pomoću formata .osm unutar alata CARLA

IV. ZAKLJUČAK

Digitalne mape su važan dio procesa treniranja i testiranja modela autonomnih vozila. Zato je bitno odabrati formate koji najbolje odgovaraju određenim situacijama, modelima i namjenama.

Oba proučena formata koriste jezik za označavanje podataka (engl. *markup language*) XML. Format OpenDRIVE koncipiran je tako da su osnovne gradivne jedinice oblika <*roads*> unutar kojih je dodana geometrija i detaljni opis traka. S druge strane, osnova OSM formata su čvorovi čije reference su povezane u ceste. OpenDRIVE se više fokusira na hijerarhiju i tako određuje odnose, npr. <*leftLane*> je roditelj trakama s *id*-em 1, 2 i 3, no tako i svaka druga cesta, dok OpenStreetMap prvo navodi sve čvorove, a onda ih povezuje referencama koje su jedinstvene. Odnosi između cesta u OpenDRIVE formatu zapisuju se s pomoću atributa *predecessor* i *successor* unutar svake ceste, no u OSM-u se to definira s pomoću elementa odnos (engl. *relation*). Za generiranje jednostavne ceste, u OpenDRIVE formatu bilo je potrebno više klasa, točnije trinaest, a u OpenStreetMap formatu šest. Što se tiče primjene, ne postoji određena baza primjera datoteka u *.xodr* formatu, dok se *.osm* datoteke mogu besplatno preuzeti sa službene OpenStreetMap stranice [8]. Konačno, OpenDRIVE koristi *s*, *t* i *x*, *y* koordinate, a OpenStreetMap geografsku duljinu i širinu za orijentaciju u prostoru.

Zaključno, OpenDRIVE pruža više detalja i mogućnosti manipulacije izgleda i funkcionalnosti cesta, no uz cijenu kompleksnosti i potrebnog dubokog poznavanja pravila formata. OpenStreetMap s druge strane osigurava jednostavnost definiranja čvorova, povezivanja ih u puteve i definiranja njihovih odnosa uz krute osnovne detalje u obliku parova ključeva i vrijednosti. Oba formata se mogu prikazati u sučelju CARLA-e i dobro služiti njenoj svrsi.

LITERATURA

[1] https://www.asam.net/standards/detail/opendrive/

[2] Hong Cheng, „Autonomous Intelligent Vehicles. Theory, Algorithms, and Implementation“

[3] Jorge Godoy, Antonio Artunedo, Jorge Villagra, „Self-Generated OSM Based Driving Corridors“

[4] https://docs.safe.com/fme/html/FME-Form-Documentation/FME-ReadersWriters/osm/osm.htm. Pristupljeno 13. siječnja 2024.

[5] https://wiki.openstreetmap.org/wiki/Category:Features. Pristupljeno 13. siječnja 2024.

[6] Alexey Dosovitskiy and German Ros and Felipe Codevilla and Antonio Lopez and Vladlen Koltun, “CARLA: An Open Urban Driving Simulator”, 2017, pp. 1-16

[7] ASAM OpenDRIVE.specifications https://www.asam.net/fileadmin/Standards/OpenDRIVE/ASAM_OpenDRIVE_BS_V1-7-0.html. Pristupljeno 15. siječnja 2024.

[8] https://www.openstreetmap.org/. Pristupljeno 16. siječnja 2024.

Implementacija miješanja animacija (engl. *animation blending*)

Adam Ergotić
Mentor: Željka Mihajlović
Fakultet elektrotehnike i računarstva, Zagreb, Hrvatska
adam.ergotic@outlook.com

Sažetak **– U radu je opisana i prikazana metoda miješanja animacija (engl. *animation blending*). Objašnjen je postupak te implementacija navedene metode korištenjem programskog sučelja OpenGL. Također, opisani su međukoraci koji dovode do krajnjeg rješenja rada, a to je miješanje, odnosno kombinacija dviju animacija. Uvođenjem parametra/faktora miješanja, omogućeno je odrediti intenzitet jedne odnosno druge animacija tijekom kombinacije animacija. Opisani su napravljeni dodaci u programu koji omogućavaju detaljniji pregled animacija te utjecaja zglobova (engl. *joint*) koji transformiraju model tijekom animacija.**

Ključne riječi – animacije; skeletalna animacija; miješanje animacija (engl. animation blending); implementacija; OpenGL

I. UVOD

Animacija je vrlo važna tehnika koja omogućuje prikazivanje pokreta različitih tijela i živih bića. Jedno od složenijih zahtjeva animacija je napraviti animacije koje izgledaju prirodno i realistično, pogotovo kod zahtjevnijih modela (poput čovjeka).

