Odabrani projekti iz područja računalne grafike
Tehnička dokumentacija
Verzija 1.0

Studentski tim: Danijel Bajlo
Tin Blažević
Ivan Karlović
Luka Mesarić
Marko Mijolović
Maja Radočaj

Nastavnik: prof. dr. sc. Željka Mihajlović

Sadržaj

1. Uvod

Ovaj projekt skupina je projekata izvedenih u sklopu kolegija Diplomski projekt. Popis pojedinačnih projekata:

- Danijel Bajlo
  - Korištenje Unity DOTS tehnologije za poboljšanje performansa igre
- Tin Blažević
  - Simulacija fluida korištenjem sjenčara temeljena na metodi stabilnog fluida
- Ivan Karlović
  - Fizikalno zasnovano iscrtavanje scene na grafičkoj kartici uz upotrebu tehnike praćenja zrake implementirane koristeći Nvidia RTX i Vulkan 1.2
- Marko Mijolović
  - Primjena metoda dubokog učenja u računalnoj animaciji
- Maja Radočaj
  - Zamjena lica korištenjem sustava DeepFaceLab
- Luka Mesarić
  - Usporedba algoritama za izračun optičkog toka

2. Korištenje Unity DOTS tehnologije za poboljšanje performansa igre

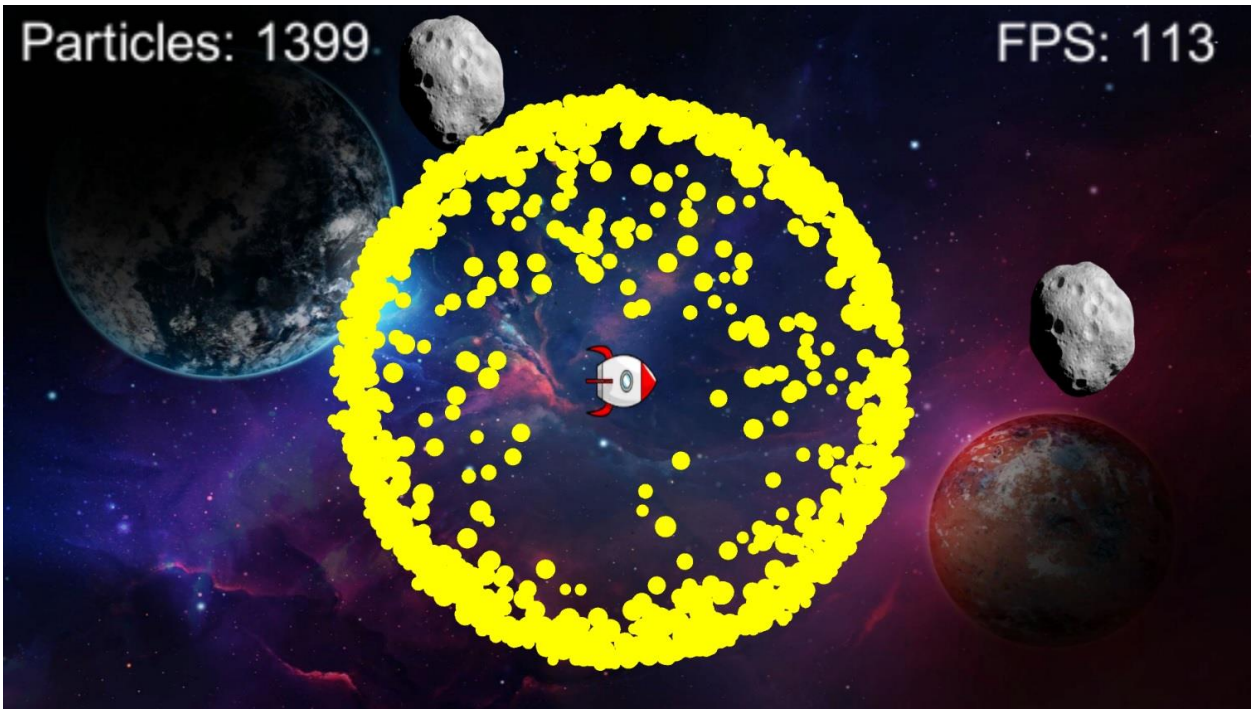
2.1 Opis razvijenog proizvoda

U sklopu projekta izrađena je igra u kojoj veliki broj objekata kruži oko igrača, prikazano na Slika 2.2. Broj raste s vremenom te je cilj projekta bio povećati maksimalan broj objekata na ekranu uz očuvanje performanse. U tu svrhu korištena je DOTS tehnologija dostupna u sklopu alata Unity.

Igrač upravlja raketom oko koje kruže čestice. Pomicanjem rakete igrač izbjegava asteroide koji dolaze te istovremeno pokušava sudariti čestice s asteroidom. Time se asteroidi uništavaju i stvaraju nove čestice. Cilj igre je trajati što duže i skupiti što više čestica.

DOTS skup tehnologija odabran je zbog srednjeg reda veličine broja objekata nad kojim se operira. Računski sjenčari, koji su alternativno mogli biti korišteni za optimizaciju, bolje su prilagođeni za veći red veličine računanja.

Tehnika GPU instanciranja (engl. *GPU Instancing*) ovdje ne bi puno pomogla jer skraćuje vrijeme prikaza, a većina vremena u ovoj igri troši se na računanja vezana uz fiziku.



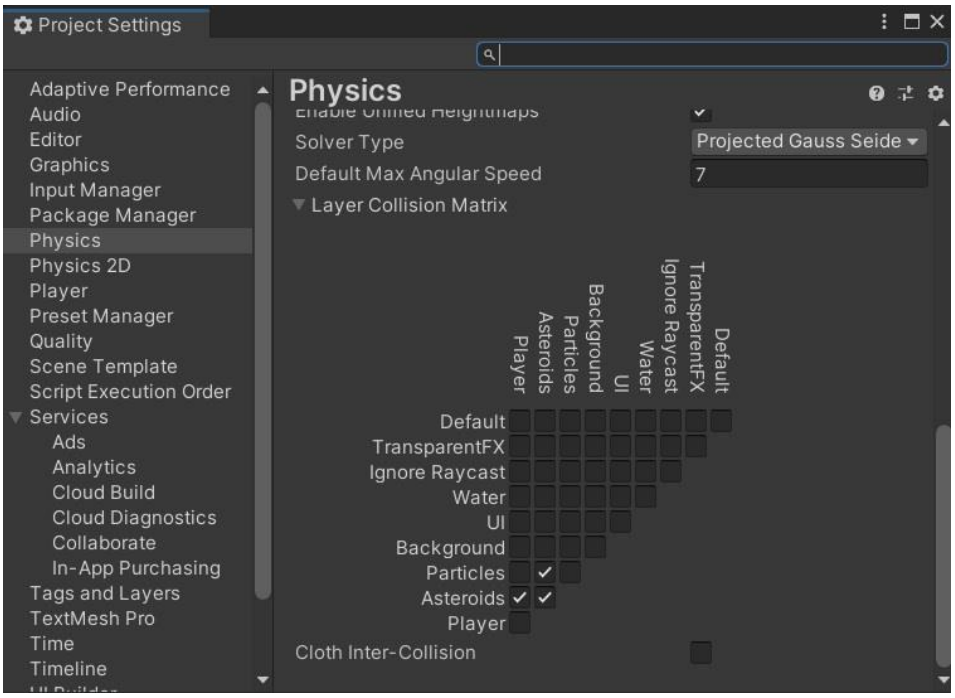
Slika 2.2. Slika igre

2.2 Tehničke značajke

2.2.1 Mehanika igre

U igri prostoje tri objekta: igrač (predstavljen sličicom rakete), čestica te asteroid. Svi objekti su trodimenzionalni te imaju sferne sudarače. Isključene su sjene te su postavke kamere namještene na ortogonalnu projekciju, kako bi se stvorio dojam dvodimenzionalnosti. Objekti rakete i asteroida nemaju svoje mreže (engl. *Mesh*), već im je kao dijete postavljen četverokut na koji je zalijepljena sličica odgovarajućeg izgleda. Korištene slike preuzete su s poveznica na referencama [4][5][6].

U svrhu poboljšanja performansi, postavljena je matrica sudara (engl. Collision matrix) tako da se nepotrebne kolizije niti ne registriraju, što prikazuje Slika 2.2. U DOTS verziji projekta ne koristi se matrica kolizija, već u samoj *PhysicsShape* komponenti se mogu odrediti slojevi s kojima će se entitet sudarati.



Slika 2.2. Matrica sudara

Asteroidi nastaju izvan ekrana te se gibaju prema centru. Postoji unaprijed određen početni broj čestica koje prate igrača. Njihovim sudaranjem s asteroidima nastane šteta na oba tijela. Kada je šteta veća od maksimalne dozvoljene, objekt se uništi. Kada je asteroid uništen, oko sebe stvara nove čestice čiji broj je određen težinom uništavanja asteroida. Pri sudaru asteroida s igračem, igra završava. U ovom projektu su sudari s igračem isključeni u svrhu lakše demonstracije performansi.

Čestice se gibaju po pravilima sličnim gravitaciji, ali s nekim modifikacijama. Ako se čestica previše približi igraču, smjer gravitacije se obrne kako bi se čestica odgurnula. Ako se previše udalji, tada se zanemaruje kvadrat udaljenosti koji se inače nalazi u nazivniku gravitacijske sile, kako bi se čestica što prije vratila igraču. Tu logiku prikazuje Slika 2.3. Osim privlačne sile,

dodaju se još tangencijalna sila te sila trenja. Tangencijalna sila vrti česticu oko igrača te se smjer vrtnje obrne nakon sudara s asteroidom. Sila trenja služi kako gibanje čestica ne bi divergiralo. U običnoj implementaciji trenje je predstavljeno faktorom kojim se množi brzina nakon primjene sile. S obzirom da se u DOTS implementaciji brzina ne može direktno podešavati, trenje je implementirano u obliku sile koja djeluje u smjeru suprotnom od vektora brzine čestice.

```
float GetForceMagnitude(float sqrDistance)
{
    float sign = 1f;
    if(!gravityReversed && sqrDistance < ParticleConstants.GRAVITY_REVERSE_THRESHOLD)
    {
        gravityReversed = true;
    }
    if (gravityReversed)
    {
        sign = -1f;
        if(sqrDistance > ParticleConstants.GRAVITY_RETURN_THRESHOLD)
        {
            gravityReversed = false;
        }
    }
    if(sqrDistance > ParticleConstants.GRAVITY_INCREASE_THRESHOLD)
    {
        //gravity ignores distance to return particle quicker to it's owner
        return ParticleConstants.G * playerMass * rigidbody.mass;
    }
    //Fg = G*m1*m2/r*r
    return sign * ParticleConstants.G * playerMass * rigidbody.mass / sqrDistance;
}
```

Slika 2.3. Iznos i orijentacija sile koja djeluje na česticu

2.2.2 Implementacija koristeći DOTS

Glavna razlika između objektno i podatkovno orijentirane implementacije ovog projekta leži u sustavima. Naime, u OO varijanti svaka čestica ima svoju skriptu koja određuje njeno ponašanje. U `Update` metodi računa se i primjenjuje privlačna sila u svakom okviru, a u metodi `OnCollisionEnter` definira se ponašanje u slučaju sudara s asteroidom.

U podatkovno orijentiranoj varijanti čestice imaju samo komponente u kojima su pohranjeni podatci, a sustavi `ParticleAttractionSystem` i `CollisionSystem` zaduženi su da pronalaze sve entitete nad kojima trebaju obaviti operacije te ih izvrše. [1]

Usporedbe obične i DOTS implementacije privlačenja čestica prikazuju Slika 2.4 i Slika 2.5.

```
void Update()
{
    Vector3 direction = playerTransform.position - transform.position;
    float sqrDistance = Vector3.SqrMagnitude(direction);
    if (sqrDistance > ParticleConstants.MIN_DISTANCE_FOR_GRAVITY)
    {
        Vector3 normalizedDirection = direction.normalized;
        float magnitude = GetForceMagnitude(sqrDistance);
        rigidbody.AddForce(normalizedDirection*magnitude);
        Vector3 tangent = Vector3.Cross(normalizedDirection, Vector3.up);
        rigidbody.AddForce(tangent*ParticleConstants.TANGENTIAL_MAGNITUDE*rotationDirection);
        rigidbody.velocity = new Vector3(rigidbody.velocity.x * ParticleConstants.DAMPING,
                                rigidbody.velocity.y * ParticleConstants.DAMPING,
                                rigidbody.velocity.z * ParticleConstants.DAMPING);
    }
}
```

Slika 2.4. Metoda za dodavanje sile česticama u običnoj implementaciji

```
0 references
protected override void OnUpdate()
{
    Entities.WithoutBurst()
        .ForEach(
    (
        ref PhysicsVelocity physicsVelocity,
        ref PhysicsMass physicsMass,
        ref ParticleData particleData,
        in Translation translation) =>
    {
        float3 velocityVector = physicsVelocity.GetLinearVelocity(physicsMass, new Translation { Value = float3.zero },
            new Rotation { Value = quaternion.identity }, float3.zero);
        float velocitySquared = Utils.Float3SquareDistance(velocityVector, float3.zero);
        float velocity = Mathf.Sqrt(velocitySquared);

        //APPLY FRICTION
        float3 frictionVector = GetFrictionVector(velocityVector, velocity);
        physicsVelocity.ApplyLinearImpulse(physicsMass, frictionVector);

        //APPLY ATTRACTION
        Translation playerTranslation = EntityManager.GetComponentData<Translation>(playerEntity);
        float3 direction = playerTranslation.Value - translation.Value;
        float sqrDistance = Utils.Float3SquareDistance(playerTranslation.Value, translation.Value);
        float3 distance = Mathf.Sqrt(sqrDistance);
        float3 normalizedDirection = direction / distance;
        float magnitude = GetForceMagnitude(sqrDistance, particleData, 1 / physicsMass.InverseMass);
        physicsVelocity.ApplyLinearImpulse(physicsMass, normalizedDirection * magnitude);

        //APPLY TANGENTIAL FORCE
        float3 tangent = Utils.Float3CrossProduct(normalizedDirection, vectorUp);
        physicsVelocity.ApplyLinearImpulse(physicsMass, tangent * particleData.rotationDirection
            * ParticleConstants.TANGENTIAL_MAGNITUDE);

    }).Run();
}
```

Slika 2.5. Metoda za dodavanje sile česticama u DOTS implementaciji

Sustav izvršava odgovarajući kod nad svim entitetima koji imaju odgovarajuće komponente, u ovom slučaju `PhysicsVelocity`, `PhysicsMass`, `Translation` te `ParticleData`. Posljednja od ovih komponenata nije jedna od standardnih, već je stvorena u sklopu ovog projekta kako bi sadržavala podatke o stanju čestice te razlikovala taj entitet od svih ostalih koji imaju iste komponente. Komponente koje ne sadrže podatke, već služe za razlikovanje entiteta od drugih, po dogovoru se nazivaju oznake (engl. *Tags*). Takvu komponentu ima entitet igrača. Komponentu koja sadrži podatke specifične za čestice prikazuje Slika 2.6.

```
[GenerateAuthoringComponent]
8 references
public struct ParticleData : IComponentData
{
    public bool gravityReversed;
    public float rotationDirection;
    public float totalHitPoints;
    public float currentHitPoints;
}
```

Slika 2.6. Komponenta koja sadrži podatke o česticama

Za primjenu sile nad entitetom koristi se metoda `ApplyLinearImpulse` iz domene `Unity.Physics.Extensions`. [2] Komponenta `Translation` označena s ključnom rječju `in`, jer se njena vrijednost ne mijenja, već se samo čita pri računanju vektora smjera sile. Te oznake omogućavaju upravitelju poslova da bolje paralelizira zadatke.

