SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1296

VIZUALIZACIJA ATRAKTORA

Dominik Šarić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1296

VIZUALIZACIJA ATRAKTORA

Dominik Šarić

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1296

Pristupnik:	**Dominik Šarić (0036540383)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentorica:	prof. dr. sc. Željka Mihajlović

Zadatak: **Vizualizacija atraktora**

Opis zadatka:

Proučiti matematičke modele za simulaciju kaotičnih procesa koji rezultiraju pojavom atraktra. Posebice proučiti Lorenzov atraktor. Razviti vizualizacijske metode koje omogućuju prikaz ovakvih atraktora kao i istraživanje ostvarenih prikaza promjenama parametara modela, primjenama pozicija kamere i postavljanjem raznih vizualnih parametara kao što je primjerice boja. Načiniti testiranje na nizu primjera. Analizirati i ocijeniti ostvarene rezultate. Diskutirati upotrebljivost ostvarenih rezultata kao i moguća proširenja. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++ i grafičko programsko sučelje OpenGL, programsku biblioteku GLFW i grafičko programsko sučelje Dear ImGui. 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: 14. lipnja 2024.

Sadržaj

Uvod

Jedno od osnovnih područja računalne grafike je vizualizacija i simulacija raznih sustava od kojih je neke moguće pronaći i u prirodi. Promatrajući prirodu, čovjek znatiželjne naravi se često može zapitati kako i zašto se neke stvari u prirodi ponašaju onako kako se ponašaju. Nakon što na kraju uspije odgovoriti na neke od ovakvih zagonetki, koristeći već razvijene matematičke alate zapisuje promatrano ponašanje u obliku raznih jednadžbi, integrala i sličnih aparata.

Jedan dio ovog područja pokrivaju diferencijalne jednadžbe koje nam govore kako se razne stvari mijenjaju kroz vrijeme. Neki od primjera diferencijalnih jednadžbi mogu se pronaći u fizikalnim sustavima kao što su transferi topline, propagacija valova, mehanika fluida, pa i u području kvantne mehanike na primjeru poznate Schrödingerove jednadžbe.

U ovome radu bit će obrađeno i vizualizirano jedno od područja koje se kao i već spomenuta, modelira pomoću diferencijalnih jednadžbi – kaotični atraktori. Kaotične atraktore i njihove jednadžbe je također moguće pronaći u prirodi na primjerima kao što su modeliranje vremenskih prognoza, predviđanja populacija te opisivanje određenih kemijskih reakcija. No iza ovih korisnih uporaba kaotičnih atraktora, skriva se i još jedno od njihovih svojstava, a to je izrazita ljepota vizualizacije njihovih putanja koju ću u ovom radu i realizirati.

1. Kaotični atraktori

Kaotični atraktori su objekti u području nelinearne dinamike i teorije kaosa, koji predstavljaju stanja sustava koja se ponašaju deterministički, ali naizgled nasumično. Oni se javljaju u mnogim prirodnim i umjetnim sustavima te su ključni za razumijevanje kompleksnog ponašanja tih sustava. Jedna od najzanimljivijih karakteristika kaotičnih atraktora je njihova osjetljivost na početne uvjete.

Osjetljivost na početne uvjete opisuje kako male promjene u početnim parametrima mogu dovesti do dramatično različitih ishoda. To znači da, iako je sustav deterministički (tj. njegovo buduće stanje je u potpunosti određeno njegovim trenutnim stanjem), dugoročno ponašanje sustava je nepredvidivo. Ova osjetljivost čini kaotične sustave izuzetno složenim za analizu i predviđanje [2].

Kaotični atraktori su matematičke strukture koje privlače putanje sustava u kompleksne i često fraktalne obrasce, koji pokazuju osjetljivost na početne uvjete, ali su ograničeni unutar određenog globalnog područja. Primjeri poznatih kaotičnih atraktora uključuju Lorenzov atraktor, Rösslerov atraktor, Henonovu mapu i mnoge druge.

