SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 133

PRIKAZ PROŠIRENIH KRIVULJA NA MODELU VISINSKE MAPE

Josip Komljenović

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 133

PRIKAZ PROŠIRENIH KRIVULJA NA MODELU VISINSKE MAPE

Josip Komljenović

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

ZAVRŠNI ZADATAK br. 133

Pristupnik: **Josip Komljenović (0036514351)**

Studij: Elektrotehnika i informacijska tehnologija i Računarstvo

Modul: Računarstvo

Mentor: prof. dr. sc. Željka Mihajlović

Zadatak: **Prikaz proširenih krivulja na modelu visinske mape**

Opis zadatka:

Proučiti grafički programski pogon Unreal Engine. Proučiti postupke izrade krivulje. Razraditi postupke izrade krivulja na visinskoj mapi uz proširenje širine krivulje. Razraditi mogućnosti implementacije na grafičkom procesoru uz korištenje sjenčara. Ostvariti prikaze rezultata na modelu visinske mape. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti grafičku programski pogon Unreal Engine. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 11. lipnja 2021.

Sadržaj

Uvod 1
1. Krivulje 2
  1.1. Općenito o krivuljama 2
  1.2. Bézierova krivulja 3
2. Korištene komponente 5
  2.1. Visinska mapa 5
  2.2. Materijali i mreže poligona 6
3. Implementacija 8
  3.1 Izrada krivulje 8
  3.2 Proširivanje krivulje 10
4. Korištenje programskog rješenja 13
5. Performanse implementacije 15
6. Rezultati i moguća poboljšanja 16
  6.1 Rezultati 16
  6.2 Moguća poboljšanja 17
7. Zaključak 18
Literatura 19

Uvod

Računalna grafika u današnjem svijetu ima široku primjenjivost. Jedan od ciljeva računalne grafike je otkriti način prikazivanja objekata na površini mape. Većina objekata koju nalazimo je u obliku krivulje. Prikazivanje krivulje nije jednostavno u usporedbi s ostalim oblicima, poput linije. Općenite krivulje pred nas postavljaju razne izazove. Kao prvo, potrebno je odrediti kako uopće zadati i zapisati krivulju. Također iz ulaznih podataka je potrebno generirati sve točke krivulje na vizualno i matematički prihvatljiv način. Dodatni problem predstavlja potreba da krivulja bude generirana na površini visinske mape. Na kraju je potrebno krivulju i iscrtati, što može predstavljati najveći problem za performanse samog programa. Cilj ovog rada je proći kroz ova pitanja i probleme i dati rješenje za njih.

U ovom radu je obrađen postupak izrade krivulje iz zadanih točaka u 2D prostoru, transformiranje točaka iz 2D prostora u 3D prostor na modelu visinske mape te proširenje te krivulje za potrebe prikazivanja na površini visinske mape. Za izradu ovog programskog rješenja korišten je grafički programski pogon *Unreal Engine* 4.26.1. u kombinaciji s programskim jezikom C++.

1. Krivulje

1.1. Općenito o krivuljama

U matematici krivulja predstavlja jedan širok pojam. Krivuljom možemo smatrati svaki skup točaka, najčešće definiranih određenom matematičkom funkcijom. Tu funkciju možemo zapisati na razne načine. Krivulju možemo zapisati eksplicitno u obliku $y = f(x)$, međutim to za posljedicu ima nemogućnost ostvarivanja jednog od poželjnih svojstava krivulje, a to je mogućnost da za jednu vrijednost x prikažemo više y vrijednosti. Drugi način je implicitno u obliku $f(x, y) = 0$, koji rješava problem višestrukog prikazivanja, međutim ne rješava problem djelomičnog prikazivanja (npr. prikazivanje samo jednog dijela krivulje). Zapis koji rješava oba prethodno navedena problema je parametarski zapis u obliku $x = f_1(t)$ i $y = f_2(t)$ iz čega vidimo da su vrijednosti x i y međusobno neovisne, te u slučaju da želimo iscrtati samo dio krivulje, potrebno je ograničiti promatrane vrijednosti parametra t. Zbog tih povoljnih svojstava parametarskog zapisa krivulje u ostatku ovog rada će se obrađivati samo taj zapis. Do prethodno navedenih jednadžbi dolazimo postupkom zadavanja krivulje. Krivulju možemo zadati na način da odredimo kroz koje točke krivulja prolazi ili da odredimo neku od derivacija (prvu ili višu derivaciju) u određenim točkama te krivulje. Za potrebe ovog rada odabrano je zadavanje krivulje pomoću točaka kroz koje krivulja mora prolaziti, i vrijednosti tangente (odnosno vrijednosti prve derivacije) u tim točkama. U kontekstu zadavanja točaka bitno je razlikovati dvije vrste krivulja: aproksimacijske i interpolacijske. Najjednostavnije gledano, za aproksimacijske krivulje je karakteristično da krivulja prolazi negdje u blizini točke, a za interpolacijske krivulje vrijedi da krivulja prolazi kroz zadane točke. Budući da je u željenoj primjeni ovog rada češća potreba za zadavanje konkretnih koordinata kroz koje je potrebno da prolazi krivulja odabrana je krivulja koja nastaje povezivanjem segmenata kubne aproksimacijske Bézierove krivulje. Jedno od bitnih svojstava korištene krivulje je da kada su za dva susjedna segmenta tangente na istom pravcu i prva derivacija jednaka na oba segmenta tada je zadovoljen C1 kontinuitet. Jednadžba krivulje se obično zapisuje korištenjem težinskih funkcija. Ukoliko sa $f_i(t)$ označimo i-tu težinsku funkciju tada jednadžbu krivulje možemo zapisati u obliku

$$T_K = \sum_{i=0}^{n} f_i(t) \cdot T_i$$

gdje je s T_K označena točka krivulje, n je stupanj krivulje (koji se računa kao broj točaka koji je zadan umanjen za 1), a T_i je i-ta zadana točka.