Najizazovniji dio projekta bila je implementacija reakcije na kolizije. Naime, za razliku od obične implementacije, gdje se koristi vrlo praktična metoda `OnCollisionEnter`, u DOTS implementaciji potrebno je stvoriti novi `JobComponentSystem` koji u sebi sadrži strukturu koja implementira sučelje `ICollisionEventsJob` u čijoj metodi `Execute` se izvodi kôd kao reakcija na koliziju. U toj metodi još je potrebno za oba entiteta koja se sudaraju otkriti koji su to zapravo entiteti, a to se obavlja provjeravajući ima li entitet neku komponentu putem strukture `ComponentDataFromEntity`, parametriziranu po toj komponenti. [3] Očigledno je da je implementacija jako nepraktična i zahtjeva veliku količinu sličnog koda koji se ponavlja. Dio kôda potreban za određivanje tipa entiteta koji su se sudarili prikazuje Slika 2.7.

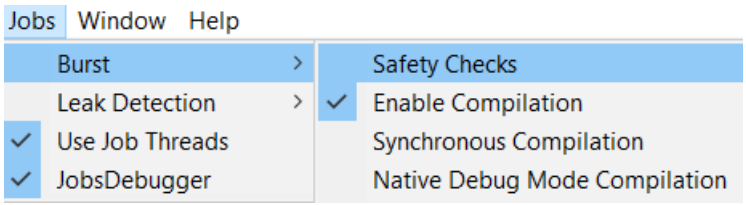
```
private struct CollisionJob : ICollisionEventsJob
{
    public ComponentDataFromEntity<AsteroidData> AsteroidDataGroup;
    public ComponentDataFromEntity<ParticleData> ParticleDataGroup;
    public ComponentDataFromEntity<PlayerTag> PlayerTagGroup;
    public ComponentDataFromEntity<Translation> TranslationGroup;
    public ComponentDataFromEntity<CompositeScale> CompositeScaleGroup;
    public ComponentDataFromEntity<PhysicsMass> PhysicsMassGroup;
    public EntityCommandBuffer CommandBuffer;
    public Entity particlePrefab;
    public const float maxMass = ParticleConstants.PARTICLE_AVG_MASS + ParticleConstants.PARTICLE_MASS_VARIATION;
    0 references
    public void Execute(CollisionEvent collisionEvent)
    {
        //Debug.Log($"Collision between entities { collisionEvent.EntityA.Index } and { collisionEvent.EntityB.Index }");
        Entity entityA = collisionEvent.EntityA;
        Entity entityB = collisionEvent.EntityB;

        bool isAsteroidA = false, isAsteroidB = false;
        bool isParticleA = false, isParticleB = false;

        bool isPlayerA = PlayerTagGroup.HasComponent(entityA);
        bool isPlayerB = PlayerTagGroup.HasComponent(entityB);

        if (!isPlayerA) isAsteroidA = AsteroidDataGroup.HasComponent(entityA);
        if (!isPlayerB) isAsteroidB = AsteroidDataGroup.HasComponent(entityB);

        if (!isPlayerA && !isAsteroidA) isParticleA = ParticleDataGroup.HasComponent(entityA);
        if (!isPlayerB && !isAsteroidB) isParticleB = ParticleDataGroup.HasComponent(entityB);
```

Slika 2.7. Isječak iz sustava za koliziju

Bitno je i spomenuti da je u komponenti PhysicsShape potrebno namjestiti opciju *Raises Collision Events* kako bi ovaj sustav registrirao koliziju.

2.2.3 Usporedbe performansi

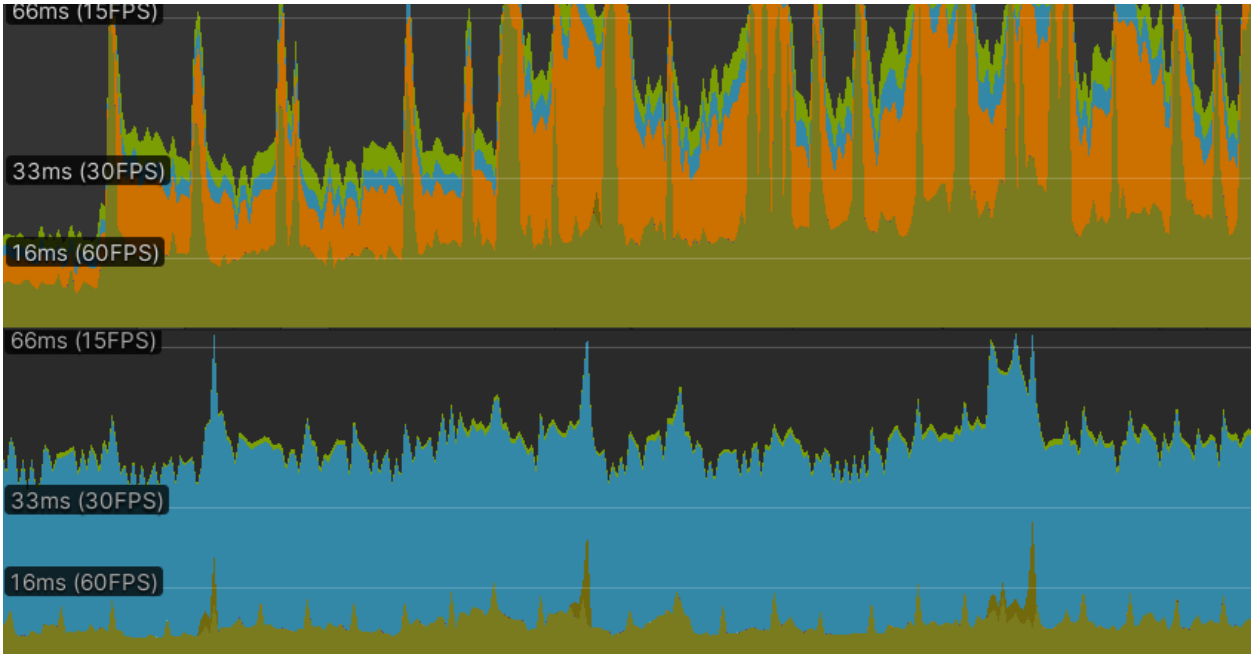
Iako je u sustavu za gibanje čestica korištena metoda `Entities.WithoutBurst()`, jer inače ne dopušta da se unutar metode koriste reference, performanse su znatno bolje kada se u opcijama uključi korištenje Burst kompajlera, što prikazuje Slika 2.8.



Slika 2.8. Uključivanje Burst kompajlera u opcijama

Mjerenja su provedena na procesoru Intel i5-9300H i grafičkoj kartici Nvidia GTX 1650. Broj okvira u sekundi (engl. *Frames per second*) dobiven je korištenjem izraza `1.0f / Time.smoothDeltaTime`. U običnoj implementaciji FPS padne ispod 60 u prisutnosti približno 3000 čestica. Igra postaje gotovo neigriva sa 10.000 prisutnih čestica. U DOTS implementaciji FPS padne ispod 60 tek oko 10.000 čestica. Sa 20.000 čestica FPS padne ispod 30, a s 30.000 čestica padne na 15. Igra postaje gotovo neigriva tek nakon 50.000 čestica. Osim što je FPS veći, u DOTS implementaciji puno je i stabilniji. U običnoj implementaciji postoje nagli skokovi u količini računanja pa FPS može varirati i do 20 okvira od trenutka do trenutka.

DOTS implementacija ima stabilniji FPS čak i pri niskim vrijednostima te najčešće varira u samo nekoliko okvira. Rezultate mjerenja korištenjem alata za profiliranje (engl. Profiler) prikazuje Slika 2.9.



Slika 2.9. Usporedba stabilnosti performanse obične (gore) i DOTS (dolje) implementacije

Iz razloga što je videoigra originalno napravljena za mobilne uređaje, provedeno je i mjerenje na uređaju Galaxy A30s. U običnoj implementaciji FPS padne ispod 30 već sa 1200 čestica, a dostigne vrijednost od 15 u prisutnosti 2500 čestica. DOTS implementaciji FPS padne ispod 30 tek oko 2500 čestica te također vrijedi da je puno stabilniji, isto kao i na računalnoj verziji.

2.3 Upute za korištenje

Projekt se može preuzeti s GitLab repozitorija na referenci [7] te pokrenuti u alatu Unity. Za korištenje DOTS verzije projekta, potrebno je direktorij *Assets* zamijeniti direktorijem *DOTS_Assets*. Također, nužno je instalirati sve potrebne pakete koji omogućavaju korištenje DOTS tehnologije.

Nakon izgradnje, aplikacija se koristi na način da se klikom miša te povlačenjem po ekranu pomiče objekt igrača. Cilj igre je uništiti asteroide time što se u njih zabijaju čestice koje prate igrača. Pritiskom na tipku razmak, dodaje se tisuću čestica što se koristi za testiranje performansi. Pritiskom na tipku „Esc“ izlazi se iz igre.

2.4 Literatura

[1] Unity manual, ECS tutorial, https://learn.unity.com/tutorial/entity-component-system#
[2] Turbo Makes Games, How to add force in DOTS physics, https://www.youtube.com/watch?v=3h0nJXPuUm0
[3] Stack exchange, How to detect collisions in Unity ECS, https://gamedev.stackexchange.com/questions/190132/how-to-detect-collisions-in-unity-ecs
[4] Pinterest, A cartoon moon rocket, https://www.pinterest.cl/pin/325596248058982494/
[5] Wallpaper-loveyou, Space wallpaper, https://wallpaper-loveyou.blogspot.com/2020/07/cool-wallpapers-horizontal.html
[6] Space reference, Asteroid image, https://www.spacereference.org/asteroid/3200-phaethon-1983-tb
[7] Gitlab, repozitorij projekta, https://gitlab.com/dBajlo/diplomskiprojekt

3. Simulacija fluida korištenjem sjenčara temeljena na metodi stabilnog fluida

3.1 Opis razvijenog proizvoda

Ovaj projekt fokusira se na simulaciju fluida temeljenu na metodi stabilnog fluida [1] korištenjem računskih sjenčara unutar programskog razvojnog okruženja Unity [2]. Metoda stabilnog fluida je Eulerova metoda što znači da koristi cjelobrojnu rešetku za reprezentaciju stanja fluida. Vektorska polja se spremaju eksplicitno u memoriji, za razliku od Lagrangeovih metoda koje koriste sustave čestica. Ta činjenica čini metodu pogodnom za implementaciju korištenjem sjenčara.

Implementacija se temelji na [3] uz opisane mogućnosti proširenja iz [4]. Detaljniji pregled matematičke podloge, Navier-Stokesovih jednadžbi i operacija metode stabilnih fluida dan je u [5].

3.1.1 Računski sjenčari

Računski sjenčari su kao i ostali sjenčari programi se izvršavaju na grafičkoj kartici, ali izvan uobičajenog grafičkog protočnog sustava. Mogu se koristiti za masovnu paralelizaciju općenitog računanja ili za ubrzavanje nekih dijelova iscrtavanja. Unity značajno olakšava pisanje računskih sjenčara time što se automatski brine za dijelove koda koji postavljaju sustave potrebne za pokretanje istih.
Sintaksa koju Unity koristi za računske sjenčare je DX11 HLSL (eng. High Level Shading Language). Funkcije definirane unutar računskih sjenčara nazivaju se jezgrama (eng. kernel). Svaki računski sjenčar mora imati barem jednu jezgru. Koriste se posebne strukture podataka – računski međuspremnici (eng. compute buffer) koji služe za spremanje proizvoljnih podataka, za razliku od uobičajenih sjenčara gdje su tipovi podataka strogo zadani. Moguće je koristiti i teksture čime se postiže mogućnost iscrtavanja.