Postoje različite metode koje služe za poboljšanje realističnosti i glatkosti animacija, a među njima je metoda miješanja animacija (engl. *animation blending*). Ova tehnika omogućuje kombinaciju više animacija odjednom unutar određenog vremenskom intervala.

Osim opisa navedene metode, u ovom radu se također prikazuje kôd implementacija metode korištenjem tehnologije OpenGL i ostalih.

II. TEHNOLOGIJE

A. *OpenGL*

OpenGL je programsko sučelje koje služi za prikazivanje 2D odnosno 3D grafike. Unutar ovog rada, OpenGL koristi se za prikazivanje modela i animacija modela. Za pisanje sjenčara (engl. *shader*) koristi se GLSL.

B. GLFW

GLFW jednostavna je biblioteka (engl. *library*) koja služi kao apstraktni sloj koji omogućuje poziv funkcija OpenGL API-ja iz programa. Također, biblioteka je napravljena tako da radi na više platformi (engl. *cross-platform*).

C. *Assimp*

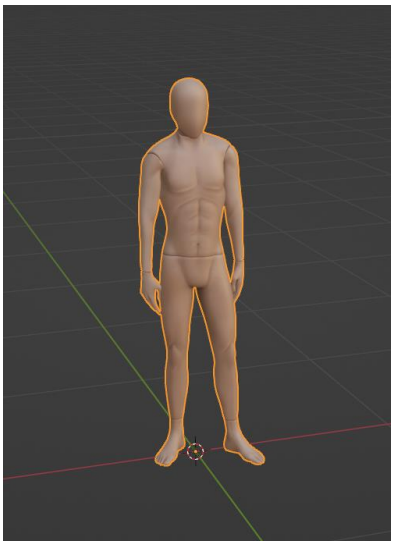
Assimp je C++ biblioteka koja služi za uvoz modela. Unutar ovog rada, Assimp se koristi za uvoz modela i animacija te svih podataka koji su potrebni za implementaciju miješanja animacija.

D. *FBX*

Za zapis modela koristi se .fbx format. FBX format omogućuje zapis modela zajedno s animacijama.

E. *Blender*

Blender se koristi za spajanje svih animacija i modela u jednu datoteku. Korišten model (Slika 1) unutar rada je s web stranice Mixamo[1].



Slika 1: Mixamo model

III. IMPLEMENTACIJA

U nastavku se nalazi postupak i opis postupka kojim se postiže metoda miješanja animacija.

A. *Opis*

Kao prvi korak, potrebno je preuzeti i konfigurirati sve potrebne biblioteke: GLFW, Assimp te GLM (biblioteka za izvođenje matematičkih operacija za grafičke potrebe).

Nakon inicijalnog koraka, prvo je potrebno učitati model s pomoću Assimp biblioteke. Objekt koji se dobije učitavanjem sadrži sve potrebne informacije za

implementaciju animacija. Nakon uvoza modela, potrebno je napisati kôd za izvođenje animacija. Implementacija animacija je obrađena u prethodnom seminaru[2]. Nakon implementacija animacije, potrebno je implementirati miješanje animacija što se opisuje u nastavku rada.

B. Implementacija

Kod uobičajenog izvođenja animacija koristi se samo jedna animacija te se između svakog ključnog okvira (engl. *keyframe*) događa interpolacija između transformacije[1] početnog ključnog okvira i transformacije krajnjeg ključnog okvira. Za miješanje animacija, osim interpolacije između početnog i krajnjeg ključnog okvira, potrebno je omogućiti kombinacija više animacije (u ovom radu omogućeno je kombinacija dvije animacije).

Za početak, potrebno je definirati faktor miješanja (engl. *blend factor*). Faktorom miješanja definiramo koliko će jedna animacija prevladavati nad drugom. Na primjer, ako je faktor jednak nuli, onda će potpuno prevladavati prva animacija. Kad je faktor jednak 0.5, dobivamo miješanje dviju animacija u kojoj obje doprinose jednako (50 % prva i 50 % druga animacija) krajnjoj animaciji koju vidimo na zaslonu. Izračun krajnje animacije možete se prikazati sljedećom formulom:

$$Finalna\ animacija = (1 - \alpha) * A_1 + \alpha * A_2$$
$$(linearna\ interpolacija)$$
$$gdje\ su\ A_1\ i\ A_2\ dvije\ različite\ animacije$$
$$\alpha\ je\ faktor\ miješanja$$

Prilikom uvoza modela pomoću biblioteke Assimp, objekt sadrži informacije o zglobovima te za svaki zglob (engl. *joint*) mapira indekse vrhova modela na koje taj zglob utječe. Naime, budući da je sjenčaru vrhova potrebno proslijediti informacije za svaki vrh zasebno, potrebno je napraviti obrnuto mapiranje (za svaki vrh odrediti indekse zglobova i koliko utječu na taj vrh, odnosno težine). Potom se ti indeksi i težine zglobova šalju u sjenčar vrhova gdje se koristi za izračun nove pozicije vrha. Izračun obrnutog mapiranja potrebno je napraviti samo jednom jer se mapiranje ne mijenja tijekom izvođenja animacija.