1.2. Bézierova krivulja

Nakon što su definirani određeni pojmovi vezani uz općenite krivulje, potrebno je definirati i krivulju koja je korištena u ovom radu, a to je Bézierova krivulja. Bézierove krivulje je otkrio francuski inženjer Pierre Bézier 60-ih godina 20. stoljeća. Matematički gledano, Bézierovu krivulju je najlakše zadati koristeći Bernsteinove težinske funkcije, koje dobivamo kao rezultat nakon provođenja De Casteljauovog algoritma. Bernsteinove težinske funkcije za Bézierove krivulje trećeg stupnja iznose redom:

$$b_{0,3}(t) = (1-t)^3$$

$$b_{1,3}(t) = 3(1-t)^2 t$$

$$b_{2,3}(t) = 3(1-t)t^2$$

$$b_{3,3}(t) = t^3$$

Ukoliko Bernsteinove težinske funkcije uvrstimo u jednadžbu krivulje dobivamo sljedeću jednadžbu krivulje:

$$\vec{p}(t) = \sum_{i=0}^{3} \vec{r_i} b_{i,3}(t)$$

$$\vec{p}(t) = \vec{r_0}(1-t)^3 + \vec{r_1} \cdot 3(1-t)^2 t + \vec{r_2} \cdot 3(1-t)t^2 + \vec{r_3} t^3$$

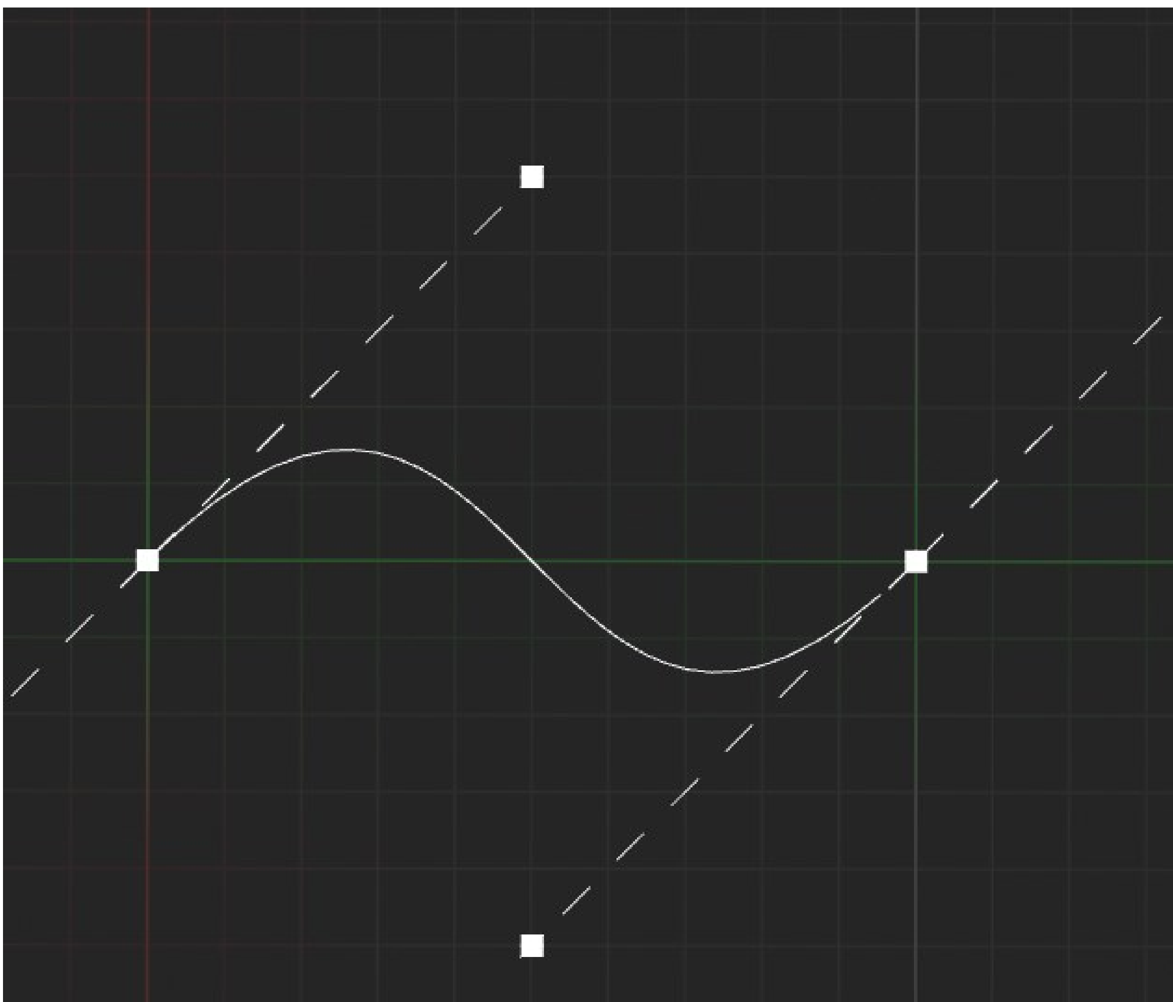
Prethodna jednadžba definira krivulju na temelju 4 zadane točke. Ukoliko želimo zadati više od 4 točke koje definiraju Bézierovu krivulju potrebno je koristiti kompozitnu Bézierovu krivulju. Kod kompozitne Bézierove krivulje završna točka jedne kubne Bézierove krivulje je početna točka sljedeće Bézierove krivulje. Time je osigurano svojstvo kontinuiranosti. U *Unreal Engine*-u osnovna komponenta za izradu krivulje, *USplineComponent*, kao osnovu za crtanje krivulje koristi upravo prethodno navedenu kompozitnu Bézierovu krivulju. Svaki dio krivulje je zadan sa 4 točke. Prva i četvrta točka su početna i završna točka krivulje, a druga i treća točka su točke koje se nalaze na tangenti na krivulju iz početne, odnosno završne točke. Ukoliko sa $\vec{P_0}$ i $\vec{P_1}$ označimo radij-vektore krajnjih točaka krivulje, a sa $\vec{T_0}$ i $\vec{T_1}$

označimo vektore koji spajaju krajnje točke krivulje sa zadanim točkama na njihovim tangentama. Tada je kontrolni poligon moguće zapisati pomoću sljedećih radij vektora: $\overrightarrow{P_0},\ \overrightarrow{P_0}+\overrightarrow{T_0},\ \overrightarrow{P_1}-\overrightarrow{T_1},\ \overrightarrow{P_1}$. Ukoliko uvrstimo navedene vrhove kontrolnog poligona u jednadžbu krivulje dobivamo sljedeći izraz za krivulju korištenu u ovome radu:

$$\vec{p}(t)=\overrightarrow{P_0}(1-t)^3+(\overrightarrow{P_0}+\overrightarrow{T_0})\cdot 3(1-t)^2t+(\overrightarrow{P_1}-\overrightarrow{T_1})\cdot 3(1-t)t^2+\overrightarrow{P_1}t^3$$

koji nakon raspisivanja po vrhovima kontrolnog poligona iznosi:

$$\vec{p}(t)=(2t^3-3t^2+1)\cdot\overrightarrow{P_0}+3(1-t)^2\cdot\overrightarrow{T_0}+3t^2(1-t)\cdot\overrightarrow{T_1}+t^2(3-2t)\cdot\overrightarrow{P_1}$$