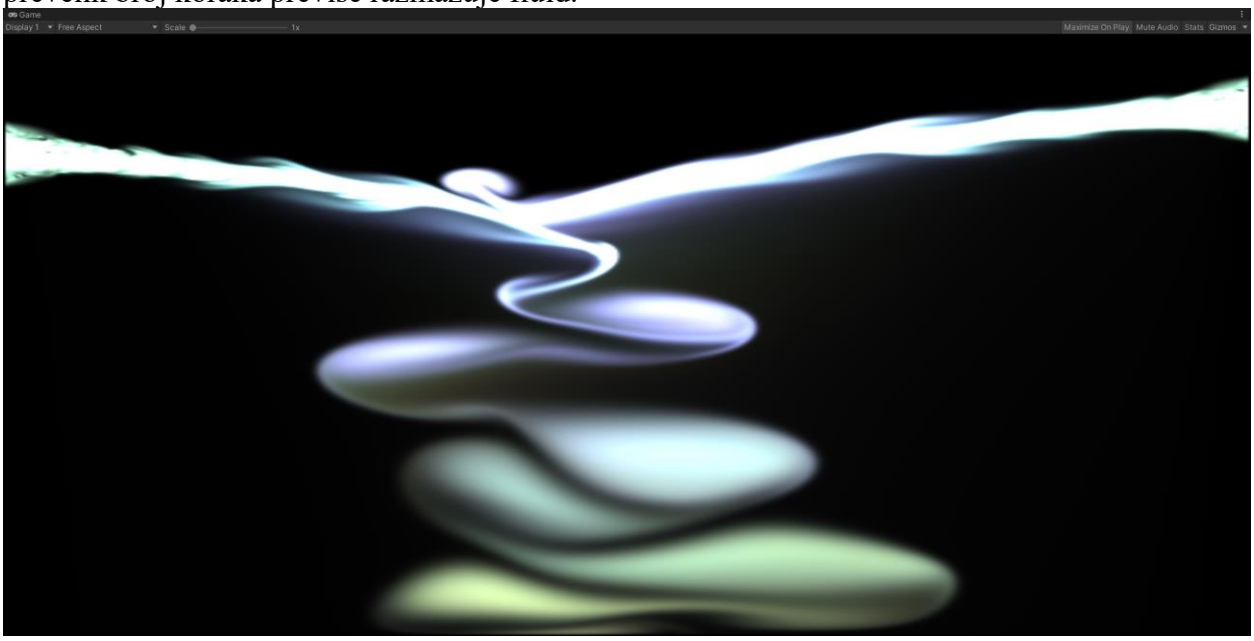
3.1.2 Definiranje operacija stabilnog fluida

Stabilni fluidi definiraju nekoliko operacija kojima se implementira specifikacija dana Navier-Stokes jednadžbama – advekcija, difuzija, projekcija i popravljanje granica. Dodatno se koristi nekakva numerička iterativna metoda za rješavanje linearnih sustava koji se pojavljuju. U originalnom radu to je Gauss-Seidelova metoda koja nije pogodna za paralelizaciju. Jacobijeva metoda sporije konvergira ali može ju se lako paralelizirati i koristiti u sjenčarima. Svaka od tih operacija je skupa kada se izvršava slijedno na procesoru. Korištenjem računskih sjenčara, svaka operacija implementira se kao zasebna jezgra.
Za svaki dio stanja fluida, odnosno za svako vektorsko polje stvara se zasebni spremnik, uz neke dodatne pomoćne spremnike zbog specifičnosti rada sa sjenčarima. Jacobijev rješavač sustava treba pomoćni spremnik za spremanje međurezultata dok ne provede sve korake. Korištenje sjenčara omogućava lako korištenje boja za prikaz stanja.
Zasebna skripta objedinjuje sve jezgre i poziva ih nad stvorenim spremnicima. Potrebno je definirati vlastiti protočni sustav korištenjem posebnog spremnika za naredbe. Svi pozivi jezgri spremaju se u obliku naredbi u taj spremnik. Spremnik naredbi je povezan s glavnom kamerom koja emitira događaj kada završi iscrtavanje trenutne sličice preko kojeg se poziva izvršavanje svih naredbi u spremniku. Jedna od pomoćnih jezgri izravno preslikava stanje fluida na teksturu

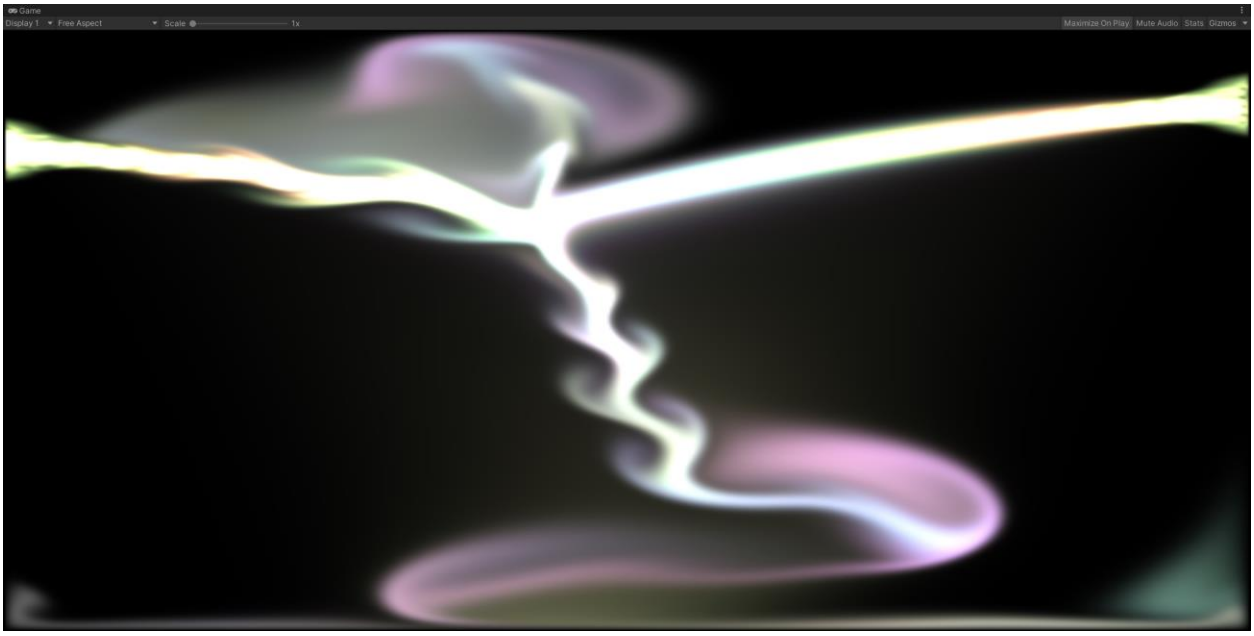
koja je nalijepljena na glavnu kameru. Time se postiže jednostavan prikaz dvodimenzionalnog fluida.

3.1.3 Rezultati

Na slikama (3.1) do (3.6) prikazani su različiti rezultati za veličinu domene 512, uz različiti broj koraka Jacobi rješavača i prisutnost viskoznosti i difuzije te različit broj izvora. Moguće je primijetiti različite uzorke i ponašanje fluida ovisno o broju koraka rješavača. Poznato je da ova metoda pati od numeričke disipacije. Premalen broj koraka ne daje dovoljnu točnost, dok prevelik broj koraka previše razmazuje fluid.

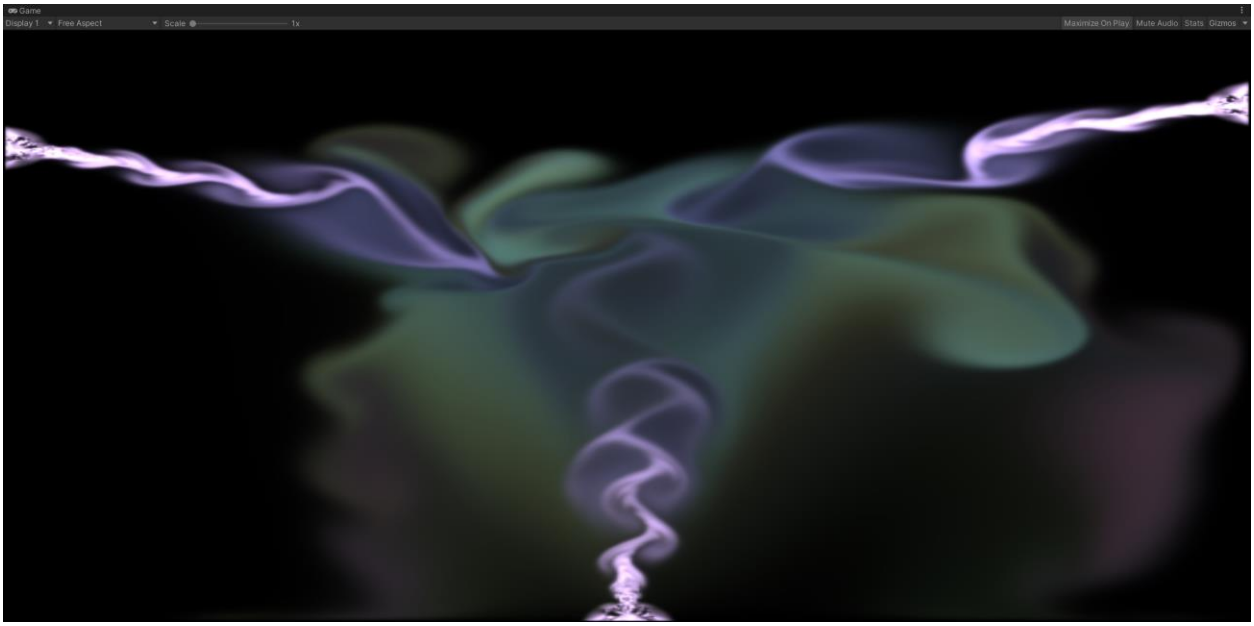


Slika 3.1. Dva izvora i 40 koraka

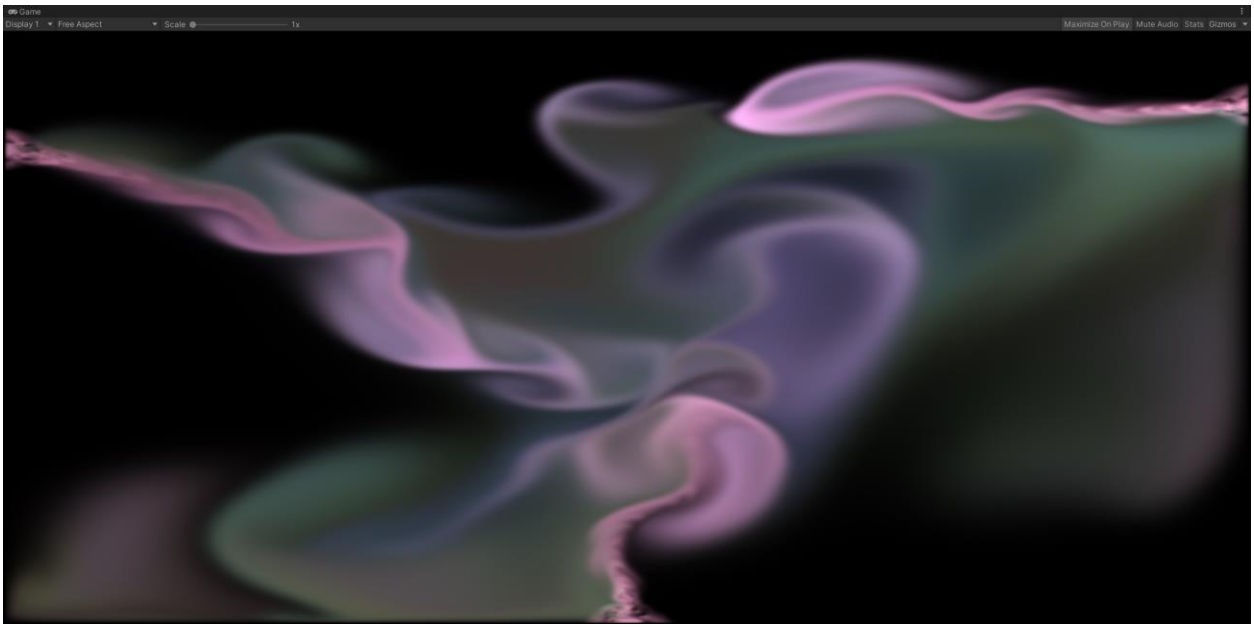


Slika 3.2. Dva izvora i 80 koraka

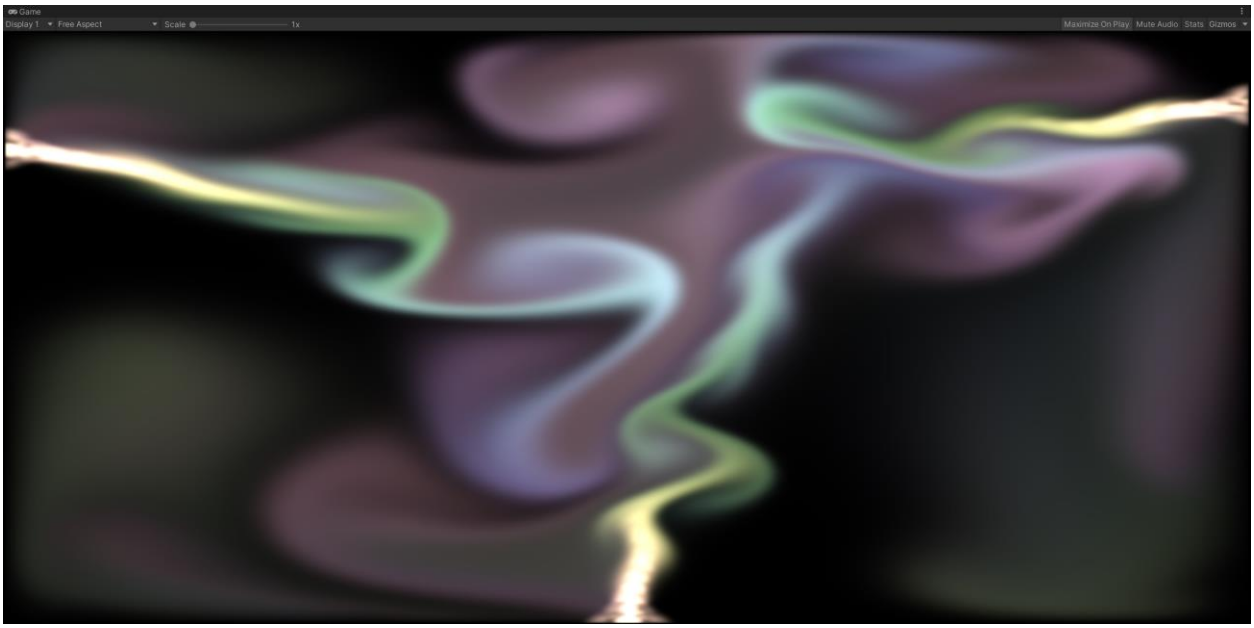
Slika 3.3. Dva izvora i 180 koraka



Slika 3.1. Tri izvora i 40 koraka



Slika 3.2. Tri izvora i 80 koraka



Slika 3.3. Tri izvora i 180 koraka

3.1.4 Mogućnosti proširenja

3D prostor

Stanje fluida lako se može proširiti na tri dimenzije korištenjem 3D tekstura umjesto dvodimenzionalnih. Međutim potrebno je prilagoditi iscrtavanje. Najjednostavniji način iscrtavanja je bacanjem zraka (eng. raycasting). Svaka zraka bi akumulirala vrijednosti prolaskom kroz model fluida, a te vrijednosti bi eksponencijalno opadale nakon prve pogođene.