Iako se kaotični atraktori mogu činiti nasumičnima, njihova dinamika je zapravo vrlo strukturirana i slijedi determinističke zakone. Ova struktura nam omogućava da ih proučavamo i modeliramo, otkrivajući pritom duboke uvide u prirodu složenih sustava. Kroz razumijevanje kaotičnih atraktora, možemo bolje shvatiti kako se složeni obrasci pojavljuju u prirodi, od vremenskih obrazaca i dinamike fluida do bioloških i kemijskih procesa.

1.1. Lorenzov atraktor

Lorenzov atraktor (Slika 1.1) je jedan od najpoznatijih primjera kaotičnog sustava, otkriven od strane meteorologa Edwarda Lorenza 1963. godine dok je proučavao pojednostavljeni model atmosferske konvekcije. Ovaj atraktor ilustrira kako deterministički sustavi mogu pokazivati kaotično ponašanje, što znači da su njihove putanje izuzetno osjetljive na početne uvjete.

Matematički, Lorenzov atraktor je definiran sustavom triju nelinearnih diferencijalnih jednadžbi [1]:

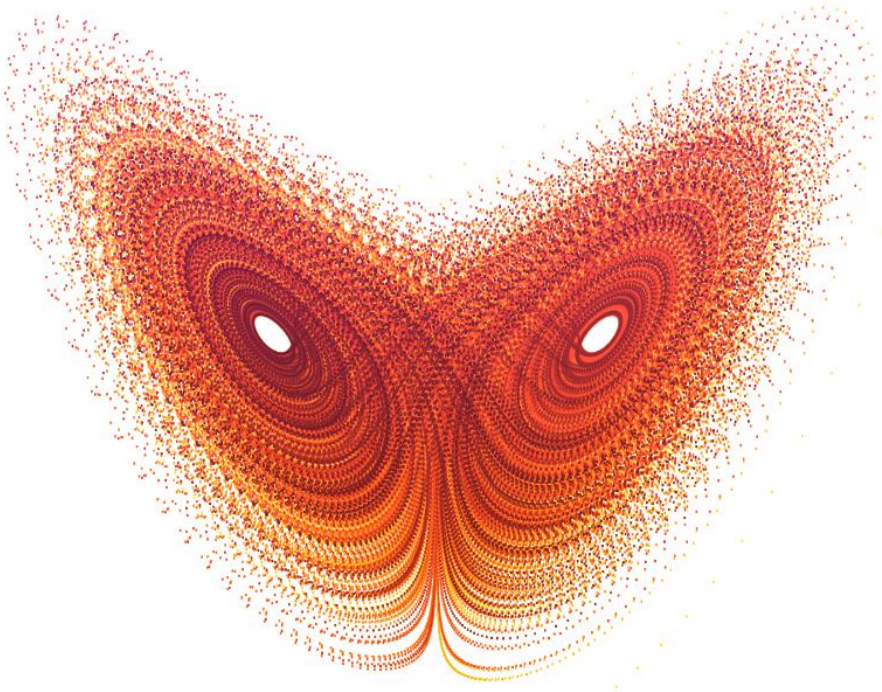
$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

gdje su σ, ρ i β parametri sustava koji određuju njegovu dinamiku. Vrijednosti ovih parametara koje generiraju kaotično ponašanje su σ=10, ρ=28 i β=8/3.

Kao što je već i rečeno jedna od karakteristika Lorenzovog atraktora je njegova osjetljivost na početne uvjete. Male promjene u početnim uvjetima rezultiraju drastično različitim putanjama, što znači da je dugoročno ponašanje sustava nepredvidivo, čak iako je sustav deterministički. Ova osjetljivost je ponekad slikovito opisana kroz metaforu "efekta leptira", gdje se kaže da lepet krila leptira u Brazilu može izazvati tornada u Teksasu [2].



Slika 1.1 - Jedna od vizualizacija Lorenzovog atraktora [3]

1.2. Metoda rješavanja diferencijalnih jednadžbi

Metoda korištena u ovome radu za rješavanje diferencijalnih jednadžbi koje opisuju atraktore je eksplicitni Eulerov postupak [8].