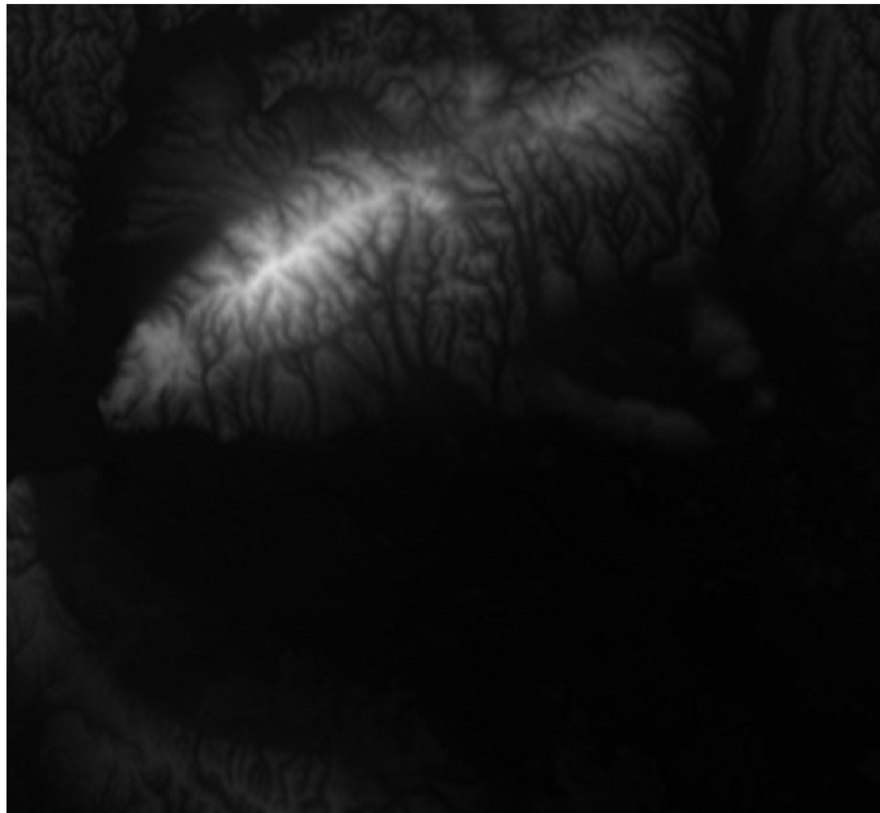
Slika 1.1: Prikaz Bézierove krivulje u *Unreal Engine*-u

2. Korištene komponente

Za izradu ovoga završnog rada korištene su razne komponente koje su dostupne u grafičkom programskom pogon *Unreal Engine*. Neke od glavnih komponenata su visinska mapa, materijali i mreža poligona. Visinska mapa je korištena kao podloga na kojoj će se iscrtavati krivulja, a materijal i mreža poligona sudjeluju u postupku sjenčanja.

2.1. Visinska mapa

Visinska mapa je tekstura koja u sebi sadrži podatke o visini terena koji se nalazi u sceni. Najčešće je riječ o crno-bijeloj slici na kojoj svijetlije nijanse boje predstavljaju koordinate na terenu koje su na većoj visini, a tamne nijanse predstavljaju koordinate koje su na nižoj visini. Kako bi *Unreal Engine* mogao uspješno učitati visinsku mapu, ona mora zadovoljavati nekoliko kriterija. Potrebno je da slika koja se koristi bude 16-bitna slika u .png ili .raw formatu (inače može doći do neočekivanih rezultata učitavanja slike, poput površine koja nije glatka.



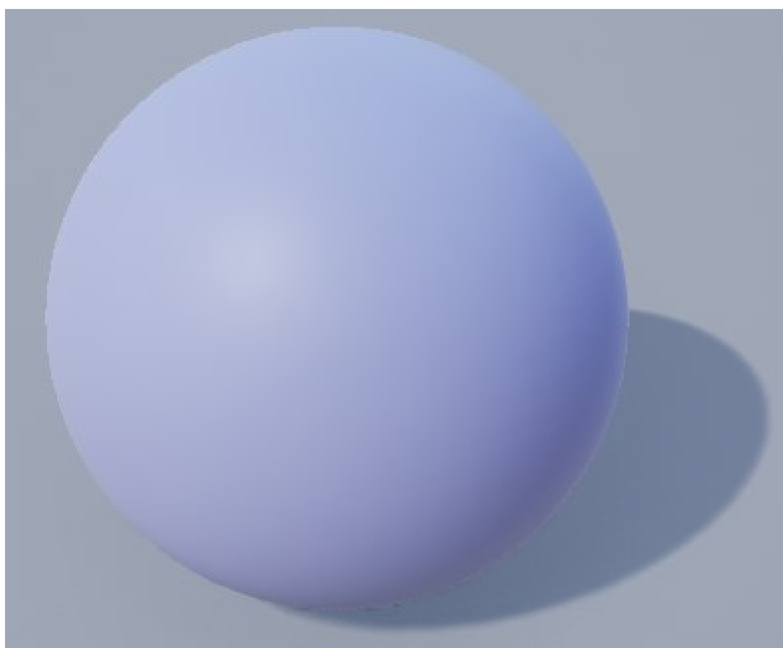
Slika 2.1: Visinska mapa zagrebačkog područja (izvor: https://tangrams.github.io/heightmapper/)

Također moguće je ručno uređivati visinsku mapu pomoću uređivača visinske mape koji dolazi ugrađen u *Unreal Engine*. Nakon što se visinska mapa učita i *Unreal*

Engine obavi proces stvaranja podloge teren se može dodatno uređivati mijenjanjem materijala.