Nakon toga, jedino što je preostalo je izračunati krajnje transformacije za svaki zglob. To se radi tako da za svaki zglob odredimo matricu za translaciju, rotaciju i skaliranje. Za to je potrebno linearno interpolirati između dvije transformacije koristeći faktor miješanja. Nakon toga sve matrice transformacija pomnožimo čime dobijemo transformacijsku matricu svakog zgloba. Izračun transformacijskih matrica je potrebno napraviti nakon svakog iscrtavanja budući da se mijenjaju ovisno o tome koliko je vremena prošlo između iscrtavanja. Cijeli postupak je prikazan u nastavku:

[1] Odnosi se na transformacijske matrice

```
// Interpolate scaling
const aiVector3D& Scale0 = StartTransform.mScaling;
const aiVector3D& Scale1 = EndTransform.mScaling;
aiVector3D BlendedScaling = (1.0f - blendFactor) * Scale0 + Scale1 * blendFactor;
aiMatrix4x4 ScalingM;
aiMatrix4x4::Scaling(BlendedScaling, ScalingM);

// Interpolate rotation
const aiQuaternion& Rot0 = StartTransform.mRotation;
const aiQuaternion& Rot1 = EndTransform.mRotation;
aiQuaternion BlendedRot;
aiQuaternion::Interpolate(BlendedRot, Rot0, Rot1, blendFactor);
aiMatrix4x4 RotationM = aiMatrix4x4(BlendedRot.GetMatrix());

// Interpolate translation
const aiVector3D& Pos0 = StartTransform.mTranslation;
const aiVector3D& Pos1 = EndTransform.mTranslation;
aiVector3D BlendedTranslation = (1.0f - blendFactor) * Pos0 + Pos1 * blendFactor;
aiMatrix4x4 TranslationM;
aiMatrix4x4::Translation(BlendedTranslation, TranslationM);

// Combine it all
NodeTransformation = TranslationM * RotationM * ScalingM;
```

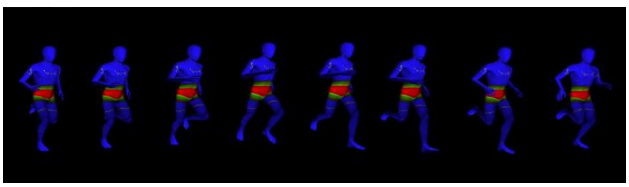
Slika 2: Interpolacija skaliranja, rotacije i translacije

Nakon izračuna transformacijski matrica za svaki zglob, iste matrice se prosljeđuju sjenčaru u obliku *uniform* varijabli. Jedino što je preostalo je izračun krajnje transformacije vrha koja se potom primjenjuje na sami vrh (Slika 3).

```
mat4 boneTransform = mat4(0.0f);

for (int j = 0; j < MAX_NUM_OF_BONES_PER_VERTEX; j++)
{
    if(j < 4)
    {
        boneTransform += uBones[boneIDs_1[j]] * boneWeights_1[j];
    }
    else
    {
        boneTransform += uBones[boneIDs_2[j-4]] * boneWeights_2[j-4];
    }
}

vFragPos = vec3(model * vec4(position, 1.0));
gl_Position = projection * view * model * boneTransform * vec4(position, 1.0f);
```

Slika 3: Primjena zglobova unutar sjenčara vrhova

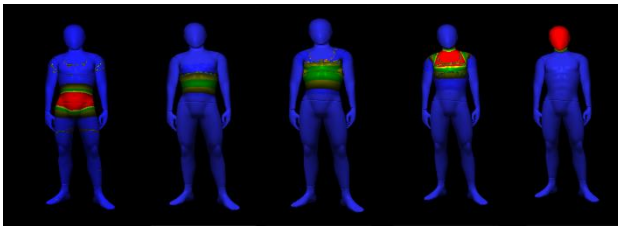
Pokretanjem programa dobije se sljedeća animacija:



Slika 4: Miješanje animacije trčanja i hodanja

U programu je također omogućeno nekoliko opcija za korisnika. S tipkom B moguće je povećati odnosno smanjiti faktor miješanja što se odmah može primijetiti na samoj animaciji na zaslonu. Tipkom N je moguće mijenjati animacije koje su uvezene zajedno s modelom. Tipkom R se rotira model.