Eulerov postupak je jednostavna numerička tehnika za rješavanje diferencijalnih jednadžbi. Koristi se za aproksimaciju rješenja diferencijalnih jednadžbi s početnim vrijednostima, tako da iterativno računa nove vrijednosti funkcije.

Da bismo opisali Eulerovu metodu za tri varijable u kontekstu kaotičnih atraktora, uzmimo kao primjer opisani Lorenzov atraktor i njegov pripadajući sustav jednadžbi.

Eulerov postupak ukratko se može opisati sljedećim koracima:

1. Postavljanje početnih vrijednosti x_0, y_0, z_0
2. Definiranje vremenskog koraka h
3. Iterativno računanje nove vrijednosti varijabli koristeći pripadne diferencijalne jednadžbe, trenutne vrijednosti varijabli i definirani vremenski korak

Koraci za jednu iteraciju Eulerove metode nad sustavom jednadžbi Lorenzovog atraktora izgledaju ovako:

$$x_{n+1} = x_n + h * \sigma(y_n - x_n)$$

$$y_{n+1} = y_n + h * [x_n(\mathrm{p} - z_n) - y_n]$$

$$z_{n+1} = z_n + h * (x_n * y_n - \beta z_n)$$

Gdje su x_n, y_n i $z_n$trenutne vrijednosti varijabli na koraku n, a x_{n+1}, y_{n+1} i z_{n+1} vrijednosti na idućem koraku.

2. Korištene tehnologije i alati

Alat koji je primarno korišten za vizualizaciju samih atraktora je OpenGL API, no pored njega korišteno je još i par drugih biblioteka koje su uvelike pomogle pri realizaciji samog projekta. Neke od njih su zaslužne za ostvarivanje korisničkog sučelja, dok su druge pomogle pri realizaciji matematičkih izračuna. Više o njima u nastavku.

2.1. OpenGL

OpenGL (Open Graphics Library) je industrijski standard za grafičko programiranje koji pruža širok skup funkcija za prikazivanje dvodimenzionalne i trodimenzionalne grafike. Razvijen od strane Silicon Graphics Inc. (SGI) početkom 1990-ih, OpenGL je postao temelj mnogih grafičkih aplikacija, uključujući video igre, simulacije, CAD programe i druge vizualne aplikacije. Jedna od ključnih karakteristika OpenGL-a je njegova platforma i jezična neovisnost, što znači da može raditi na različitim operacijskim sustavima, uključujući Windows, MacOS, Linux i druge, te se može koristiti s raznim programskim jezicima kao što su C, C++, Python, Java i mnogi drugi.

OpenGL se široko koristi u industriji zabave za razvoj video igara i filmskih specijalnih efekata. Također se koristi u znanstvenim vizualizacijama, medicinskim slikama, virtualnoj stvarnosti i mnogim drugim područjima gdje su potrebne visoke performanse i kvaliteta grafike. U svrhe učenja OpenGL-a izrazito su pomogle lekcije Joeya de Vriesa [4].

2.2. Korištene biblioteke

2.2.1. GLFW

GLFW [7] je popularna biblioteka otvorenog koda koja služi za upravljanje prozorima, kontekstima za crtanje i unosom iz perifernih uređaja. Dizajnirana je za rad s OpenGL-om, Vulkanom i drugim grafičkim API-ima, pružajući jednostavno i učinkovito sučelje za razvoj grafičkih aplikacija. GLFW olakšava kreiranje i upravljanje prozorima te postavljanje konteksta za crtanje, što je ključno za razvoj interaktivnih 3D aplikacija i igara. Osim toga, podržava unos s tipkovnice, miša i joysticka, omogućujući razvoj intuitivnih i responzivnih

korisničkih sučelja. Jedna od ključnih prednosti GLFW-a je njegova multiplatformska priroda, što znači da aplikacije razvijene pomoću ove biblioteke mogu raditi na različitim operacijskim sustavima, uključujući Windows, Linux i MacOS, bez potrebe za značajnim promjenama u kodu.