2.2. Materijali i mreže poligona

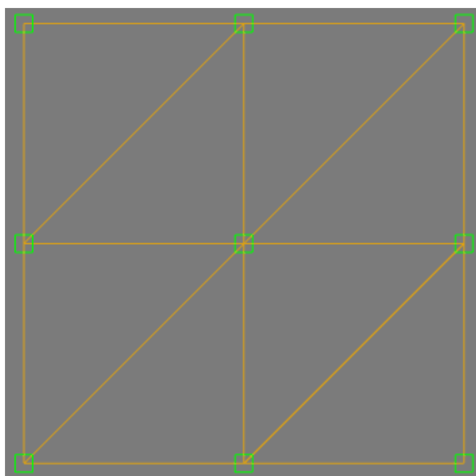
Svaki objekt koji se prikazuje u programu definiran je s dvije stvari, materijalom i mrežom poligona. Materijal definira karakteristike površine svakog objekta. Također možemo reći da se materijal koristi za izračune kako se svjetlost ponaša nakon što udari u površinu objekta. Pomoću materijala se mogu uređivati razne stvari, poput boje površine ili slike koja će se prikazati na površini. Također je, između ostalog, moguće mijenjati hrapavost ili sjaj površine, kao i razna svojstva sjenčara koji se koristi. U ovom radu je kao materijal korištena obična plava boja.



Slika 2.2 Materijal korišten u izradi krivulje

Mreža poligona određuje geometrijska svojstva svakog objekta. Riječ je o skupu koji se sastoji od niza međusobno povezanih vrhova i trokuta koji izgrađuju objekt. *Unreal Engine* omogućava razne optimizacije te mreže koje poboljšavaju performanse programa (poput spremanja podataka o mreži u memoriju grafičke kartice) te je također ugrađena potpora za osnovne geometrijske transformacije, koje uključuju translaciju, rotaciju i skaliranje. Za izradu tih mreža poligona se koriste razni programi, poput *Blender*-a, nakon čega se te mreže mogu učitati u projekt, a osim toga je podržano i proceduralno generiranje poligona. U ovom radu je korištena mreža od 8 jednakokračnih pravokutnih trokuta koji su međusobno povezani u kvadrat. Brojka od 8 trokuta je odabrana zbog dobrog omjera između

performansi (zato što je mreža dovoljno mala da bi se omogućilo brzo crtanje) i izgleda (zato što u slučaju da je korišteno manje od 8 trokuta razne transformacije koje su potrebne kako bi ta mreža pratila točke krivulje ne bi imale dobre rezultate).



Slika 2.3 Mreža poligona

3. Implementacija

3.1 Izrada krivulje

Prilikom izrade krivulje na visinskoj mapi prvi korak je stvaranje 2D prikaza krivulje pomoću dijela koda koji je prikazan u nastavku:

```
for (int index = 0; index < positions.Num(); index++)
{
    SplineComponent->AddSplinePoint(FVector(positions[index].X,
positions[index].Y, 0), ESplineCoordinateSpace::Local);
}
```

Prethodno navedeni kod prolazi kroz sve koordinate koje korisnik odredi kao koordinate kroz koje prolazi krivulja i od tih točaka stvara krivulju uz pomoć formule definirane u prvom poglavlju. U isto vrijeme se vrši i izračun točaka koje se nalaze na tangenti na krivulju koja prolazi kroz zadanu točku krivulje. Zatim nakon što je krivulja izrađena u 2d prostoru sljedeći korak je transformirati tu krivulju u 3D prostor. To je učinjeno pomoću sljedećeg isječka koda:

```
TArray<FVector> locationsToAdd;
float currentDistance = 0;
float totalDistance = SplineComponent->GetSplineLength();
int numberOfPoints = totalDistance / pointPrecision;
for (int index = 0; index <= numberOfPoints; index++)
{
    FVector oldPosition = SplineComponent-
>GetLocationAtDistanceAlongSpline(currentDistance,
ESplineCoordinateSpace::Local);
    locationsToAdd.Add(this-
>getSplineLocationAtPosition(FVector2D(oldPosition.X,
oldPosition.Y)));
    currentDistance += pointPrecision;
}
FVector endPoint = SplineComponent-
>GetLocationAtSplinePoint(SplineComponent-
>GetNumberOfSplinePoints() - 1, ESplineCoordinateSpace::Local);
locationsToAdd.Add(this-
>getSplineLocationAtPosition(FVector2D(endPoint.X, endPoint.Y)));
return locationsToAdd;
```

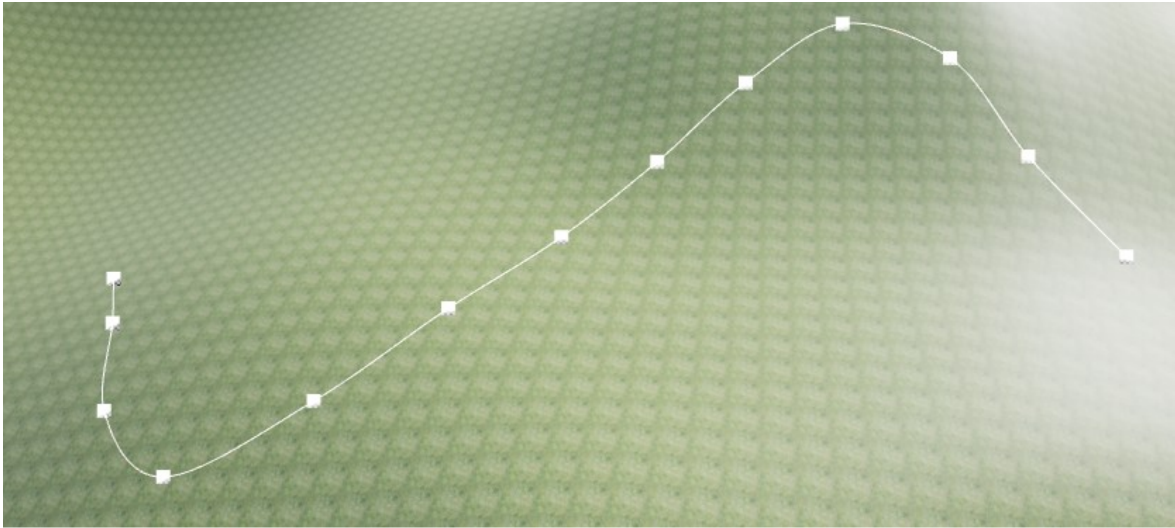
Navedena metoda služi za izračunavanje visine točaka koje određuju krivulju na visinskoj mapi. Točke koje se izračunavaju se nalaze na određenom razmaku duž