Dodatno, omogućen je i prikaz težina svakog zgloba. U početku se prikazuju težine za nulti zglob, a pritiskom tipke *Spacebar* moguće je vidjeti i utjecaj ostalih zglobova na pojedine dijelove modela. Težine različitih zglobova moguće je vidjeti u nastavku:



Slika 5: Prikaz utjecaja različitih zglobova modela

Poveznica na videozapis u kojemu se prikazuju funkcionalnosti aplikacije: https://www.youtube.com/watch?v=oCOYA4nW01s

IV. ZAKLJUČAK

Miješanje animacija je jedna od mnogih tehnika koje omogućuju poboljšanje realističnosti animacija. Pomoću nekoliko parametara i animacija nam omogućuje stvaranje vrlo prirodne tranzicije između dvije animacije.

Osim toga, metoda je vrlo fleksibilna. Umjesto dvije animacije, moguće je napraviti miješanje tri ili više animacija ako je potrebno. Također, umjesto linearne interpolacije, mogu se koristi i druge vrste interpolacije kako bi se dobio drugačiji efekt miješanja animacija.

LITERATURA

[1] Adobe, “Mixamo”, mixamo.com

[2] A. Ergotić, “Implementacija animacija pomoću Vulkan API”, drive.google.com/file/d/1FZekVlj4ppCPoI8bQ3bXlN6yV0I4u34o/view?usp=sharing

[3] E. Meiri, “ogldev”, github.com/emeiri/ogldev

[4] freeCodeCamp.org, “Advanced OpenGL Tutorial – Skeletal Animations with Assimp”, https://www.youtube.com/watch?v=GZQkwx10p-8

[5] Assimp documentation, https://assimp.sourceforge.net/lib_html/annotated.html

[6] Glm documentation, https://glm.g-truc.net/0.9.9/api/index.html

Poveznica na repozitorij: https://github.com/Wuffll/DiplomskiProjekt

Agent-based Evacuation Simulation

Luka Marković-Đurin
University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia
lm52473@fer.hr

***Abstract* – This paper presents an implementation for an agent-based evacuation simulation. The implementation is used to analyze different problematics related to autonomous agents such as obstacle avoidance and path following.**

***Keywords* – agents, simulation, evacuation, emergency, boids**

I. INTRODUCTION

In the face of increasing urbanization and the ever-present threat of natural or human-made disasters, the optimization of evacuation processes has become a critical aspect of emergency management. Consequently, it is vital to consider the evacuation configuration of the building to best organize the evacuation upstream.

Agent-based models are computational models used for simulating actions of autonomous agents and their interactions with the environment. They provide a natural description for modeling emergent phenomena such as traffic jams and flocking.

An autonomous agent is an entity that is capable of acting independently, without direct human control, to achieve specific goals or objectives.

This paper presents an agent-based model for simulating an event of evacuation from a building. The implementation of the model is then analyzed in different scenarios to highlight the advantages and flaws of the model's approach.

II. RELATED WORK

In the paper "Algorithm and Examples of an Agent-Based Evacuation Model" [1] authors have described an autonomous agent model based on real-world parameters such as field of view, reaction time and response time.

The paper "Social force model for pedestrian dynamics" [2] explores a computational model that simulates the movements of pedestrians by considering the interaction of social forces, providing insights into collective behavior in crowded environments.

III. IMPLEMENTATION

A. Implementation Framework

The simulation was implemented using the JavaScript programming language and the p5.js library [3].

B. Environment Representation

Simulation environment is comprised of three types of entities: wall, exit and obstacle.

A wall is described with two points, signifying its starting and end points. Walls are used to construct the general layout of the building.

Exit is defined by its center position and its length. Exits are defined on walls and are typically used as objective targets by autonomous agents.

Obstacle is defined by its center position and a radius. Agents are required to steer away from all obstacles in their path.

C. Agent Model

An agent is defined by its radius, position and velocity. In each simulation frame an agent evaluates its environment and calculates forces which are acting on it. The forces are then used to calculate the agent's acceleration. Acceleration is used to update the agent's velocity which is in turn used to update its position. The described algorithm is represented by equations (1) – (4).

$$\vec{F}_{total} = calculateForces \tag{1}$$

$$\vec{a} = \frac{\vec{F}_{total}}{m} \tag{2}$$

$$\vec{v} = \vec{v}_{prev} + \vec{a} \tag{3}$$

$$\vec{p} = \vec{p}_{prev} + \vec{v} \tag{4}$$

D. Modeling Behaviour

The underlying method for calculating forces acting on an agent is based on Reynolds's boid algorithm [4]. Rather than calculating forces that push the agent, Reynolds defines a steering force which directs the agent's movement towards its target. Reynolds's steering formula is stated in equation (5).

$$\vec{F}_{steer} = \vec{v}_{desired} - \vec{v}_{current} \tag{5}$$

As the agent already keeps track of its current velocity, it is only necessary to define the desired velocity.