2.2.2. Dear ImGUI

Dear ImGui [6] je biblioteka korisničkog sučelja koja omogućuje brzu i jednostavnu izradu grafičkih sučelja za aplikacije. Namijenjena je prvenstveno za razvojne i debug alate, ali se može koristiti i u konačnim proizvodima. Dear ImGui se ističe svojom lakoćom integracije, performansama i fleksibilnošću. Biblioteka je posebno popularna u zajednici za razvoj igara i grafičkih aplikacija zbog svoje mogućnosti brzog prototipiranja i interaktivnog prilagođavanja sučelja bez potrebe za ponovnim kompajliranjem aplikacije.

Jedan od najvažnijih aspekata Dear ImGui je njegova značajka renderiranja u stvarnom vremenu, što znači da su svi elementi korisničkog sučelja prikazani odmah nakon što su definirani. To omogućuje izradu dinamičkih i responzivnih sučelja koja se lako mogu mijenjati u stvarnom vremenu. Osim toga, Dear ImGui podržava razne kontrole kao što su gumbi, klizači, tekstualna polja, grafovi i druge komponente koje su potrebne za izradu složenih sučelja.

2.2.3. GLM

GLM (OpenGL Mathematics) [5] je popularna biblioteka za matematičke izračune koja je dizajnirana s ciljem olakšavanja razvoja grafičkih aplikacija koristeći OpenGL. Bazirana na specifikaciji GLSL-a (OpenGL Shading Language), GLM pruža funkcionalnosti za rad s vektorima, matricama, kvaternionima i drugim matematičkim strukturama koje su ključne za računalnu grafiku. Biblioteka je napisana u C++ jeziku i koristi mnoge njegove značajke, poput predložaka, kako bi omogućila efikasne i fleksibilne matematičke operacije.

Jedna od glavnih prednosti GLM-a je njegova usklađenost sa sintaksom GLSL-a, što programerima olakšava prijelaz između koda na CPU-u i shadera na GPU-u. Ovo je posebno korisno u kontekstu 3D grafike, gdje su matematičke transformacije poput translacije, rotacije i skaliranja neophodne za manipulaciju objektima i kamerama u prostoru. GLM

omogućuje jednostavno korištenje ovih transformacija kroz intuitivno sučelje koje podržava osnovne matematičke operacije, kao i naprednije funkcije poput interpolacije i generiranja perspektivnih matrica.

2.3. Programsko okruženje

Tijekom rada korišteno je programsko okruženje koje se sastojalo od operacijskog sustava Linux, kompajlera g++ i urednika teksta Vim. Linux je odabran zbog svoje stabilnosti, fleksibilnosti i snažne podrške za razvojne alate otvorenog koda. Kao operacijski sustav, Linux pruža bogato okruženje za razvoj softvera s mnoštvom dostupnih biblioteka i alata koji olakšavaju proces razvoja.

Kompajler g++ (GNU Compiler Collection) korišten je za kompajliranje C++ koda. g++ je poznat po svojoj pouzdanosti, brzini i podršci za najnovije standarde C++ jezika..

Vim je služio kao urednik teksta zbog svoje jednostavnosti, brzine i moćnih mogućnosti uređivanja. Vim je vrlo prilagodljiv i podržava širok spektar dodataka koji mogu proširiti njegove funkcionalnosti, uključujući sintaksno isticanje, automatsko dovršavanje i upravljanje projektima. Sposobnost brzog uređivanja i manipulacije kodom u Vimu značajno je ubrzala razvojni proces.

Kombinacija Linuxa, g++ i Vima pružila je robusno i učinkovito razvojno okruženje, omogućujući produktivan rad na projektu. Linux je osigurao stabilnu osnovu s bogatim skupom alata, g++ je omogućio pouzdano kompajliranje i optimizaciju koda, dok je Vim omogućio brzo i učinkovito uređivanje koda.