krivulje jedna od druge. To je napravljeno zato što je moguće da, ovisno u razmaku početno zadanih točaka i nagibu na visinskoj mapi, kao rezultat dobijemo krivulju koja ide ispod ili iznad visinske mape. Prethodni isječak koda na početku stvara polje u koje će se spremati sve izračunate koordinate točaka u 3D prostoru. Zatim računa broj točaka koje je potrebno generirati kao količnik između ukupne duljine krivulje i željenog razmaka između točaka. Nakon toga se ponavlja sljedeći proces za svaku od točaka koje treba izračunati: izračunaju se koordinate točke koja je na određenoj udaljenosti od početne točke duž krivulje, zatim se na tim koordinatama računa visina koja je na tom mjestu na visinskoj mapi, iz koordinata u 2D prostoru i visine se stvaraju 3D koordinate koje se dodaju u polje koordinata konačnih točaka i na kraju petlje se povećava trenutna udaljenost kako bi tu udaljenost mogli koristiti za računanje sljedeće točke. Na kraju je isti postupak potrebno provesti i za završnu točku i nakon što smo izračunali koordinate za sve točke od tog polja stvaramo novu konačnu krivulju kojoj su zadane točke upravo te točke iz poligona. Računanje visine koordinate na visinskoj mapi se provodi na sljedeći način:

```
UWorld* world{ this->GetWorld() };
if (world)
{
    FVector startLocation{ point.X, point.Y, 5000 };
    FVector endLocation{ point.X, point.Y, -5000 };
    FHitResult hitResult;
    world->LineTraceSingleByObjectType(
        OUT hitResult,
        startLocation,
        endLocation,
      FCollisionObjectQueryParams(ECollisionChannel::ECC_WorldStatic),
        FCollisionQueryParams()
      );
      if (hitResult.GetActor()) return hitResult.ImpactPoint.Z;
}
return 0;
```

U gornjem kodu prvo je potrebno dohvatiti referencu na objekt u kojem se spremaju podaci o svijetu u kojem se krivulja nalazi. Zatim se zadaju dvije točke dužine koja se koristi za algoritam praćenja linije. Bitno je da zadana linija bude okomita na xy-ravninu. Nakon što provedemo algoritam praćenja linije kao rezultat dobivamo lokaciju na kojoj se sijeku podloga i linija. Zatim kao rezultat je potrebno samo vratiti

visinu na kojoj je došlo do presijecanja. Nakon čitavog postupka imamo krivulju koja se nalazi na površini visinske mape i sljedeći korak je proširiti tu krivulju.



Slika 3.1 Rezultat nakon stvaranja krivulje na visinskoj mapi

3.2 Proširivanje krivulje

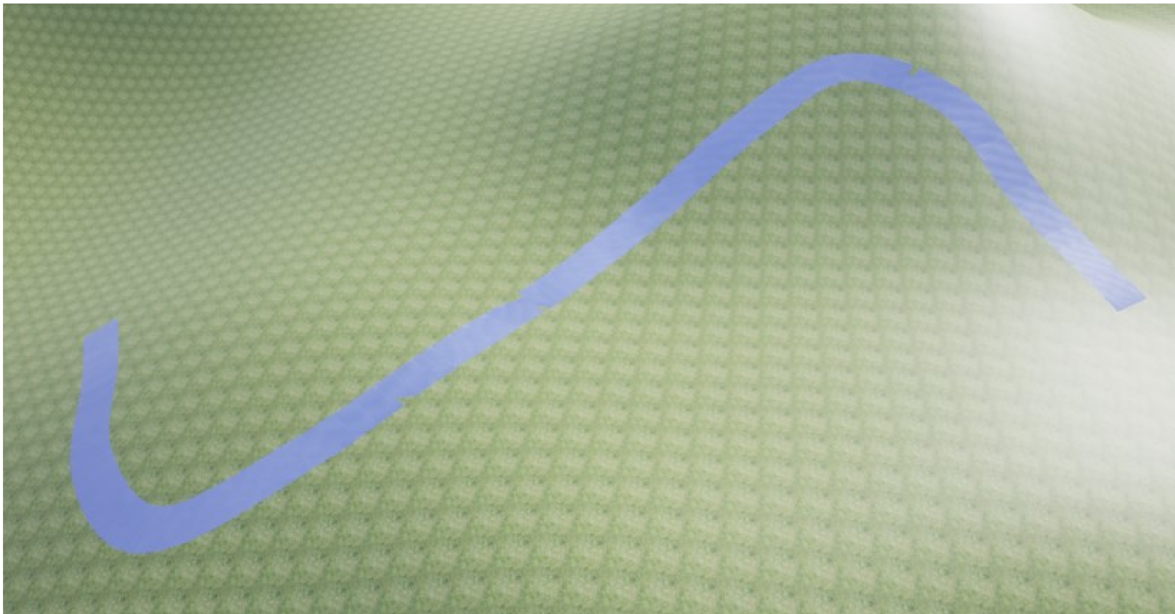
Proširivanje krivulje se vrši pomoću materijala i mreže poligona navedenih u 2. poglavlju.

```
float curveLength = SplineComponent->GetSplineLength();

int numberOfMeshes = (curveLength - fmod(curveLength, spacing)) / spacing;

float scaleFactor = lineWidth / Mesh->GetBoundingBox().GetSize().X;

for (int index = 0; index < numberOfMeshes + 1; index++)

{

    USplineMeshComponent* splineMeshComponent = NewObject<USplineMeshComponent>(this, USplineMeshComponent::StaticClass());

    SplineMeshComponents.Add(splineMeshComponent);

    splineMeshComponent->SetStaticMesh(Mesh);


    FVector startLocation = SplineComponent->GetLocationAtDistanceAlongSpline(index * spacing, ESplineCoordinateSpace::Local);
```

```
    FVector startTangent = SplineComponent-
>GetDirectionAtDistanceAlongSpline(index * spacing,
ESplineCoordinateSpace::Local);


    FVector endLocation = SplineComponent-
>GetLocationAtDistanceAlongSpline((index + 1) * spacing,
ESplineCoordinateSpace::Local);

    FVector endTangent = SplineComponent-
>GetDirectionAtDistanceAlongSpline((index + 1) * spacing,
ESplineCoordinateSpace::Local);


    splineMeshComponent->SetStartAndEnd(startLocation,
startTangent, endLocation, endTangent);

    splineMeshComponent->SetSplineUpDir(this-
>getUpVectorAtPoint(FVector2D(startLocation.X, startLocation.Y)),
true);


    splineMeshComponent->SetStartScale(FVector2D(scaleFactor,
scaleFactor), true);

    splineMeshComponent->SetEndScale(FVector2D(scaleFactor,
scaleFactor), true);

    splineMeshComponent->SetMobility(EComponentMobility::Movable);

    splineMeshComponent->AttachToComponent(SplineComponent,
FAttachmentTransformRules::KeepRelativeTransform);

}
```