Dinamičke prepreke

Prepreke proizvoljnog oblika mogu se dodati vokselizacijom objekata koji predstavljaju

prepreke. Za ažuriranje tih novih granica potrebno je i ponavljati vokselizaciju. Dinamički objekti se često aproksimiraju jednostavnim geometrijskim oblicima, odnosno tijelima. Na dobivenim novim granicama potrebno je računati normale i postupak postaje nešto složeniji od onog na granicama domene.

Dim

Brzina sama po sebi nije dovoljna za zanimljiva gibanja fluida. Za dim su potrebne dodatne veličine kao što je temperatura. Operacije u postupku također moraju zadovoljavati nove jednadžbe s temperaturom.

Vatra

Vatra je slična dimu i uvodi dodatni veličinu zvanu reakcijska koordinata koja mjeri količinu goriva u nekoj točki domene i konstantno se smanjuje. Vrijednost 1 znači da je vatra tek upaljena, dok vrijednost 0 znači da je svo gorivo potrošeno i vatra se gasi.

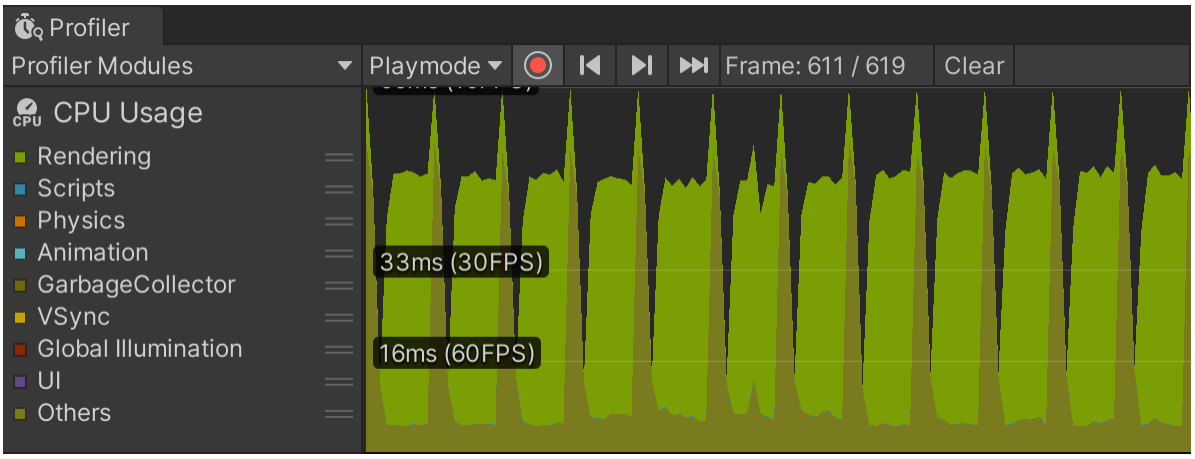
Voda

Voda je značajno drugačija od vatre i dima jer za razliku od njihovih gustoća, ovdje zanimljiv fenomen predstavlja površina fluida koju je potrebno drugačije iscrtati. Koristi se *level set* tehnika koja predstavlja dodatno svojstvo fluida koje služi za predstavljanje stvarne domene fluida.

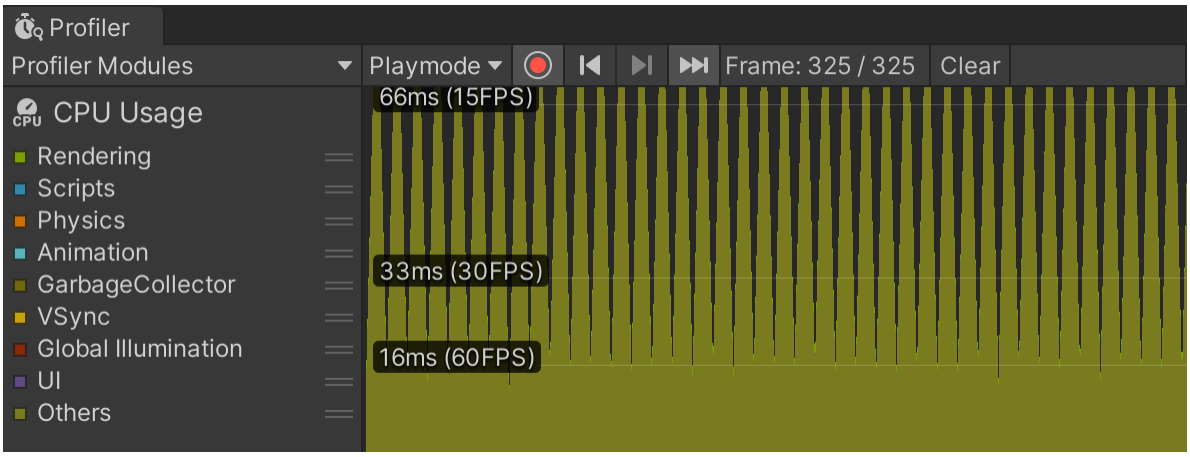
3.2 Tehničke značajke

Unutar Unity projekta nalazi se scena Simple2DFluid u direktoriju *Scenes*. Unutar scene je objekt StableFluid2D s parametrima simulatora. Mogu se mijenjati veličina domene simulacije, veličina iscrtane teksture, broj koraka Jacobi rješavača i svojstva izvora fluida. Pritiskom na tipku Play pokreće se program.

Program je pokrenut na računalu s četverojezgrenim Intel i7 procesorom i Nvidia MX150 grafičkom karticom. Broj sličica u sekundi uvelike ovisi o veličini domene simulacije i o broju koraka Jacobi rješavača. Za 40 koraka dobivaju se zadovoljavajući rezultati i većina vremena se gubi na iscrtavanje (slika 3.7). Za 180 koraka izračun stanja u računskim sjenčarima postaje usko grlo (slika 3.8).



Slika 3.7. Analiza performansi za 3 izvora i 40 koraka



Slika 3.8. Analiza performansi za 3 izvora i 180 koraka

3.3 Literatura

[1] J. Stam, "Stable Fluids", http://graphics.stanford.edu/courses/cs468-05-fall/Papers/p121-stam.pdf, 1999. godina

[2] Unity dokumentacija, "Compute shaders", https://docs.unity3d.com/Manual/class-ComputeShader.html, prisupano 1.2022.

[3] M. J. Harris, "GPU Gems - Chapter 38. Fast Fluid Dynamics Simulation on the GPU ", https://developer.download.nvidia.com/books/HTML/gpugems/gpugems_ch38.html, 2004. godina

[4] K. Crane, I. Llamas, S. Tariq, "GPU Gems 3 - Chapter 30. Real-Time Simulation and Rendering of 3D Fluids ", https://developer.nvidia.com/gpugems/gpugems3/part-v-physics-simulation/chapter-30-real-time-simulation-and-rendering-3d-fluids, 2006. godina

T. Blažević, ''Diplomski seminar - Simulacija fluida metodom materijalne tocke i ˇ programski jezik Taichi'' 2021. godina

4. Fizikalno zasnovano iscrtavanje scene na grafičkoj kartici uz upotrebu tehnike praćenja zrake implementirane koristeći Nvidia RTX i Vulkan 1.2

4.1 Opis razvijenog proizvoda

Proizvod je malen program koji nasumično generira trodimenzionalno polje voksela te ih prikazuje koristeći algoritam praćenja zrake i fizikalno zasnovano iscrtavanje.

Korisnik se može slobodno kretati kroz prostor te podešavati parametre materijala na vokselima (metalnost i hrapavost). U sceni se nalazi nekoliko izvora svjetla koji osvjetljavaju voksele.

4.2 Tehničke značajke

Program je ostvaren upotrebom Vulkan APIja i njegovih proširenja u vidu ekstenzija za omogućavanje sklopovsko ostvarenje algoritma praćenja zrake. Prilikom pokretanja program stvara predefinirani broj trodimenzionalnih segmentata koji svaki mogu sadržavati predefinirani broj voksela[1]. Algoritam nasumično odabire koji će vokseli biti prisutni, a koji ne. Nakon što su svi vokseli generirani, program prolazi kroz svaki segment te u svakoj od tri dimenzije prolazi kroz dvodimenzionalne presjeke voksela te generira mrežu trokuta koristeći pohlepni algoritam.

```
std::vector<uint32_t> ChunkGenerator::generateXSlice(const bool blocks[CHUNK_PADDED_SIZE][CHUNK_PADDED_SIZE][CHUNK_PADDED_SIZE], int32_t side) const {
    std::vector<uint32_t> slices;
    for (uint32_t i = 1; i < CHUNK_SIZE + 1; ++i) {
        bool meshed[CHUNK_SIZE][CHUNK_SIZE] = {};
        for (uint32_t j = 1; j < CHUNK_SIZE + 1; ++j) {
            for (uint32_t k = 1; k < CHUNK_SIZE + 1; ++k) {
                if (meshed[j - 1][k - 1] == false
                    && blocks[i][j][k] == true
                    && blocks[i + side][j][k] == false) {

                    uint32_t dim1 = k - 1;
                    do {
                        ++dim1;
                    } while (meshed[j - 1][dim1 - 1] == false
                        && blocks[i][j][dim1] == true
                        && blocks[i + side][j][dim1] == false
                        && dim1 < CHUNK_SIZE + 1);

                    uint32_t dim2 = j + 1;
                    do {
                        bool holeOrObstructionFound = false;
                        for (uint32_t m = k; m < dim1; ++m) {
                            if (blocks[i][dim2][m] == false
                                || blocks[i + side][dim2][m] == true) {
                                holeOrObstructionFound = true;
                                break;
                            }
                        }

                        if (holeOrObstructionFound) {
                            break;
                        }

                        ++dim2;
                    } while (dim2 < CHUNK_SIZE + 1);

                    updateMeshed(meshed, j, k, dim1, dim2);

                    if (side == -1) {
                        uint32_t v1 = getVertexIndex(i - 1, k - 1, j - 1);
                        uint32_t v2 = getVertexIndex(i - 1, k - 1, dim2 - 1);
                        uint32_t v3 = getVertexIndex(i - 1, dim1 - 1, j - 1);
                        uint32_t v4 = getVertexIndex(i - 1, dim1 - 1, dim2 - 1);

                        pushRectangle(slices, v1, v2, v3, v4);
                    }
                    else {
                        uint32_t v1 = getVertexIndex(i, k - 1, j - 1);
                        uint32_t v2 = getVertexIndex(i, dim1 - 1, j - 1);
                        uint32_t v3 = getVertexIndex(i, k - 1, dim2 - 1);
                        uint32_t v4 = getVertexIndex(i, dim1 - 1, dim2 - 1);

                        pushRectangle(slices, v1, v2, v3, v4);
                    }
                }
            }
        }
    }
    return slices;
}
```

Slika 4.1. Kod za generiranje trokuta svih presjeka s normalama paralelnima X osi.

Takvi se podaci predaju grafičkoj kartici koja iz njih generira akceleracijske strukture koje se kasnije koriste za algoritam praćenja zrake. Sam čin pretrage sjecišta pojedinih zraka s trokutima sklopovski je ostvaren, dok se u određenim uvjetima pokreću razne vrste sjenčara. Sjenčar generator zraka služi za generiranje inicijalnih zraka. Generira se jedna zraka za svaki slikovni element na ekranu se zraka usmjerava sukladno perspektivnoj projekciji. Nakon što se izračunaju svih sudari, ako nije došlo do sudara ni sa jednim trokutom, poziva se sjenčar promašaja. Sjenčar promašaja vraća boju pozadine. Ako je do sudara došlo, nad sudarom najbližim ishodištu zrake poziva se sjenčar najbližeg sudara. Zadatak sjenčara najbližeg sudara jest da izračuna konačnu vrijednost boje slikovnog elementa u toj točci. Za izračun se koristi fizikalno zasnovan model osvjetljenja uz upotrebu Cook-Torrance BDRF-a. Za uzrokovanje se koristi normalna distribucijska funkcija Trowbridge-Reitz GGX[2].

```
float DistributionGGX(vec3 N, vec3 H, float roughness)
{
    float a = roughness*roughness;
    float a2 = a*a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH*NdotH;

    float nom   = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;

    return nom / denom;
}
```

Slika 4.2. Implementacija Trowbridge-Reitz GGX normalne distribucije

Za simulaciju odsjaja ovisno o kutu gledanja koristi se Schlickovo pojednostavljenje Fresnelove formule koje značajno smanjuje količinu potrebnog računanja[3].

```
vec3 fresnelSchlick(float cosTheta, vec3 F0)
{
    return F0 + (1.0 - F0) * pow(clamp(1.0 - cosTheta, 0.0, 1.0), 5.0);
}
```

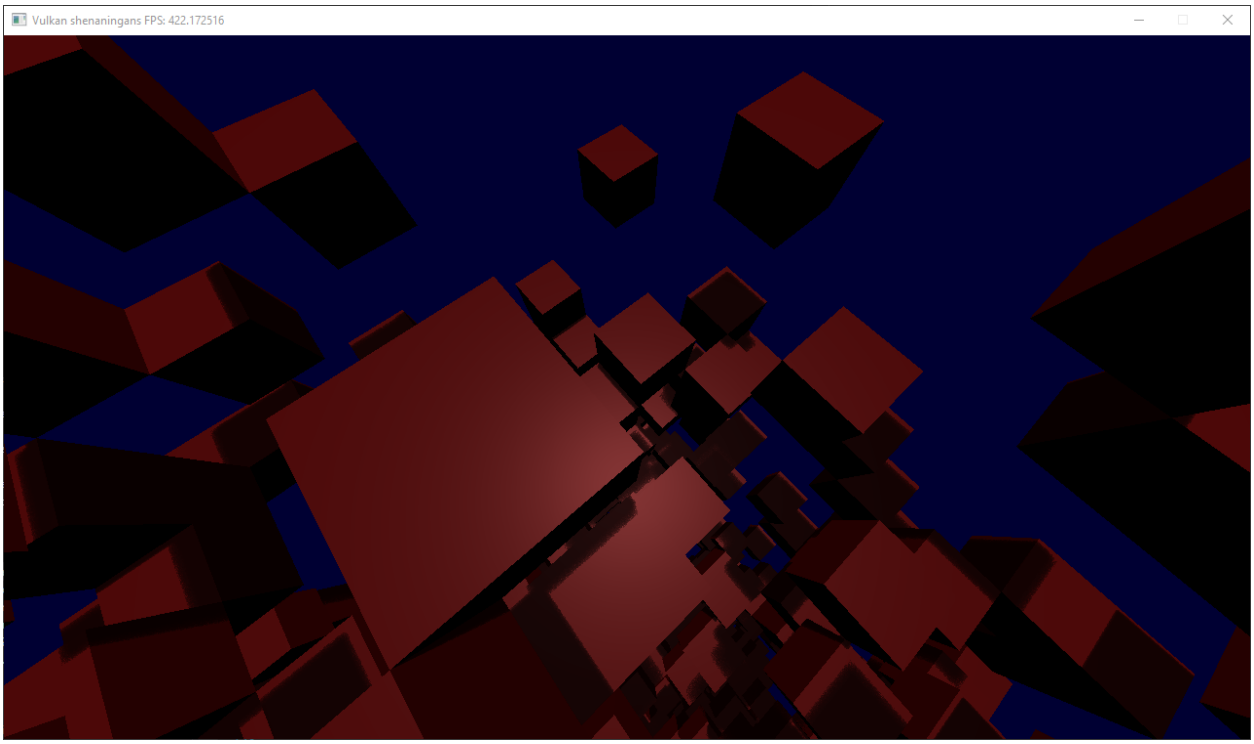
Slika 4.3. Schlick-Fresnel pojednostavljena funkcija

Za simulaciju samosjenčanja koristi se Smithova metoda.