3. Implementacija vizualizacije atraktora

3.1. Razred *Point*

Razredom `Point` modeliramo pojedinu točku sustava kaotičnog sustava. Kako je sustav prikazan u 3D prostoru, ona se sastoji od komponenata `x`, `y` i `z`. Također, razred se sastoji i od odgovarajućih konstruktora kojima se inicijaliziraju spomenute varijable. Kako bi poziciju svake od točaka osvježavali prema odgovarajućoj jednadžbi, napisana je metoda `update()` (Programski kod 3.1) u kojoj za rješavanje pripadnih diferencijalnih metodi koristimo već opisani eksplicitni Eulerov postupak u poglavlju 1.2. U ispisu ove metode izuzete su jednadžbe atraktora 4-9.

```
glm::vec3 Point::update(int chosen_equation,  float dt, float *constants,
float delta){
    float dx, dy, dz;
    float x = this->x;
    float y = this->y;
    float z = this->z;

    switch(chosen_equation){
        case 0: // lorenz
            dx = ((y - x) * (*constants));
            dy = (x * (*(constants + 1) - z) - y);
            dz = (x * y - (*(constants + 2)) * z);
        break;

        case 1: // aizawa
            dx = (z - (*constants)) * x - (*(constants+1)) * y;
            dy = *(constants+1) * x + (z - (*(constants+2))) * y;
            dz = *(constants+3) + (*(constants+4)) * z - z * z * z - x *
                  x + (*(constants+5)) * z * x * x * x;
        break;

        case 2: // rossler
            dx = -(y + z);
```

```
            dy = x + (*(constants)) * y;
            dz = *(constants) + z * (x - (*(constants + 1)));
        break;

        case 3: // chen
            dx = *(constants) * x  - y * z;
            dy = -(*(constants + 1)) * y + x * z;
            dz = -(*(constants + 2)) * z + x * y / (*(constants+3));
        break;
    }
    dx *= dt;
    dy *= dt;
    dz *= dt;

    this->x += (dx * delta * 60);
    this->y += (dy * delta * 60);
    this->z += (dz * delta * 60);

    return glm::vec3(dx, dy, dz);
}
```

Kod 3.1 - Programski kod osvježavanja koordinata pojedine točke

Argument `chosen_equation` ove metode označava indeks odabrane jednadžbe koju korisnik želi vizualizirati. Argument `dt` određuje tzv. timestamp kojime se odabire proizvoljna brzina simulacije. Ova komponenta je vrlo osjetljiva, te za prevelike vrijednosti uzrokuje odskakanje točaka u beskonačnost jer novodobivena koordinata točke ne odgovara onoj koja leži na krivulji kaotičnog atraktora. Argument `*constants` je pokazivač na polje predefiniranih konstanti koje se uobičajeno koriste kod svakog od atraktora. U samoj vizualizaciji je korisniku omogućeno mijenjanje pojedine konstante kako bi se vidjelo da neki atraktori prestaju biti kaotični uz minimalne promjene istih. Posljednji argument, `delta`, služi kako bi se postigla konstantna brzina simulacije neovisna o trenutnom broju sličica u sekundi (FPS).

3.2. Razred *Camera*

Ovim razredom korisniku je omogućeno proizvoljno kretanje kamere kroz tipkovnicu. Ona se sastoji od nekoliko privatnih varijabli kao što su: `glm::vec3 pos, float radius, float fov, float speed_xz, float speed_y`. Ove varijable su potrebne kako bi se realizirale neke od značajki kamere kao što su pozicija, rotacija i zoomiranje. Uz sam konstruktor, gettere i settere ove klase, ona se sastoji još i od metoda `glm::mat4 get_view_matrix()` i `glm::mat4 get_projection_matrix(int width, int height)`.