Na početku je potrebno izračunati broj mreža poligona koje je potrebno napraviti. To se računa na način da se ukupna duljina krivulje podijeli s međusobnom udaljenošću između 2 susjedne mreže poligona koje zadaje korisnik. Također se računa faktor skaliranja s kojim će biti potrebno pomnožiti mrežu koja će nastati u postupku proširivanja krivulje, taj faktor je korišten kako bi se korisniku omogućilo da mijenja širinu krivulje. Nakon toga za svaku mrežu poligona koja se treba napraviti provodi se sljedeći postupak. Prvo se stvara novi objekt u sceni i tom objektu se postavlja mreža poligona. Zatim je potrebno odrediti lokaciju te mreže u

sceni. Kao početna točka se koristi točka koja je na udaljenosti jednakoj indeksu mreže pomnoženom s razmakom između dvije mreže. Na sličan se način računa i konačna točka. Osim početne i završne točke mreže potrebno je i odrediti tangente mreže koje se koriste kako bi se mreža mogla saviti u smjeru krivulje. Kao tangente za mrežu se uzimaju tangente krivulje na početnoj, odnosno završnoj lokaciji mreže. Osim tih parametara potrebno je odrediti i vektor normale mreže koji kao svrhu ima postavljanje nagiba mreže, koja bi inače bila paralelna s xy-ravninom i na taj način ne bi pratila zakrivljenost površine. Vektor normale se računa na sličan način kao i visina na koordinati visinske mape. Provodi se algoritam praćenja linije i nakon što je izračunata kolizija u točki u kojoj je došlo do kolizije se računa vektor normale podloge i taj vektor se koristi kao vektor normale mreže. Nakon toga postavimo sve vrijednosti mreže na prethodno izračunate parametre, skaliramo mrežu poligona na željenu veličinu kako bi postavili širinu proširene krivulje i zatim mrežu spojimo s komponentom koja sprema podatke o krivulji.



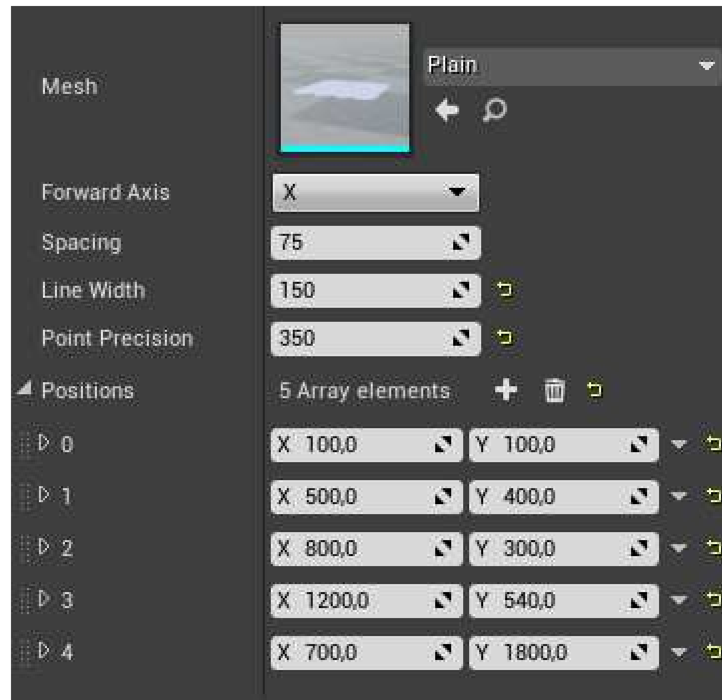
Slika 3.1 Rezultat algoritma za prikaz krivulje nakon proširivanja

4. Korištenje programskog rješenja

Kako bi se navedeno rješenje moglo iskoristiti u korisničkoj aplikaciji najprije je potrebno učitati programski kod napisan za ovaj projekt u postojeći *Unreal Engine* projekt. Nakon što je kod za krivulju dio projekta potrebno je u pregledniku sadržaja (engl. Content Browser) dodati novi razred zasnovan na shemi (engl. Blueprint Class). U sljedećem izborniku se izabire roditeljski razred (engl. Parent Class), i to je potrebno odabrati razred *MySplineActor* (razred čiji je kod obrađen u ovom radu). Na kraju tog postupka u pregledniku sadržaja imamo novi razred kojeg zatim možemo stvoriti u sceni. Jednom kada u sceni imamo instancu objekta u detaljima objekta možemo mijenjati razne parametre. U nastavku su navedeni ti parametri:

- Mreža poligona (engl. Mesh): određuje koja se mreža poligona koristi za iscrtavanje krivulje prilikom njenog proširivanja. Pomoću ovog parametra možemo mijenjati sve moguće vrijednosti materijala ili mreže poligona, kao na primjer: boju, sjaj, oblik koji se iscrtava…
- Prednja os (engl. Forward Axis): određuje smjer iscrtavanja, odabir ovisi o svojstvima mreže poligona. Ukoliko je čitava mreža u jednoj ravnini, a ta ravnina je paralelna s nekom od osi, tada je preporuka da se odabere ta os.
- Razmak (engl. Spacing): određuje za koliko su udaljene dvije početne točke mreže poligona, u slučaju da se koristi manji broj tada je poboljšana preciznost prilikom proširivanja krivulje, međutim u isto vrijeme korisnik dobiva smanjene performanse. Preporuka je da ovaj broj bude manji od širine krivulje.
- Širina linije (engl. Line Width): određuje širinu iscrtane proširene krivulje.
- Preciznost koordinata (engl. Point Precision): određuje razmak između dvije točke duž krivulje na kojima se računa visina. Preporuka je da se ovaj parametar prilagodi karakteristikama visinske mape i međusobnim udaljenostima zadanih točki. Ukoliko je na određenom dijelu krivulje nagib na visinskoj mapi velik, ili je udaljenost između točaka koje je korisnik zadao velika tada se preporučuje da korisnik odabere manji broj, U slučaju da je riječ o manjem nagibu na visinskoj mapi tada korisnik može odabrati veći broj kako bi smanjio broj točaka na kojima se računa visina na visinskoj mapi i tako poboljšao performanse prilikom generiranja krivulje.