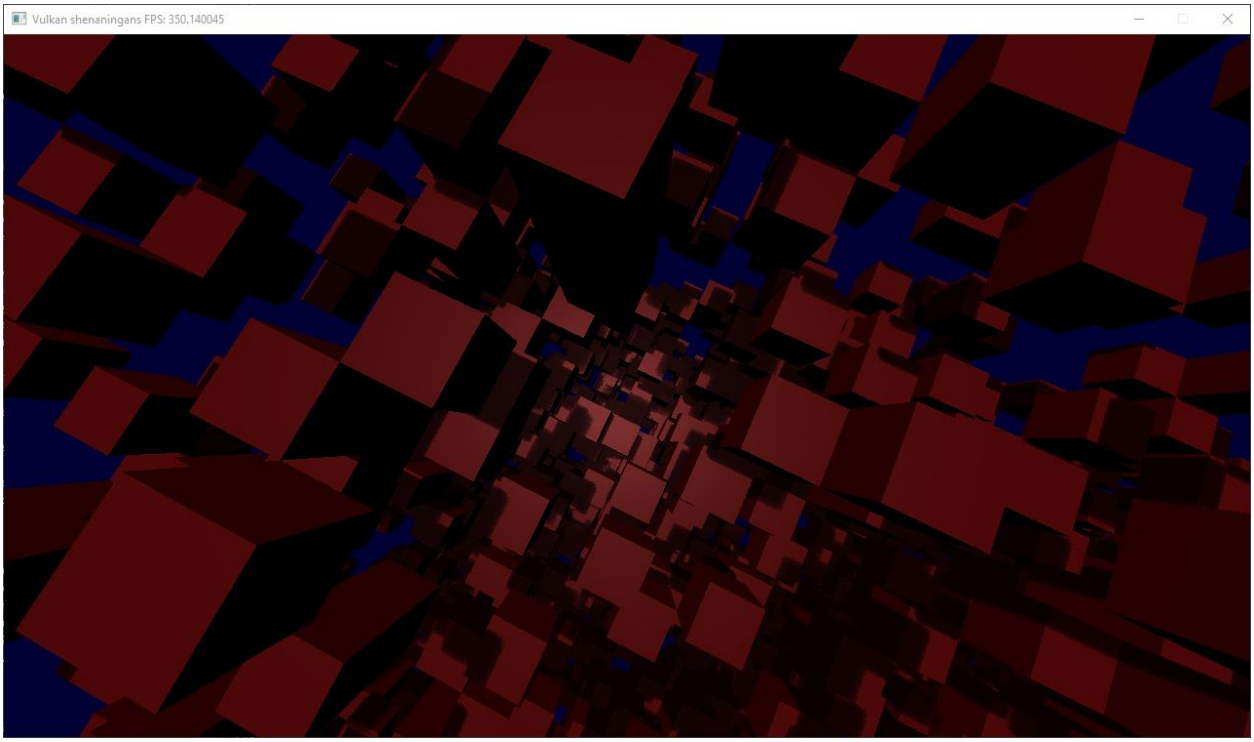
```
float GeometrySchlickGGX(float NdotV, float roughness)
{
    float r = (roughness + 1.0);
    float k = (r*r) / 8.0;

    float nom   = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}

float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx2 = GeometrySchlickGGX(NdotV, roughness);
    float ggx1 = GeometrySchlickGGX(NdotL, roughness);

    return ggx1 * ggx2;
}
```

Slika 4.4. Smithova metoda samosjenčanja

Osim PBR modela, postoji i rudimentaran sustav sjena gdje se iz svake točke šalje 20 uzoraka(zraka) prema izvoru svjetla te je konačno osvjetljenje ovisi o postotku zraka koje nesmetano dolaze do izvora svjetla.



Slika 4.5. Prikaz iz programa



Slika 4.6. Prikaz iz programa

4.3 Upute za korištenje

Za pokretanje programa potreban je Nvidia driver 443.41 te grafička kartica koja ima sklopovski omogućeno praćenje zrake. Nakon pokretanja programa tipke W, A, S i D se koriste za kretanje naprijed, lijevo, unazad i desno. Space i ctrl mogu se koristiti za kretanje gore i dolje.

Tipke Y i H koriste se za kontrolu metalnosti materijala, dok se tipke U i J koriste za kontrolu hrapavosti.

4.4 Literatura

[1] Interaktivan prikaz vokseliziranog prostora s Vulkanom uz sklopovski ubrzano praćenje zrake, 2020., https://www.fer.unizg.hr/_download/zavrsni_radovi/work/Final_0108069417_41.pdf

[2] Physically Based Rendering:From Theory To Implementation, Matt Pharr, Wenzel Jakob, and Greg Humphreys, 2018., https://www.pbr-book.org/3ed-2018/Reflection_Models/Microfacet_Models

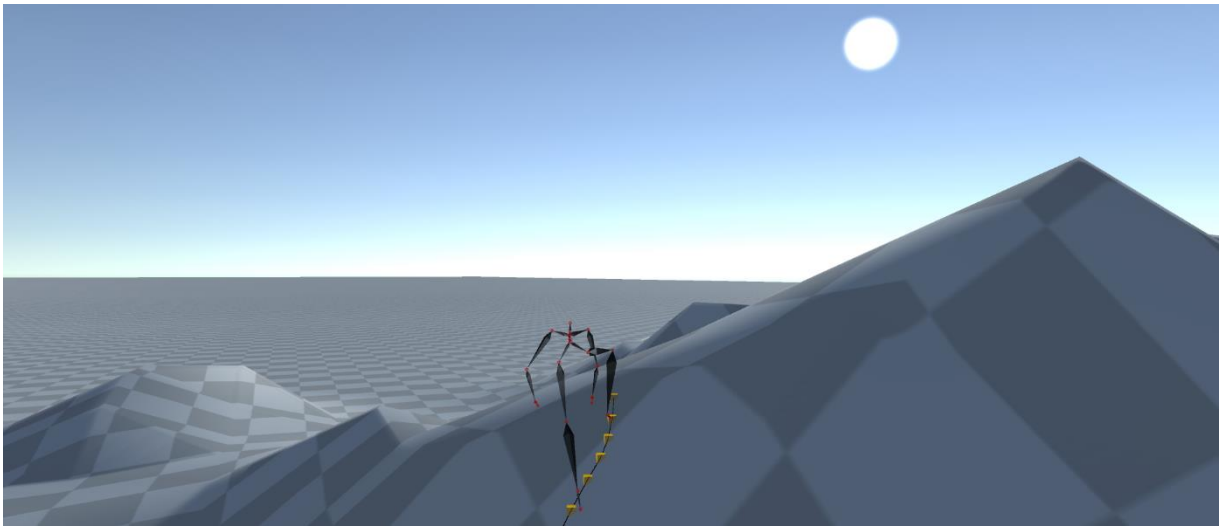
[3] Fresnel Equations, Schlick Approximation, Metals, and Dielectrics, 2020., http://psgraphics.blogspot.com/2020/03/fresnel-equations-schlick-approximation.html

5. Primjena metoda dubokog učenja u računalnoj animaciji

5.1 Opis razvijenog proizvoda

U sklopu ovog projekta ispitana je mogućnost korištenja neuronske mreže s faznom funkcijom (engl. *phase-functioned neural network*) za kontroliranje virtualnog lika (čovjeka). Ovaj pristup je prvi put predstavljen u radu [1]. Autori su kao dodatak radu priložili podatke za treniranje dobivene hvatanjem pokreta (engl. *motion capture*), izvorni kod za treniranje mreže te izvorni kod demonstracijske aplikacije napisane u programskom jeziku C++. Demonstracijska aplikacija zahtijeva Xbox kontroler da bi ispravno radila.

U sklopu ovog projekta bit će demonstrirano kako koristiti izvorni kod za treniranje mreže za dobivanje težina mreže. Također će biti demonstrirano kako iskoristiti te težine za kontroliranje virtualnog lika. U tu svrhu napravljena je demonstracijska aplikacija u razvojnom okruženju Unity koja ne zahtijeva Xbox kontroler, već radi i s tipkovnicom. Isječak iz demonstracijske aplikacije je prikazan na slici Slika 5.1 Demonstracijska aplikacija.



Slika 5.1 Demonstracijska aplikacija

5.2 Tehničke značajke

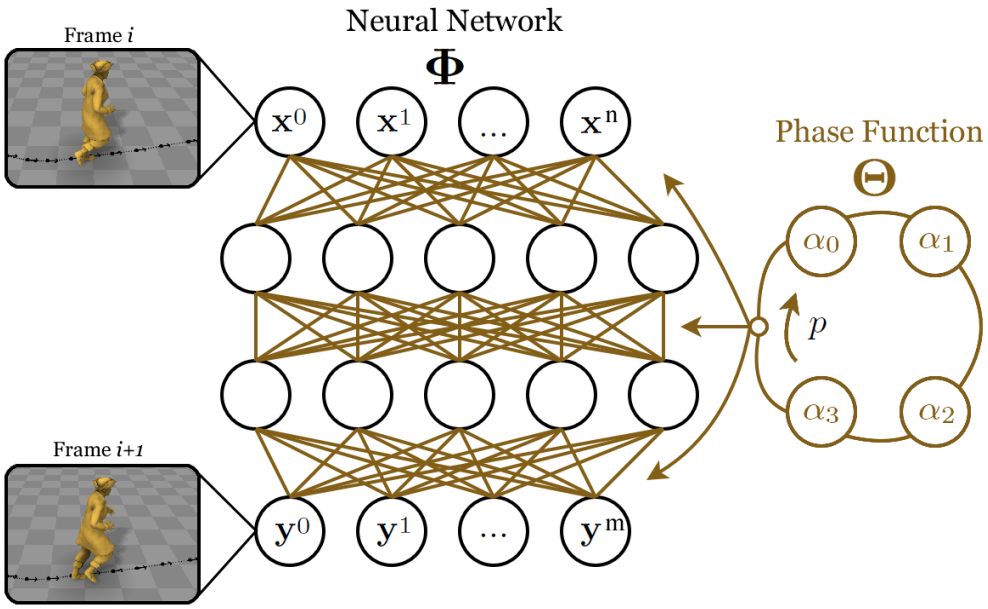
5.2.1 Opis mreže

Neuronska mreža s faznom funkcijom Φ je obična unaprijedna neuronska mreža čije težine u svakom trenutku ovise o vrijednosti fazne funkcije Θ. Fazna funkcija kao argument prima fazu, a kao izlaz daje težine neuronske mreže. Fazna funkcija je definirana kao periodična funkcija s periodom $2 \times \pi$. Ulaz u faznu funkciju, tj. faza označava u kojem trenutku u ciklusu pokreta se trenutno nalazimo.

Motivacija za dodavanje faze kao dodatne varijable je ta da je potrebno nekako jednoznačno definirati problem. Ako ne koristimo fazu, kod npr. hodanja, do iste krajnje pozicije je moguće

dospjeti kretanjem s lijevom nogom te kretanjem s desnom nogom. Neuronska mreža koja ne koristi fazu nauči nekakvu mješavinu kretanja s lijevom nogom i kretanja s desnom nogom što rezultira time da lik „lebdi“. Fazu nije dovoljno koristiti kao još jednu ulaznu varijablu jer se pokazalo da je tada njen utjecaj na izlaz mreže jako mal te nisu dobiveni zadovoljivi rezultati. Korištenjem faze za generiranje težina neuronske mreže ima puno veći utjecaj te se pokazalo da takva mreža postiže jako dobre rezultate.

Primjer neuronske mreže s faznom funkcijom koja je korištena u radu je prikazan na slici Slika 5.2. Neuronska mreža s faznom funkcijom.



Slika 5.2. Neuronska mreža s faznom funkcijom

Ulazne varijable (***x***) čine trenutna poza lika te korisnikove kontrole, dok izlazne varijable (***y***) sadržavaju promjenu u fazi te sljedeću pozu lika. Mreža ima 2 skrivena sloja dimenzije 512, ulazni sloj je dimenzije 342, a izlazni sloj je dimenzije 311. Kao aktivacijska funkcija je korištena eksponencijalna zglobnica (engl. *exponential rectified linear function - ELU*). Kao fazna funkcija korištena je kubna Catmull-Rom krivulja. Početna i krajnja kontrolna točka se poklapaju kako bi se osigurala periodičnost. Korištenje ovakve krivulje efektivno znači da svaka kontrolna točka (na slici $\alpha_0,\alpha_1,\alpha_2,\alpha_3$) predstavlja jednu konfiguraciju neuronske mreže, a fazna funkciju obavlja glatku interpolaciju između susjednih kontrolnih točaka. U radu je korištena krivulja sa svega 4 kontrolne točke što se pokazalo dovoljno.

5.2.2 Podaci za treniranje

Podaci za treniranje dobiveni hvatanjem pokreta su dostupni na [2]. Podaci sadrže razne pokrete - šetanje, trčanje, skakanje, čučanje itd. Također, podatci sadrže i pokrete prelaženja preko raznih prepreka. Podaci su snimani u 60 sličica u sekundi (engl. *frames per second*). Ukupno je snimljeno oko 1 sat isječaka što je rezultiralo podacima veličine oko 1.5 GB. Kao osnova za snimanje je korišten model ljudskog tijela koji se sastoji od 30 rotacijskih zglobova.