E. Seeking a Target

$$\vec{n} = \vec{p}_{target} - \vec{p}_{agent} \tag{6}$$

Equation (6) describes a vector ***n*** which points from the agent towards a given target.

$$\vec{v}_{desired} = m \cdot \frac{\vec{n}}{|\vec{n}|} \tag{7}$$

Equation (7) describes a formula for calculating the desired velocity where *m* represents the maximum speed of the agent. The equation describes the desired velocity as moving towards the target at maximum speed.

F. Agents Separation

To avoid collisions with other agents, each agent should keep a minimum distance from its neighbors.

$$\vec{F}_{sep} = \sum_{i=1}^{N} \frac{\vec{p}-\vec{p}_i}{|\vec{p}-\vec{p}_i|^2} \quad (8)$$

Equation (8) describes a formula for a separation force between an agent and its neighbors. The formula is only applied for neighbors that are closer than the desired separation.

G. Obstacle Avoidance

Each frame of the simulation, an agent checks for obstacles and calculates a necessary steer angle to avoid them. An obstacle must meet two conditions to be deemed significant to an agent. First it must be in range of the agent, and second the agent must be moving towards it.

$$\vec{u} = \vec{p}_{obstacle} - \vec{p}_{agent} \quad (9)$$

$$\theta_1 = \theta_{tangent} = \sin^{-1}\left(\frac{r_{obstacle}+r_{agent}}{|\vec{u}|}\right) \quad (10)$$

$$\theta_2 = \theta_{direction} = \cos^{-1}\left(\frac{\vec{u}\cdot\vec{v}}{|\vec{u}|\cdot|\vec{v}|}\right) \quad (11)$$

$$\theta_{steer} = \theta_{tangent} - \theta_{direction} \quad (12)$$

Equation (9) describes a vector $\boldsymbol{u}$ pointing from the agent to a given obstacle. To check if an agent is moving towards the obstacle, the dot product of the agent's velocity and the vector $\boldsymbol{u}$ is calculated. If the dot product is positive the agent is moving towards the obstacle and if it is negative the agent is moving away from the obstacle. If the agent is moving towards the obstacle, it is necessary to check if the agent is on the collision course with the obstacle. Equation (10) describes the formula for an angle that the tangent from agent makes with the line joining the center of the obstacle. Equation (11) describes the formula for the angle between the direction of the agent and the line joining the center of the obstacle. If θ_2 is smaller than θ_1 the agent is on a collision course with the obstacle. Equation (12) describes the formula for the steer angle. Finally, the agent must decide which way to circumvent the obstacle. If the angle between the vector $\boldsymbol{n}$ and the vector $\boldsymbol{u}$ is positive the steer angle is negated. The calculated steer angle is used to rotate the velocity vector of the agent.

Obstacle avoidance does not include avoidance from the walls. However, collision detection algorithm has been implemented to ensure that the agents cannot move through walls.

H. Path Following

A path is a series of points defining line segments. Each path has a radius defining the width of the path. An agent can be assigned a path which it will then follow. To follow the path an agent will try to predict its future position by adding a scaled version of the velocity vector to its position. The agent then calculates the normal to the path segment it is currently on. The magnitude of the normal equals the distance between the predicted position and the path segment. If that distance is greater than the path radius the agent will look to steer towards the path. To do this, the agent calculates a point on the path that is some distance away from the normal and then uses it as a seek target.