- Pozicije (engl. Positions): određuje koordinate točaka kroz koje krivulja mora prolaziti. Točke se mogu birati proizvoljno, broj točaka koje korisnik zada nije određen. Korisnik zadaje koordinate u 2D obliku.



Slika 4.1 Popis parametara

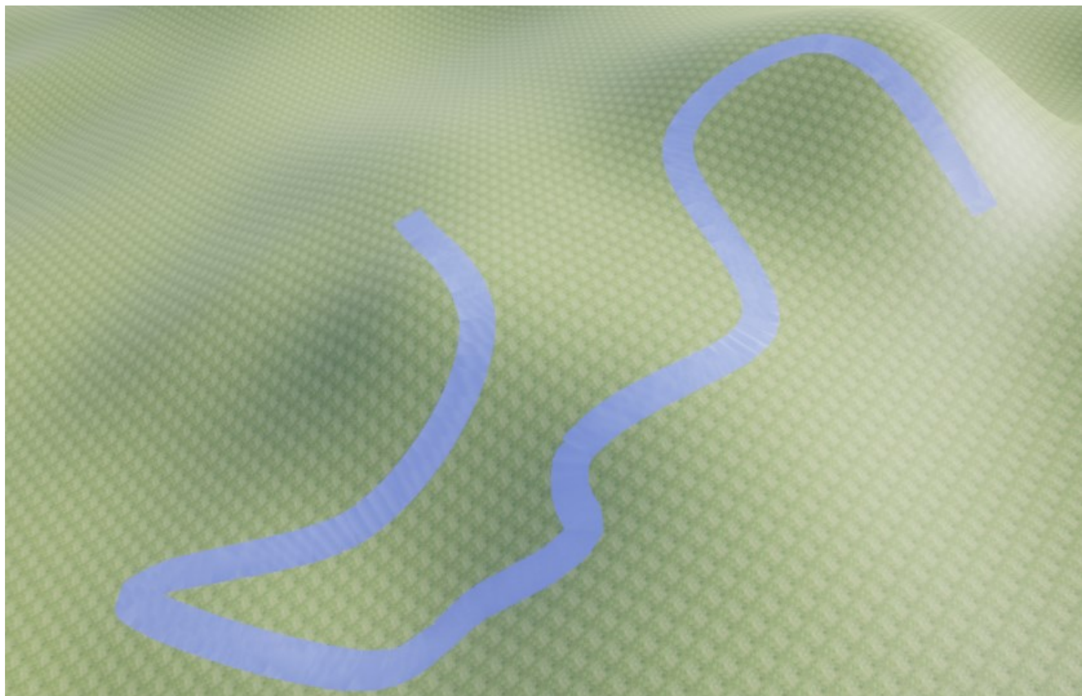
5. Performanse implementacije

Na vremenske karakteristike ovog rješenja najveći utjecaj ima duljina krivulje, dulja krivulja kao posljedicu ima stvaranje većeg proja mreža poligona i njihovog iscrtavanja, što je najzahtjevnija funkcionalnost. Najbolje performanse su ostvarive ukoliko se za mrežu poligona koristi kvadrat koji je izgrađen od dva jednakostranična pravokutna trokuta, međutim to za posljedicu ima manju kvalitetu krivulje. Također na samo vremensko trajanje utjecaj ima i materijal koji se koristi. Ukoliko je riječ o materijalu koji je izgrađen od samo jedne boje i bez dodatnih parametara tada sjenčar izvodi 118 instrukcija. Ukoliko se dodaju neki novi parametri, poput sjaja ili hrapavosti površine, taj broj se razlikuje. Na performanse možemo utjecati i promjenom ostalih parametara, ukoliko smanjimo razmak između početnih točki dvije mreže poligona tada je potrebno generirati više poligona što ima negativan utjecaj na brzinu izvođenja. Manji utjecaj također ima i preciznost koordinata. Što je udaljenost između točaka kojima se računa visina manja, to je potrebno izračunati visinu za više točaka. Iako taj postupak je brži u odnosu na samo generiranje mreža poligona tako da to ne predstavlja veliki problem za performanse. Tijekom testiranja manji problemi vezani uz performanse su uočeni kod krivulja čija duljina iznosi oko 40000 piksela, dok problemi s performansama postaju očigledni kada je krivulja duga 65000 piksela (korišteni razmak između početnih točki mreža poligona je iznosio 50 piksela, a širina krivulje 100 piksela)

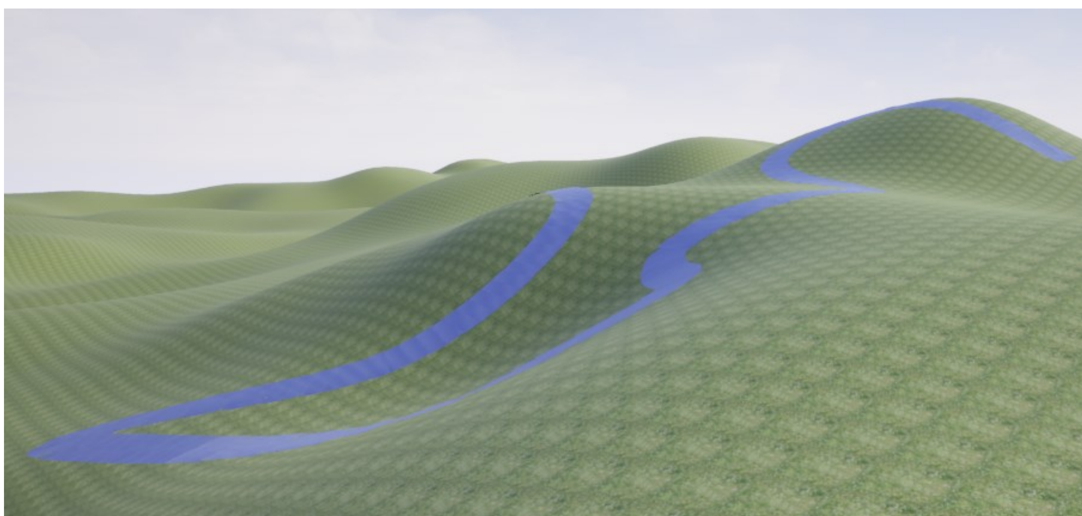
6. Rezultati i moguća poboljšanja

6.1 Rezultati

U ovom radu je prikazana osnovna funkcionalnost prikazivanja proširenih krivulja na modelu visinske mape. Kao ulazni parametar korisnik zadaje koordinate točaka u 2D prostoru. Nakon toga se provodi algoritam određivanja krivulje, računanja visine i nagiba na površini visinske mape. Zatim se iz krivulje i izračunatih vrijednosti generiraju mreže poligona koje je potrebno iscrtati. Na kraju postupka kao rezultat dobivamo krivulju prikazanu na površini visinske mape.