5.2.3 Pretprocesiranje

Prije treniranja mreže potrebno je obraditi podatke kako bi ih doveli u format prikladan za

treniranje. Pretprocesiranje svakog snimljenog isječka se odvija u nekoliko koraka [1]:

- **Označavanje faze**. Za ovo se koristi poluautomatski postupak - sličice kada je desna noga na podu su označene s 0, kada je lijeva noga na podu s π, a kada je ponovno desna noga na podu s $2 \times \pi$. Vrijednosti faze za sličice između toga se dobiju interpolacijom.
- **Označavanje tipa pokreta (npr. trčanje, šetanje itd.)**. Ovo se radi ručno.
- **Odabiranje geometrije okoline**. Pošto istovremeno snimanje pokreta i geometrije okoline nije jednostavno, potrebno je odabrati (virtualnu) okolinu koja najbolje odgovara snimljenom isječku. Ovo se radi automatski pomoću baze podataka visinskih mapa preuzetih iz nekoliko videoigara.

Pretprocesiranje se izvršava sljedećim naredbama:

```
1. python generate_patches.py

2. python generate_database.py
```

Rezultat pretprocesiranja je datoteka database.npz koja služi kao ulaz za proces treniranja. Proces pretprocesiranja je dosta zahtjevan te je trajao oko 3 sata na prijenosnom računalu s Intel i7-7700HQ procesorom.

5.2.4 Treniranje

Tijekom treniranja mreže optimiziramo parametre fazne funkcije (kontrolne točke), a time i težine neuronske mreže. Cilj je što bolje reproducirati skup podataka za treniranje (iz ulaznih podataka generirati izlazne podatke). U radu se kao funkcija pogreške koristi srednja kvadratna pogreška. Također se koristi L2 regularizacija. Kao optimizacijski algoritam se koristi Adam – varijanta stohastičkog gradijentnog spusta. Tijekom treniranja koristi se tehnika isključivanja čvorova grafa (engl. *dropout*).
Učenje se izvršava sljedećom naredbom:

```
3. python train_pfnn.py
```

Proces učenja proveden u 20 epoha je trajao oko 20 sati na prijenosnom računala s Nvidia GTX 1050 Ti grafičkom karticom. Solidne rezultate je moguće dobiti i nakon samo jedne epohe što je trajalo oko sat vremena na istom računalu.

5.2.5 Evaluacija

Mreža se koristi tako da se učitaju težine (koje zauzimaju oko 10MB što je puno manje od veličine podataka za treniranje koji su veliki oko 1.5GB) te se nakon toga pri svakoj sličici na ulaz mreže dovedu tražene varijable. Mreža će nakon toga izračunati izlazne varijable koje koristimo da bi ažurirali pozu lika i vrijednost faze.

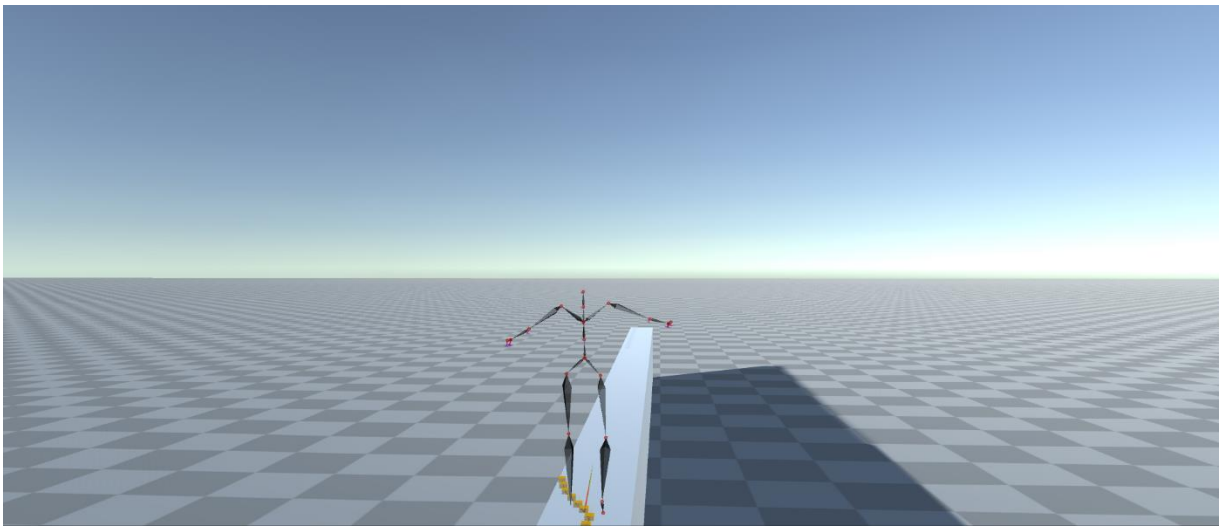
5.3 Upute za korištenje

Pokretanjem demonstracijske aplikacije korisnik odmah dobiva kontrolu nad virtualnim likom koji se nalazi u demonstracijskoj sceni. Kontrole su uobičajene:

- W – kretanje unaprijed
- A – kretanje ulijevo

- S – kretanje unazad
- D – kretanje udesno
- Q – rotacija ulijevo
- E – rotacija udesno
- *Ctrl* – prelazak u čučanj (engl. *crouch*)
- *Shift* - trčanje
- *Space* – skakanje

Demonstracijska scena sadrži nekoliko brda i jedan relativno uzak zid na koji se korisnik može popeti. Šetanjem po tom zidu vidimo kako je lik naučio balansirati, tj. održavati ravnotežu. Taj scenarij je prikazan na slici Slika 5.3. Primjer balansiranja.



Slika 5.3. Primjer balansiranja

5.4 Literatura

[1] Daniel Holden, Taku Komura, and Jun Saito. 2017. Phase-functioned neural networks for character control. ACM Trans. Graph. 36, 4, Article 42 (August 2017), 13 pages. DOI: https://doi.org/10.1145/3072959.3073663

[2] https://theorangeduck.com/page/phase-functioned-neural-networks-character-control, pristupano 21.01.2022.

6. Zamjena lica korištenjem sustava DeepFaceLab

6.1 Opis razvijenog proizvoda

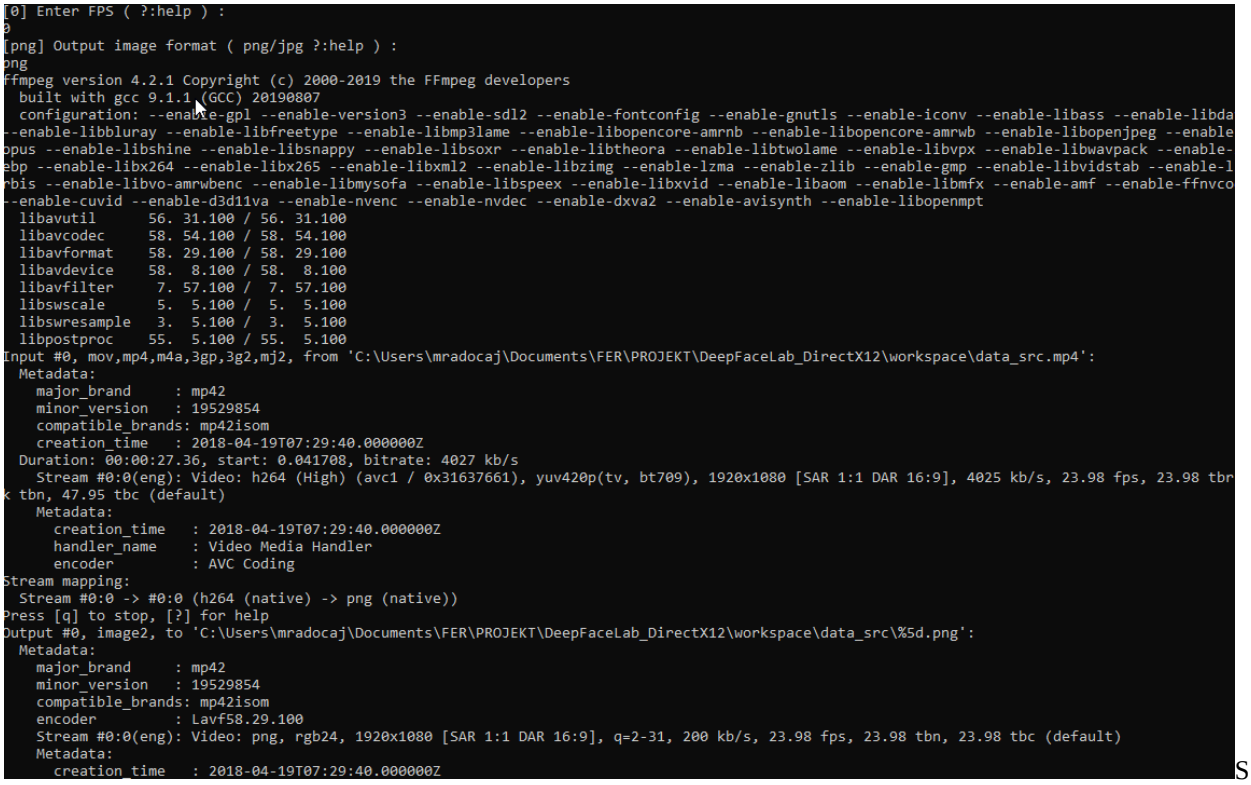
DeepFaceLab je sustav za zamjenu lica otvorenog koda koji nudi potpuni cjevovod za generiranje realističnih digitalnih zamjena lica. U sklopu ovog projekta ispitane su mogućnosti i ograničenja DeepFaceLab-a na konkretnom primjeru izvornog i odredišnog videozapisa. Analiza cjevovoda i njegovih komponenti dana je u radu autora sustava [1] te je opisana u seminarskom radu Sustav za zamjenu lica DeepFaceLab [2]. U nastavku slijedi praktični opis koraka za ekstrakciju podataka i učenje modela za zamjenu lica. Videozapisi korišteni u ovoj demonstraciji data_src.mp4 i data_dst.mp4 početni su primjeri dobiveni prilikom instalacije DeepFaceLaba. Nalaze se u /workspace direktoriju.

Slika 6.1. data_dst.mp4

Slika 6.2. data_src.mp4

6.1.1 Ekstrakcija podataka

U direktoriju s instaliranim sustavom nalaze se brojne skripte. Potrebno je u početku obaviti ekstrakciju lica iz odredišnog i izvornog videozapisa. Ekstrakcija iz izvornog videozapisa obavlja se pokretanjem skripte "2) extract images from video data_src.bat":



Slika 6.3. Izvođenje skripte za ekstrakciju slika iz videozapisa

Rezultat izvođenja skripte je ukupno 655 slika u .png formatu generiranih iz izvorišnog videozapisa i spremljenih u /workspace/data_src. Ove slike ne sadrže samo lice već čitavu scenu prikazanu u izvorišnom videozapisu. Isti postupak potrebno je ponoviti i za odredišni videozapis pomoću skripte "3) extract images from video data_dst FULL FPS.bat". Te slike pohranjene su u /workspace/data_dst direktoriju. Svaka od ovih skripti nije previše vremenski zahtjevna (izvođenje traje do minutu).

Sljedeći korak je iz ekstrahiranih slika pronaći lice u svakoj te generirati nove slike koje će sadržavati samo poravnato lice. Ovaj kompleksan proces koji uključuje detekciju lica korištenjem S3FD (*Single Shot Scale-invariant Face Detector*) detektora lica i poravnanje lica pomoću algoritama 2DFAN i PRNet obuhvaćen je u po jednoj skripti sa svaki videozapis: "4) data_src faceset extract.bat" i "5) data_dst faceset extract.bat". Ovo je vremenski i računski zahtjevniji zadatak te korisnik može odabrati hoće li se za njega koristiti CPU ili GPU (ili više njih).

```
[y] Continue extraction? ( y/n ?:help ) : y

Choose one or several GPU idxs (separated by comma).

[CPU] : CPU
  [0] : NVIDIA Quadro M1000M
  [1] : Intel(R) HD Graphics 530

[0,1] Which GPU indexes to choose? : 0,1
0,1

[wf] Face type ( f/wf/head ?:help ) :
wf
[0] Max number of faces from image ( ?:help ) :
0
[512] Image size ( 256-2048 ?:help ) :
512
[90] Jpeg quality ( 1-100 ?:help ) :
90
[n] Write debug images to aligned_debug? ( y/n ) :
n
Extracting faces...
Running on Intel(R) HD Graphics 530
Running on NVIDIA Quadro M1000M
 21%|#######################6
```

Slika 6.4. Pokretanje skripte za ekstrakciju lica

Rezultat izvođenja skripti su slike s poravnatim licima u /workspace/data_src/aligned i /workspace/data_dst/aligned direktorijima.

Slika 6.5. Slika iz videozapisa

Slika 6.6. Slika kao rezultat ekstrakcije lica

6.1.2 Treniranje modela

Uz poravnata lica izvorišnog i odredišnog videozapisa moguće je početi trenirati model. U okviru projekta isprobana su dva različita modela: Quick96 prilagođen slabijim grafičkim karticama ili treniranju sa CPU te SAEHD za treniranje s naprednijim grafičkim karticama s minimalno 6G B VRAM-a.
Za treniranje je korišten prilagođeni DFL sustav za Google Colab [3]. Već spreman kod u Pythonu nudi podršku za treniranje modela tako da se željeni workspace prvo učita na Google Disk. Potrebno je pokrenuti treniranje sa željenim modelom te namjestiti parametre. Svi modeli trenirani su na NVIDIA Tesla K80 grafičkoj kartici.

Prvo isprobani je jednostavniji Quick96 model. Svaka iteracija trajala je oko 200 ms.

Slika 6.7. Ekran za interaktivni pregled treniranja

Na Slici 6.7 prikazani su rezultati treniranja Quick96 nakon 69 tisuća iteracija (desno) u usporedbi sa originalnim videozapisom (lijevo).

Slika 6.8. Originalno lice (lijevo) i lice generirano pomoću DeepFaceLab-a (desno)

Nadalje, treniran je i novi SAEHD model koji nudi brojne mogućnosti konfiguracije. SAEHD nudi rezultate veće rezolucije te se pokazuje uspješniji u generiranju novog lica prikazanog i iz većih kuteva. Slika 5.8 prikazuje usporedbu Quick96 i SAEHD modela: dok kod Quick96 (lijevo) nastaju zamućenja u ovom položaju lica (posebice u području očiju), SAEHD (desno) daje mmnogo realističnije rezultate. Pri tome su oba modela trenirana na isti broj iteracija (69 tisuća).