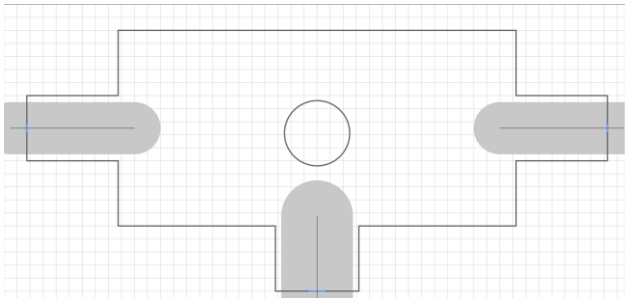


Figure 1. Scenario 1 layout

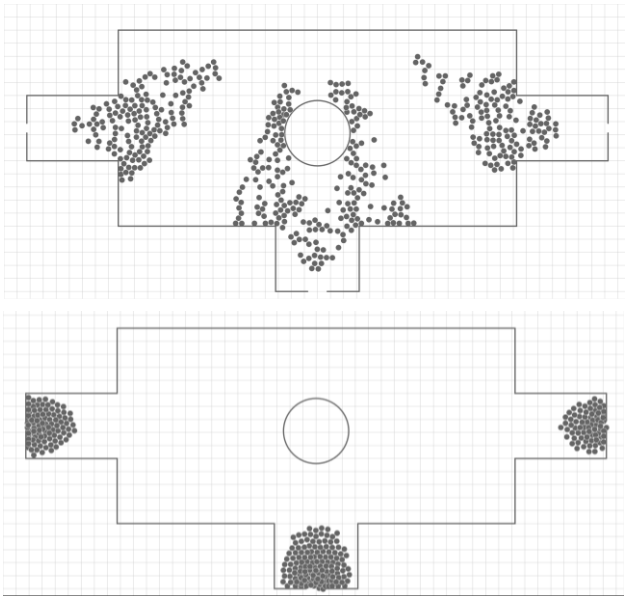


Figure 2. Screenshots at different timestemps of the simulation

I. Imporving Efficiency

To avoid the N^2 complexity of interaction checks in agent separation, a uniform grid was implemented. Each simulation frame, agents are stored in appropriate cells of the grid. The algorithm for agent separation performs checks only on the agent's cell and neighboring cells.

IV. Results and Analysis

To display some of the applications of the implemented algorithms, two scenarios were constructed.

A. Scenario 1

Scenario 1 layout can be seen in Figure 1. It contains 3 exits located in the left, right and bottom corridors. The layout also contains an obstacle in the middle, represented by a circle. To ensure better agent pathing each exit has been assigned a path, represented by a gray surface and a corresponding line. At the start of the simulation each agent was assigned the closest exit as a seek target and the corresponding path to follow. Figure 2 shows the simulation at different time steps.

Lack of paths would result in agents near the walls adjacent to the corridors experiencing a tendency to become trapped when attempting to seek the exit. Applying a path-following algorithm enabled agents to maneuver around the corners of the walls in a manner more reminiscent of human-like navigation. Similar results could potentially be achieved by defining a series of seek targets: around the corners of the corridors and in the middle of the corridors and applying the seek behavior on those points before seeking the exit.

B. Scenario 2

The layout of scenario 2 can be seen in Figure 3. It contains a single exit located in the center area and a series of corridors and paths. The purpose of this scenario was to test the application of paths for directing agent pathing through a complex building layout. When spawned the agents would determine the closest path based on their distance to the path's starting point. After an agent reaches the end of the path it would select the next path to follow in the predefined order. Figure 4 shows a screenshot of the simulation.

Rather than defining one path that spans throughout the layout, multiple paths were used. When a single path was defined, agents would focus too much on following the middle of the path which resulted in less human-like navigation. This was especially evident in the bottom right corner of the layout. Breaking the path into multiple smaller paths allowed agents more freedom while navigating the layout. However, this approach required more involvement when defining the layout and would be more challenging to incorporate with path finding algorithms.

V. Potential Improvements

The main outlier of the presented model is the lack of intelligent navigation. The alternative presented by applying a path following algorithm for navigating complex layouts could be upgraded in several ways. For example, the paths could be described as a curve instead of a collection of line segments, to better achieve navigation around corners. Instead of manually defining the path, path-finding algorithms could be implemented. The path properties such as its points and radius could be dynamically adjusted to accommodate a changing environment.

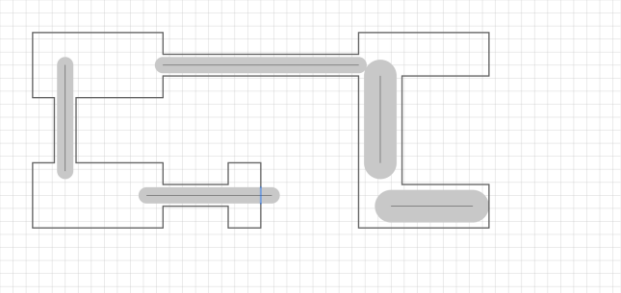


Figure 3. Scenario 2 layout

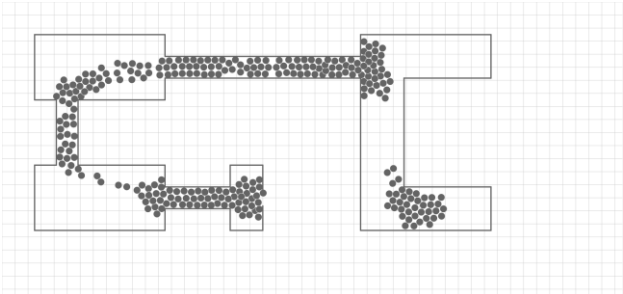


Figure 4. Screenshot of the simulation

Additionally, although the achieved results exhibit human like navigation during an emergency, the model lacks foundation in reality. A more realistic simulation could be achieved by implementing findings of other papers such as those listed in the Related Works section.

VI. Conclusion

This paper presented some of the methods used when modeling an agent-based simulation. Reynolds's boid behavior modeling was applied to different scenarios pertaining to the evacuation context. While the achieved results resemble human-like behavior during an emergency evacuation, the autonomous agents lack autonomy and individuality. More work is needed to achieve realistic modeling of human behavior during an emergency.