Slika 6.1 Primjer proširene krivulje promatran iz ptičje perspektive



Slika 6.2 Prethodna krivulja promatrana s površine

6.2 Moguća poboljšanja

Krivulje se trenutno zadaju s lokacijama točaka kroz koje prolazi krivulja, a tangente koje se koriste u izračunu jednadžbe krivulje se automatski zadaju. Jedan od načina za poboljšanje rješenja bi bilo omogućiti korisniku ručno unošenje tangente koja će se koristiti. Time bi korisnik dobio veću mogućnost konfiguracije zakrivljenosti krivulje, međutim to bi za posljedicu imalo teže zadavanje krivulje, stoga je u ovom radu odabrano samo zadavanje točaka kroz koje krivulja prolazi. Također u radu je podržano crtanje Bézierove krivulje. Bézierova krivulja nam pruža mogućnost crtanja mnogih krivulja. Bilo bi dobro omogućiti crtanje i nekih drugih krivulja koje je lakše zadati od Bézierove krivulje, poput sinusoide ili kružnice. Primjerice, za sinusoidu bi bilo potrebno samo zadati amplitude, perioda, pomaka po x osi i po želji korisnika ograničiti područje domene funkcije (što je manji broj parametara u odnosu na sličnu Bézierovu krivulju. Uz to, još jedno od poboljšanja bi bilo direktno međusobno spajanje pojedinih mreža poligona kako bi se dodatno popravile performanse i popunile sitne pukotine koje mogu nastati prilikom proširivanja krivulje ukoliko na određenom mjestu dolazi do velike promjene nagiba terena.

7. Zaključak

Cilj ovog rada je bio istražiti postupak izrade proširene krivulje na modelu visinske mape.

Stvaranje krivulje je odrađeno koristeći kubnu Bézierovu interpolacijsku krivulju. Korisnik zadaje točke kroz koje krivulja mora prolaziti u 2D prostoru nakon čega se za te točke generira krivulja u xy-ravnini. Zatim se za točke duž krivulje na određenim razmacima računa visina te točke na visinskoj mapi koristeći algoritam praćenja linije, nakon čega je moguće generirati krivulju na površini visinske mape.

Jednom kada imamo krivulju slijedi postupak proširivanja krivulje. On je napravljen iscrtavanjem mreža poligona duž krivulje. Mreže se translatiraju i rotiraju u odnosu na točke krivulje i vektor normale na tangencijalnu ravninu koja prolazi točkom krivulje, čiji izračun je također ostvaren korištenjem algoritma praćenja linije. Nakon toga se mreže skaliraju kako bi bile širine koju zadaje korisnik.

Korisniku je omogućeno mijenjanje izgleda krivulje, širine krivulje te raznih drugih parametara koje utječu na kvalitetu, odnosno performanse krivulje. U budućnosti kao nadogradnju je moguće omogućiti korisniku ručno postavljanje tangenti u određenim točkama čime bi smo mogli preciznije određivati oblik krivulje, kao i dodati neke druge vrste krivulja.

Literatura

[1] Željka Mihajlović, Marko Čupić, *Interaktivna računalna grafika kroz primjere u OpenGL-u*, 2021.

[2] WorldofLevelDesign, *Everything You Need to Know About Landscape Heightmaps for UE4* (2020, travanj), Poveznica: https://www.worldofleveldesign.com/categories/ue4/landscape-heightmap-guide.php; pristupljeno 3. lipnja 2021.

[3] Epic Games, *Materials*, Poveznica: https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/Materials/; pristupljeno 5. lipnja 2021.

[4] Epic Games, *Static Meshes*, Poveznica: https://docs.unrealengine.com/4.26/en-US/WorkingWithContent/Types/StaticMeshes/; pristupljeno 6. lipnja 2021.

[5] Epic Games, *Static Meshes*, Poveznica: https://docs.unrealengine.com/4.26/en-US/API/Runtime/Engine/Components/USplineComponent/; pristupljeno 5. svibnja 2021.

Sažetak

Prikaz proširenih krivulja na modelu visinske mape

Ovaj završni rad opisuje postupak prikazivanja proširenih krivulja na modelu visinske mape u grafičkom programskom pogon *Unreal Engine*. Kao osnova za izgradnju krivulja je korištena kubna Bézierova interpolacijska krivulja. U radu je opisan postupak dobivanja svih točaka koje čine krivulju. Za određivanje z vrijednosti koordinata krivulja korišten je algoritam praćenja zrake. Proširivanje je provedeno iscrtavanjem mreža poligona duž krivulje, za što je opisan postupak. Korisniku je omogućena izmjena raznih parametara koji utječu na kvalitetu krivulje i brzinu izvođenja. Napravljena je analiza trajanja izvođenja algoritma i analiza mogućih nadogradnji postupka.

Ključne riječi: krivulja, visinska mapa, Unreal Engine, algoritam praćenja zrake, mreže poligona

Summary

Visualization of Extruded Curves on Elevation Map Model

This thesis describes the algorithm for visualization of extruded curves on elevation map model in the Unreal Engine. Cubic Bézier interpolation curve has been used as the basis for the construction of the curve. Thesis also describes the process for generating all the points that makes the curve. Ray tracing algorithm has been used to get the z coordinate of the curve. Extruding of the curve has been done by rendering the meshes along the curve. User can manually change different parameters that affect the duration of the rendering and quality of the result. The thesis also contains the time analysis of the process and possible improvements.

Keywords: curve, heightmap, Unreal Engine, ray tracing algorithm, mesh