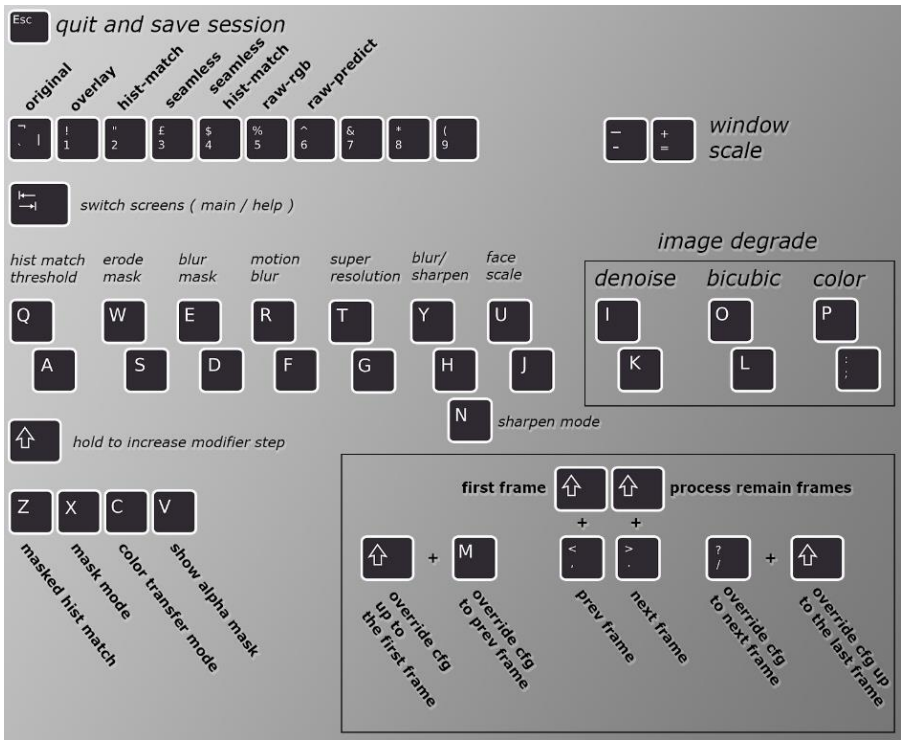
Slika 6.9. Usporedba Qucik96 (lijevo) i SAEHD (desno)

Isprobano je i treniranje unaprijed treniranog modela. Takav model treniran je tisućama iteracija na različitim licima te se koristi kako bi se proces treniranja ubrzao i dao bolje rezultate. Korišten je unaprijed treniran model arhitekture LIAE rezolucije 192 piksela. Trenirana su dva modela: jedan sa korištenjem unaprijed treniranog modela te drugi bez početnog modela. Oba modela trenirana su kroz 10 tisuća iteracija. Iako dobiveni rezultati ne odgovaraju u potpunosti očekivanjima (pogotovo u području usta i donjeg dijela lica), vidi se bolja rezolucija u području očiju.

Slika 6.10. Rezultat bez (lijevo) i s unaprijed treniranim modelom (desno)

6.1.3 Konverzija

Proces konverzije (primjene modela za generiranje manipuliranih videozapisa) također se svodi na poziv skripte. Konverzija ovisi o načinu treniranja modela pa tako postoje skripte "7) merge SAEHD.bat" i "7) merge Quick96.bat". Interaktivni način rada omogućava razne manipulacije lica: njegovo smanjenje na odredišnom videozapisu, zamućivanje granica, itd. Na Slici 6.11 prikazano je sučelje za interaktivni *merge* te sve opcije koje on nudi.



Slika 6.10. Sučelje za interaktivnu konverziju

6.2 Tehničke značajke

Ako se radi lokalno, potrebno je preuzeti DeepFaceLab. Instalaciju je moguće pronaći na službenom GitHub-u projekta [4]. Također, preporuča se korištenje grafičke kartice kako bi se dugotrajni proces treniranja što brže izveo.

Za treniranje modela na DFL Google Colab-u potreban je jedino Google račun. Treniranje se može odvijati po pravilima Google Colaba (za besplatnu verziju maksimalno trajanje je 12 sati).

6.3 Literatura

[1] Ivan Perov, Daiheng Gao, Nikolay Chervoniy, Kunlin Liu, Sugasa Marangonda, Chris Umé, Mr. Dpfks, Carl Shift Facenheim, Luis RP, Jian Jiang, Sheng Zhang, Pingyu Wu, Bo Zhou, Weiming Zhang. DeepFaceLab: A simple, flexible and extensible face swapping framework, (2020, svibanj)

[2] Maja Radočaj. Sustav za zamjernu lica DeepFaceLab, (2021, svibanj)

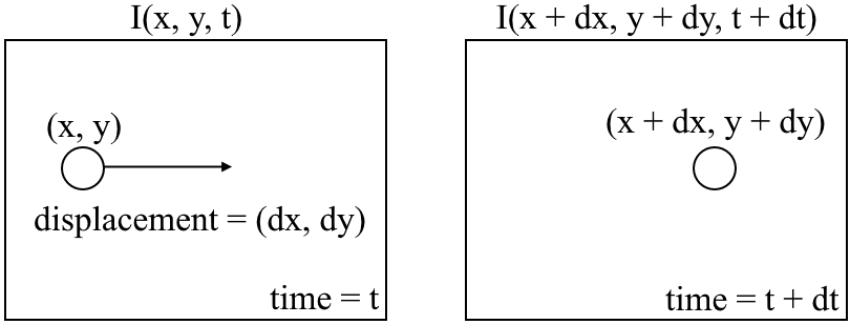
[3] DFL-Colab. Poveznica: https://colab.research.google.com/github/chervonij/DFL-Colab/blob/master/DFL_Colab.ipynb

[4] Ivan Perov. DeepFaceLab. Poveznica: https://github.com/iperov/DeepFaceLab

7. Usporedba algoritama za izračun optičkog toka

7.1 Opis razvijenog proizvoda

U ovom projektu analiziraju se odabrani algoritmi za izračun optičkog toka (engl. *optical flow*) nad videozapisom. Koriste se implementacije algoritama iz biblioteke *OpenCV*.
Optičkim tokom iz niza slika analiziraju se pokreti u sceni te se određuje njihov smjer i relativna brzina. Za tu potrebu razvijeni su brojni algoritmi koji rade na cijeloj slici ili na njenim dijelovima, zbog čega razlikujemo gusti i rijetki optički tok. Kao izvor podataka za analizu koristit će se scene kreirane u alatu Blender.
Računanje optičkog toka svodi se na izračun, odnosno procjenu, pomaka svakog piksela između dvije ulazne slike videozapisa. Drugim riječima, potrebno je odrediti vektora pomaka svake točke na slici, bilo zbog pomaka objekta ili pomaka kamere.
Pretpostavimo da ne radimo s RGB slikama, već da se koriste samo nijanse sive boje (engl. *grayscale*), odnosno da imamo dvodimenzionalnu matricu intenziteta.
Definirajmo funkciju $I(x,y,t)$ koja predstavlja intenzitet pojedinog piksela. Parametri x i y predstavljaju koordinate piksela, a t je redni broj slike u videozapisu. Parametar t jest ekvivalent proteklog vremena jer pretpostavljamo videozapis s konstantnim brojem sličica u sekundi (engl. *frames per second*, FPS).
Kao početnu pretpostavku uzimamo da pomak objekta u videozapisu ne mijenja intenzitet pripadnog piksela. Matematički, to predstavlja jednakost $I(x,y,t) = I(x+dx, y+dy, t+dt)$. S obzirom na to da računamo samo sa slikama koje su u videozapisu susjedne, postavljamo $dt = 1$. Konačan problem odrediti je vektor pomaka (dx, dy), što je vizualizirano na slici niže.



Slika 7.2. Vizualizacija pomaka objekta između dvije slike. Izvor: nanonets.com

Koristeći Taylorov red uz zanemarivanje viših članova dobivamo:

$$I(x+dx, y+dy, t+dt) = I(x,y,t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt + \cdots$$

$$\Rightarrow \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt = 0$$

Dijeljenjem s dt, te uz pojednostavljenje $u = \frac{dx}{dt};\ v = \frac{dy}{dt}$, dobivamo konačnu formulu:

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v = -\frac{\partial I}{\partial t}$$

Razlomci $\frac{\partial I}{\partial x}; \frac{\partial I}{\partial y}; \frac{\partial I}{\partial t}$ predstavljaju gradijente po horizontalnoj i vertikalnoj osi, i u vremenu. Traženje vektora pomaka može se poistovijetiti s računanjem u i v. Međutim, javlja se problem da je iz samo jedne jednadžbe potrebno odrediti dvije nepoznate vrijednosti. Tu zapreku na različite načine rješavaju algoritmi koji se obrađuju u nastavku.
Rijetki optički tok (engl. *sparse optical flow*) računa vektore pomaka samo za neke specijalne točke koje se moraju unaprijed odrediti, primjerice rubove i vrhove objekata. S druge strane, gusti optički tok (engl. *dense optical flow*) provodi izračun vektora za svaki piksel slike. Očekivano, u pravilu se gustim optičkim tokom dobivaju točniji rezultati, uz cijenu većeg utroška računalne snage, odnosno sporijeg izračuna.

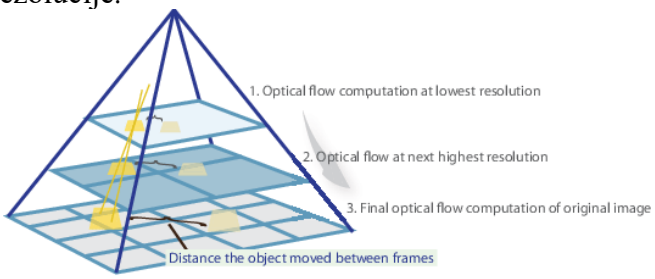
7.1.1 Lucas-Kanade algoritam

Algoritam Lucas-Kanade često se koristi za računanje rijetkog optičkog toka [4]. Baziran je na pretpostavci da su vektori pomaka jednaki za točke koje su prema poziciji međusobno bliske. Konkretno, koriste se prozorčići fiksne veličine te se za sve piksele u njima pretpostavlja da imaju jednak vektor pomaka (u, v), odnosno (dx, dy). Iz toga se izvodi linearni sustav s više jednadžbi nego nepoznanica, što se kasnije rješavanja metodom najmanjih kvadrata, odnosno pseudoinverzom:

$$A = \begin{bmatrix} \frac{\partial I}{\partial x} q_1 & \frac{\partial I}{\partial y} q_1 \\ \vdots & \vdots \\ \frac{\partial I}{\partial x} q_n & \frac{\partial I}{\partial y} q_n \end{bmatrix} \qquad \gamma = \begin{bmatrix} u \\ v \end{bmatrix} \qquad b = \begin{bmatrix} -\frac{\partial I}{\partial t} q_1 \\ \vdots \\ -\frac{\partial I}{\partial t} q_n \end{bmatrix}$$

$$A\gamma = b \;\Rightarrow\; A^T A\gamma = A^T b \;\Rightarrow\; (A^T A)^{-1}(A^T A)\gamma = (A^T A)^{-1} A^T b \;\Rightarrow\; \gamma = (A^T A)^{-1} A^T b$$

U prethodnim jednadžbama varijable q_i predstavljaju n piksela u jednom prozoru.
Kako bi se poboljšale performanse algoritma kod naglih pokreta, često se dodatno koristi piramidalna metoda [5]. U suštini, ista slika se skalira na različite rezolucije, primjerice s faktorom dva. S obzirom na to da se koristi prozor fiksne veličine, na manjoj rezoluciji on će prekrivati veći dio slike te ima veću vjerojatnost da uspije detektirati nagli pokret. Rezultantni vektori koriste se kao početne informacije u sljedećoj iteraciji s većom rezolucijom, i tako dok se ne dođe do originalne rezolucije.



Slika 7.3, Vizualizacija piramidalne metode. Izvor: mathworks.com

Kako je prethodno spomenuto, za rijetki optički tok potrebno je odabrati specijalne točke. U implementaciji rješenja za to se koristi Shi-Tomasi algoritam za detekciju vrhova, koji neće biti detaljnije opisivan.

7.1.2 Gusti Lucas-Kanade algoritam

Kako bi se zadržale dobre strane algoritma za izračun rijetkog optičkog toka, a kada je potrebna informacija o pomacima točaka na cijeloj slici, moguće je interpolirati rezultate Lucas-Kanade algoritma na dijelovima gdje nemamo informacije, čime se efektivno dobiva gusti Lucas-Kanade algoritam.
Takav postupak očekivano je sporiji od običnog Lucas-Kanade algoritma, ali je ipak brži od većine ostalih algoritama specijaliziranih za izračun gustog optičkog toka. Naravno, ovisno o karakteristikama videozapisa, rezultati dobiveni interpolacijom mogu biti relativno neprecizni.

7.1.3 Farnebackov algoritam

U inicijalnom izvodu jednadžbe za optički tok prilikom korištenja Taylovorovog reda zanemarili smo sve članove nakon linearnog. Farnebackov algoritam na tom mjestu radi izmjenu te dodaje i kvadratni član [6], što je u radu opisano sljedećom jednadžbom: $I(x) \sim x^T Ax + b^T x + c$. Time je povećana preciznost aproksimacija. Glavna ideja jest da se pokušava izračunati pomak d takav da vrijedi $I_2(x) = I_1(x - d)$.