References

[1] X. Cui, J. Ji, X. Bai, "Algorithm and Examples of an Agent-Based Evacuation Model", 2023

[2] D. Helbing, P. Molnar, "Social force model for pedestrian dynamics", 1998

[3] https://p5js.org/

[4] C. W. Reynolds, "Steering Behaviors for Autonomous Characters", 1980

Simulacija krzna

Luka Samac
Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva, Zagreb, Hrvatska
ls52477@fer.hr

***Sažetak* – Ovaj rad će predstaviti jednu od metoda simulacije krzna gdje se simuliraju pojedinačne dlake krzna. Obraditi će potrebna ograničenja koja su potrebna kako bi se održao oblik dlake te njezina nerastezljivost. Uvedena je simulacija vjetra kako bi se bolje uočila realističnost simulacije krzna.**

Ključne riječi – krzno, simulacija, simulacija u stvarnom vremenu, 3D

I. UVOD

U računalnoj grafici, prikaz uvjerljivog krzna čest je problem. Mnogi pristupi probali su riješiti geometrijsku kompleksnost krzna. Međutim, česti su empirijski modeli ili modeli bazirani na ljudskoj kosi.

Realistične simulacije krzna su rijetka pojava u prikazima u stvarnom vremenu. Razlog je mali broj simulacijskih metoda koje su dovoljno brze i robusne da mogu simulirati tisuće dlaka krzna. Osim toga, svaka dlaka krzna bi trebala biti nerastezljiva. Ova činjenica uzrokuje probleme jer je teško jamčiti da se nijedna dlaka neće rastezati pri simulaciji deformabilnog objekta.

Mnogo metoda simulacije krzna koriste više iteracija simulacije kako bi se osigurala stabilnost pri izračunu ograničenja potrebnih za nerastezljivost krzna. Zbog ovih razloga, simulacija krzna jako je računalno skupa.

Ovaj rad će predstaviti jedan od mogućih načina simulacije krzna te implementaciju jednostavnog modela krzna.

II. METODA SIMULACIJE

Metoda simulacije korištena u ovome radu simulira krzno kao pojedinačne dlake te uvodi fizikalna ograničenja koja drže dijelove dlake skupa. Računanje simulacije krzna za svaku dlaku u simulaciji računalno je skupo te je bolja alternativa simulacija dijela krzna i dobivanje ostatka dlaka interpolacijom. Pokazalo se da je dovoljno simulirati samo oko 15-20% dlaka kako bi se dobilo uvjerljiv model.

A. Integracija

U svakom korak integracije se računa nova pozicija i brzina dlake krzna pomoću Verlet integracije. Pozicija se računa pomoću formule 1:

$$X_{i+1} = X_i + v_i \Delta t + a_i \Delta t^2 \tag{1}$$

Brzina se računa pomoću formule 2:

$$v_{i+1} = X_{i+1} - X_i \tag{2}$$

B. Fizikalna ograničenja

Fizikalna ograničenja drže krzno u određenoj poziciji. Globalna ograničenja održavaju strukturu dlake u položaju odmora. Lokalna ograničenja održavaju oblik dlake između pojedinih segmenata jedne dlake. Ograničenje duljine omogućuje da dlaka uvijek ostane jednake duljine.

Globalno ograničenje gura vrhove dlake prema poziciji odmora. Računa se prema formuli 3 gdje je i indeks segmenta dlake, X_i trenutna pozicija segmenta dlake, X_i^{rest} pozicija odmora segmenta dlake, k_{max} konstanta koja određuje maksimalnu snagu ograničenja te je iznosa između 0 i 1, a k_G snaga ograničenja koja otpada kako se približavamo rubu dlake.

$$X_i = X_i + k_G(X_i^{rest} - X_i),$$

$$k_G = k_{max} * \frac{n_{segments} - i}{n_{segments}},$$

$$k_{max} = konst \tag{3}$$

Lokalno ograničenje ispravlja deformaciju dlake ovisno o susjednim segmentima dlake. Računa se prema formuli 4 gdje je i indeks segmenta dlake, X_i trenutna pozicija segmenta dlake, X_i^{rest} pozicija odmora segmenta dlake, a k_L snaga ograničenja te je iznosa između 0 i 1.

$$X_{i-1} = X_{i-1} - \frac{1}{2} k_L(X_i^{rest} - X_i),$$

$$X_i = X_{i-1} + \frac{1}{2} k_L(X_i^{rest} - X_i),$$

$$k_L = konst \tag{4}$$

Ovo ograničenje mora biti primijenjeno više puta kako bi funkcija konvergirala.