7.1.4 Robusni lokalni optički tok

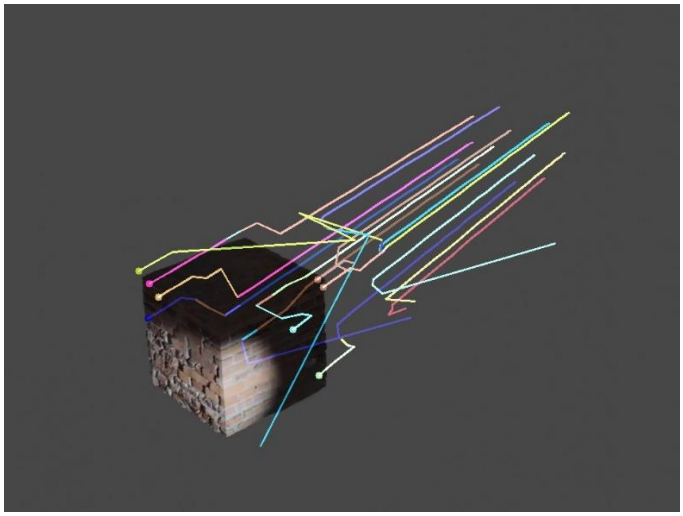
Robusni lokalni optički tok (engl. *Robust Local Optical Flow*, RLOF) noviji je algoritam objavljen 2012. godine, uz nekoliko nadogradnji u naredne četiri godine [7]. RLOF se odmiče od pretpostavke da je intenzitet piksela prije i poslije pomaka jednak jer ona nije realistična. Primjerice, može biti narušena zbog utjecaja refleksija, sjena ili pomicanja izvora svjetlosti, odnosno raznih promjena u osvjetljenju.
Baziran je na modelu osvjetljenja kojeg su originalno predložili Gennert i Negahdaripour [8]:
$I(x, y, t) + m * I(x, y, t) + c = I(x + u, y + v, t + 1)$
Varijable m i c su parametri modela osvjetljenja. Slično prethodnim algoritmima, pretpostavlja se lokalna konzistentnost, odnosno da su u pojedinom malom području vrijednosti parametara m i c jednake.
Algoritam definira minimizacijsku funkciju vezanu uz model osvjetljenja te iterira do konvergencije. Dok svi ostali opisani algoritmi rade samo sa sivim vrijednostima, RLOF zahtijeva RGB slike.

7.2 Tehničke značajke

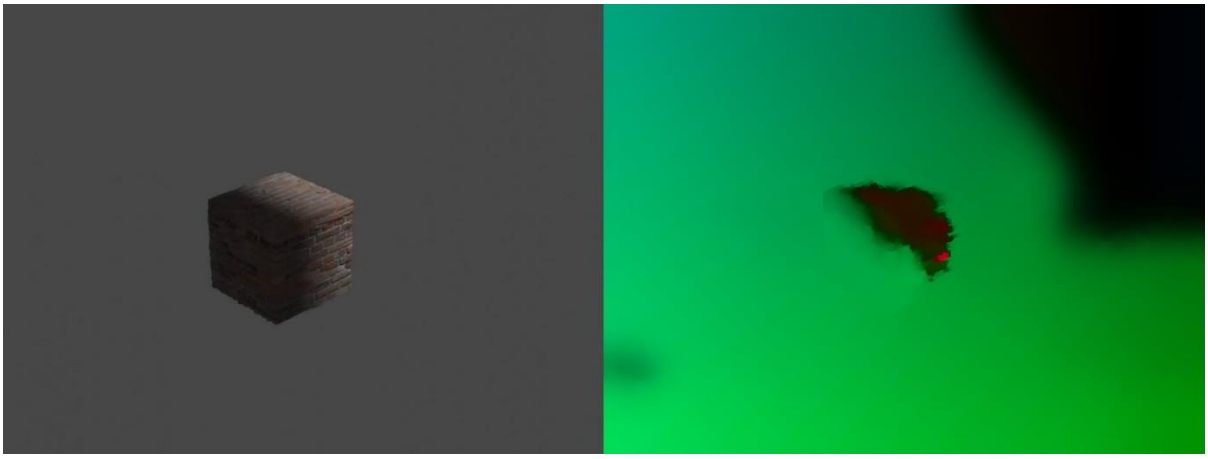
Rijetki optički tok vizualiziran je kao skup raznobojnih krivulja koje prate poziciju svake točke odabrane Shi-Tomasi algoritmom.
Gusti optički tok kao rezultat daje dvodimenzionalno polje dvokomponentnih vektora. Vrlo čest i vizualno atraktivan prikaz takvog polja jest u HSV formatu. Veći pomaci, tj. dulji vektori, imaju veću zasićenost boje (engl. *saturation*), dok se različiti smjerovi prikazuju različitim nijansama (engl. *hue*). Prije konačnog prikaza HSV se naravno pretvara u RGB vrijednosti.
Videozapis nad kojim se uspoređuju algoritmi ima 72 sličice (3 sekunde i 24FPS). Rezolucija je 960 x 720 piksela. Ta jednostavna animacija napravljena je u alatu Blender kako bismo imali punu kontrolu nad konačnim videozapisom. Kocka s uzorkom cigli linearno se pomiče iz gornjeg

desnog u donji lijevi kut ekrana, nakon čega se isto tako vraća na početnu poziciju. Kao primarno osvjetljenje je korišten efekt Sunca. Radi testiranja ponašanja algoritma pod promijenjivom rasvjetom, u donjem lijevom kutu kocka dolazi u sjenu. Dodatno, upaljen je i jak reflektor na dvije strane kocke dok je ona u sjeni. Pozadina je uniformno siva. Na kocki je korišten i modifikator pomaka (engl. *displacement modifier*) kako bi pojedine cigle bile izdignute.
U nastavku se nalaze primjeri rezultata nad opisanim videozapisom. Vidimo da se u sjeni svi algoritmi ponašaju nepoželjno. Gusti Lucas-Kanade i RLOF pokazuju značajne artefakte u okolini kocke. Moguće je da parametri algoritma nisu optimalni, ali ni nakon pokušaja izmjene nisu dobiveni značajno bolji rezultati.
Vremena izvođenja su sljedeća:

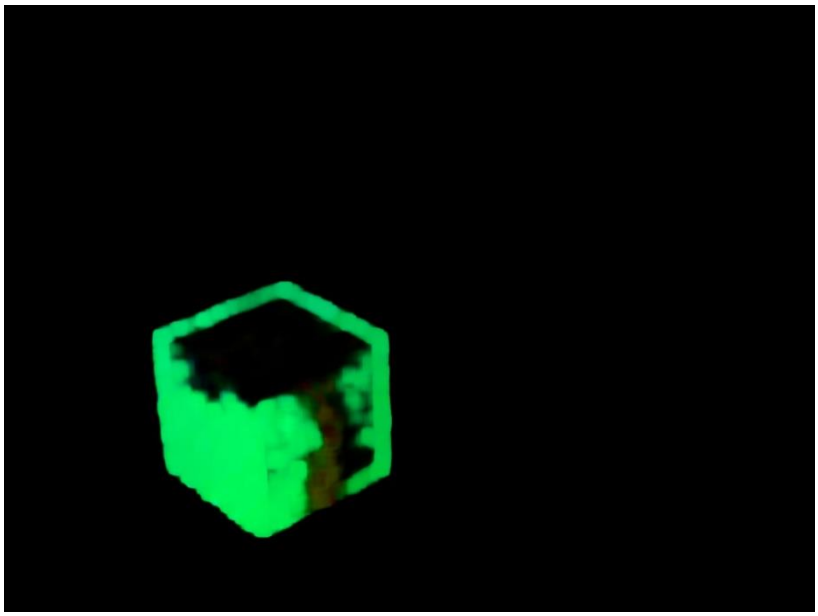
- Lucas-Kanade algoritam: 1,8 sekundi,
- Gusti Lucas-Kanade algoritam: 5,6 sekundi,
- Farnebackov algoritam: 10,1 sekundi,
- Robusni lokalni optički tok: 18,5 sekundi.



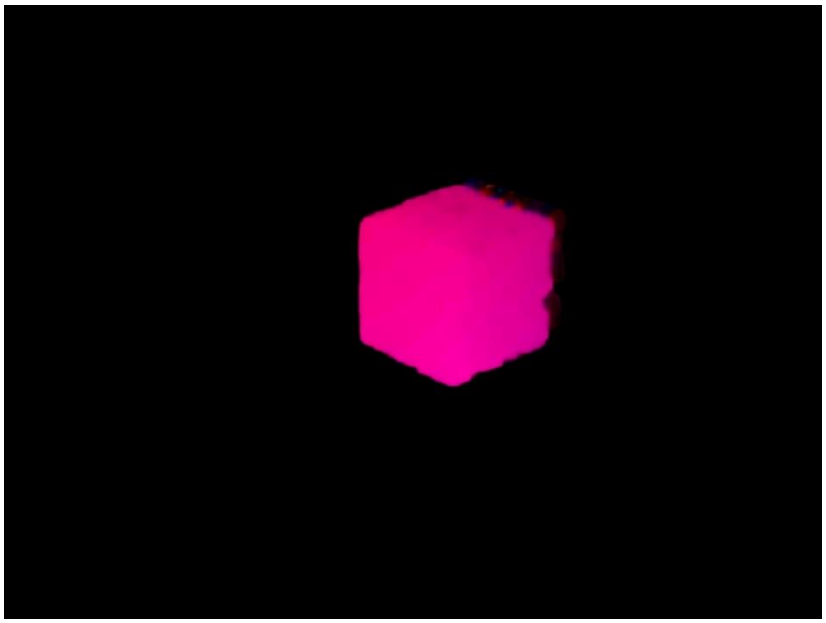
Slika 7.4. Lucas-Kanade algoritam



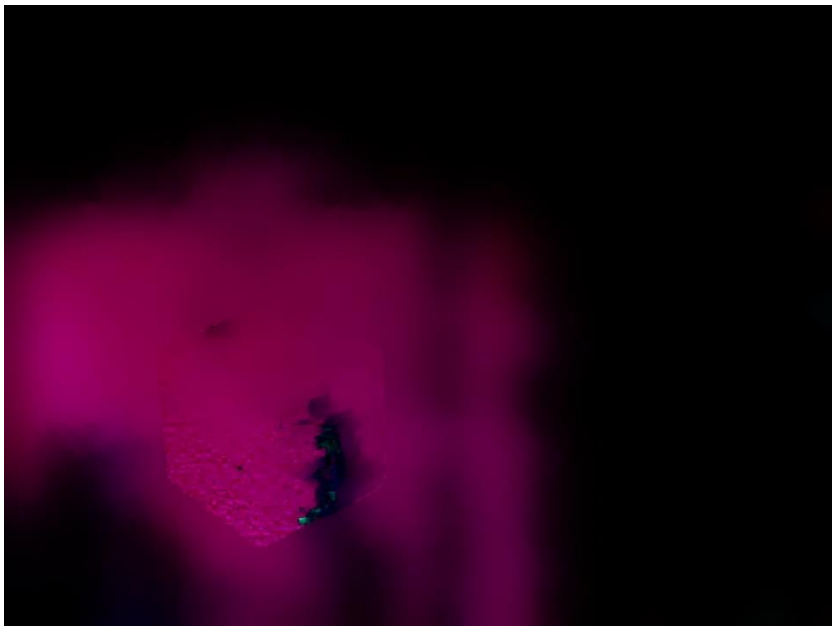
Slika 7.5. Gusti Lucas-Kanade algoritam



Slika 7.6. Farnebackov algoritam, trenutak ulaska u reflektor



Slika 7.7. Farnebackov algoritam, trenutak povratka prema početnoj poziciji



Slika 7.8. Robusni lokalni optički tok

Vidimo da se vrijeme izvođenja povećava na svakom sljedećem algoritmu. Jedino Lucas-Kanade algoritam radi u realnom vremenu, dok ostalima treba i do šest puta više vremena nego što traje sam videozapis. Naravno, moguće je smanjiti FPS ulaznog videozapisa i time ukupan broj sličica, ali tada između dvije slike imamo veće pomake što automatski rezultira većim greškama. Također, mogli bismo smanjiti rezoluciju videozapisa, ali ići niže od korištene rezolucije vjerojatno nema pretjeranog smisla iz perspektive korisnosti konačnog rješenja.
Za računanja korišten je CPU. Naravno, paralelizacija značajno ubrzava izvođenje takvih algoritama, primjerice koristeći grafičke procesore. Postoje implementacije bazirane na CUDA tehnologiji, ali to nije istraženo u ovom projektu.
Kao i u mnogim drugim područjima, danas se razmatraju i vrlo uspješni algoritmi bazirani na dubokom učenju.

7.3 Upute za korištenje

Program se u terminalu pokreće naredbom sljedećeg formata:
`python main.py oznaka_algoritma putanja_do_videozapisa`
`oznaka_algoritma` može biti jedna od sljedećih vrijednosti:

- `lk` ili `0` (Lucas-Kanade algoritam, rijetki),
- `lk_dense` ili `1` (Lucas-Kanade algoritam, gusti),
- `farneback` ili `2` (Farnebackov algoritam, gusti),
- `rlof` ili `3` (Robusni lokalni optički tok, gusti).

Unos brojevnih oznaka omogućen je radi njihovog bržeg unosa prilikom pokretanja.
`putanja_do_videozapisa` jest relativna ili apsolutna putanja do videozapisa na disku nad kojim se želi pokrenuti odabrani algoritam.

Program će završiti kada izračuna optički tok za svaku sliku u videozapisu, ili kada se pritisne tipka `Escape`.
S obzirom na to da program uz Python 3 koristi i biblioteke, potrebno ih je imati instalirane, globalno ili npr. u *venv* okruženju. Konkretno, zahtjevi su sljedeći:

- numpy>=1.22.0
- opencv-python>=4.5.5
- opencv-contrib-python>=4.5.5

Izvjesno je da bi program funkcionirao i sa starijim verzijama navedenih biblioteka, ali to nije posebno testirano.

7.4 Literatura

[1] Igor Smolkovič, Analiza i vrednovanje odabranih izvedbenih detalja postupaka optičkog toka
[2] Patel, Ekta & Shukla, Dolley. (2013). Comparison of Optical Flow Algorithms for Speed Determination of Moving Objects. International Journal of Computer Applications. 63. 32-37. 10.5120/10465-5180.
[3] OpenCV, Optical Flow, https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html
[4] B. D. Lucas and T. Kanade (1981), An iterative image registration technique with an application to stereo vision, Proceedings of Imaging Understanding Workshop, str. 121—130
[5] J. Y. Bouguet, (2001). Pyramidal Implementation of the Affine Lucas Kanade Feature Tracker: Description of the algorithm. Intel Corporation, 5.
[6] G. Farneback. Two-Frame Motion Estimation Based on Polynomial Expansion
[7] T. Senst, V. Eiselein and T. Sikora, "Robust Local Optical Flow for Feature Tracking," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, no. 9, pp. 1377-1387, Sept. 2012, doi: 10.1109/TCSVT.2012.2202070.
[8] S. N. Gupta and J. L. Prince, "Stochastic formulations of optical flow algorithms under variable brightness conditions," Proceedings., International Conference on Image Processing, 1995, pp. 484-487 vol.3, doi: 10.1109/ICIP.1995.537677.
[9] OpenCV, Shi-Tomasi Corner Detector & Good Features to Track, https://docs.opencv.org/4.x/d4/d8c/tutorial_py_shi_tomasi.html