Ograničenje duljine provjerava duljinu dva susjedna segmenta dlake te im ispravlja duljinu kako bi jednaka inicijalnog duljini između njih. Računa se prema formuli 5 gdje je i indeks segmenta dlake, X_i trenutna pozicija segmenta dlake, Δl razlika trenutne i inicijalne duljine segmenta dlake, a $\vec{n}$ normaliziran vektor od jedne do druge točke segmenta dlake.

$$\Delta X_i = -\frac{1}{2}\Delta l * \vec{n},$$

$$\Delta X_{i+1} = \frac{1}{2}\Delta l * \vec{n} \quad (5)$$

Ovo ograničenje mora biti primijenjeno više puta kako bi funkcija konvergirala.

C. Vjetar

Simuliranjem vjetra unosimo nasumičnu promjenu pozicije segmenata kose. Nasumičnost možemo unijeti podjelom vektora vjetra na više vektora kojima je malo promijenjen smjer. Korištenjem interpolacije između tih vektora uz nasumične koeficijente dobijemo konačan smjer vjetra. Konačan smjer vjetra za podjelu na dva vektora računa prema formuli 6 gdje je W smjer vjetra, k_r nasumičan koeficijent, i indeks dlake, a 20 je uzet kao neki nasumičan broj.

$$k_r = \frac{i \,\%\, 20}{20},$$

$$W = k_r\, W_1 + (1 - k_r)\, W_2 \quad (6)$$

Sada možemo odrediti magnitudu vjetra pomoću formule 7. Ovom formulom omogućujemo da se magnituda vjetra mijenja tijekom simulacije.

$$magnitude = sin^2(0.05 * frame) + 0.5 \quad (7)$$

Nova pozicija segmenta dlake se određuje prema formuli 8 gdje je i indeks segmenta dlake, a T tangenta dlake.

$$X_i = X_i - magnitude * ((TxW)xT)\Delta t^2 \quad (8)$$

D. Model osvjetljavanja

Osvjetljenje dlake krzna se dobiva određivanjem difuzne i zrcalne komponente svjetlosti.

Difuzna komponenta je dobivena pomoću Lambertovog zakona. Računa se prema formuli 9 gdje je k_d koeficijent difuzne refleksije, I_P intenzitet točkastog izvora, T tangenta dlake, a L vektor prema izvoru svjetlosti.

$$I_d = k_d I_P \sin(T, L) \quad (9)$$

Zrcalna komponenta je izračunata po uzoru na Phongov model uz male izmjene kako se bi aproksimirala defrakcija svjetla oko dlake. Računa se prema formuli 10 gdje je k_d koeficijent zrcalne refleksije, n određuje prostornu razdiobu i vezan je uz materijal, a E je vektor prema očištu.

$$I_s = k_d I_P(\sin(T, L)\sin(T, E) + (T * L)(T * E))^n \quad (10)$$

III. Zaključak

Simulacija krzna težak je problem zbog količine podataka koji se moraju obraditi u stvarnom vremenu. Nerastezljivost dlake posebno stvara probleme u simulaciji jer je potrebno više iteracija kako bi se postigla stabilnost modela. U ovom radu je korištena metoda simulacije gdje se izračun odvija samo na dijelu dlaka krzna, a ostale dlake su dobivene interpolacijom. Koristi se Verlet integracija za izračun pozicije i brzina dlaka. Uvedena su ograničenja koja osiguravaju da se održi inicijalni oblik dlake. Globalna ograničenja osiguravaju da pozicija dlake teži u poziciju odmora. Lokalna ograničenja osiguravaju da susjedni segmenti dlake nemaju veliku deformaciju u odnosu na poziciju u odmoru. Ograničenje duljine osigurava nerastezljivost dlake. Vjetar je postignut dodavanjem sile na dlaku u jednom smjeru uz određenu količinu nasumičnosti. Korišten je model svjetlosti nalik Phongovom modelu svjetlosti uz izmjene koje aproksimiraju defrakciju oko dlake.

Literatura

[1] M. Müller, T.Y. Kim, N. Chentanez, Fast Simulation of Inextensible Hair and Fur, Workshop on Virtual Reality Interaction and Physical Simulation VRIPHYS, 2012

[2] Jerome Lengyel, Emil Praun, Adam Finkelstein and Hugues Hoppe. "Real-Time Fur over Arbitrary Surfaces". In proceedings of the 2001 symposium on Interactive 3D graphics, 2001.

[3] Dongsoo Han and Takahiro Harada. RealTime Hair Simulation with Efficient Hair Style Preservation. In VRIPHYS, edited by Jan Bender, Arjan Kuijper, Dieter W. Fellner, and Eric Guerin, pp. 4551. Aire-la-Ville, Switzerland: Eurographics Association, 2012.

[4] Petrovic L., Henne M., Anderson J.: Volumetric methods for simulation and rendering of hair. In Tech. rep., Pixar Animation Studios (2005)