Metoda `get_view_matrix()` (Kod 3.2.1) koristi se za dobivanje view matrice i njena je implementacija uvelike olakšana koristeći glm metodu `glm::lookAt()`.

```
glm::mat4 Camera::get_view_matrix(){
    return glm::lookAt(this->get_radius() * this->get_position(),
                       glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));
}
```

Kod 3.2.1 – Metoda za matricu pogleda

Metoda `get_projection_matrix()` (Kod 3.2.2) je također realizirana koristeći već napisanu glm metodu `glm::perspective()`. U ovu metodu šaljemo trenutnu širinu i visinu prozora kako bi sama matrica bila pravilno konstruirana.

```
glm::mat4 Camera::get_projection_matrix(int width, int height){
    return glm::perspective(glm::radians(this->get_fov()),
                            (float)width/height, 0.01f, 500.f);
}
```

Kod 3.2.2 – Metoda za matricu projekcije

3.3. Razred *Shader*

Razred Shader sastoji se od varijable `unsigned int ID`, koja je jedinstveni identifikator svakog inicijaliziranog sjenčara. Osim nje, u razredu je implementirana jednostavna metoda `use()` kojom se pokreće specificirani sjenčar. Što se tiče same inicijalizacije sjenčara ona se obavlja u konstruktoru `Shader::Shader(const char* vertexPath, const char* fragmentPath, const char* geometryPath = nulltpr)`. Sadržaj samog konstruktora sastoji se od učitavanja i kompajliranja koda te ispisa pronađenih grešaka unutar samog sjenčara. Sami sjenčari (vertex, fragment i geometry shader ako je naveden kao parametar metode) bit će objašnjeni u idućim poglavljima.

Što se tiče metoda kao što su `setVec4()` i `setMat4()` (Programski kod 3.3), one su vrlo kratke i jednostavne te se sve temelje na sličnom principu.

```
void Shader::setMat4(const std::string &name, const glm::mat4 &mat) const
    {
        glUniformMatrix4fv(glGetUniformLocation(ID,   name.c_str()),   1,
                           GL_FALSE, &mat[0][0]);
    }
```

Kod 3.3 – Postavljanje vrijednosti matrice u sjenčar

Metoda prima argumente `std::string &name` kojim se specificira naziv varijable u sjenčaru čiju vrijednost postavljamo i `glm::mat4 &mat` u čemu su spremljene vrijednosti matrice koju postavljamo.

Koristeći ove parametre i funkciju za pronalazak adrese uniforme varijable unutar sjenčara `glGetUniformLocation()`, sada lagano postavljamo vrijednost uniformne varijable na danu matricu pomoću funkcije `glUniformMatrix4fv()`.

3.4. Vizualizacija smjera kretanja

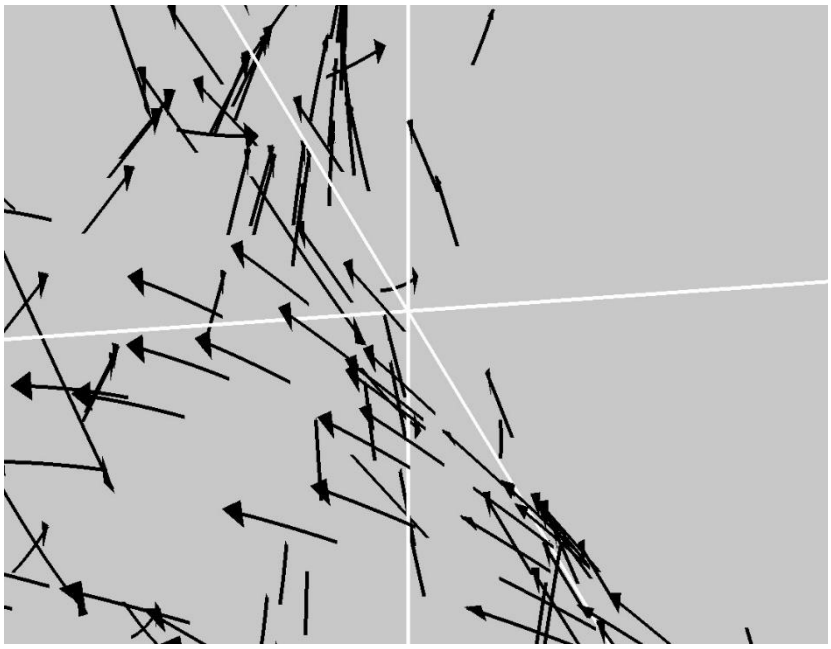
U svrhe lakšeg vizualiziranja trenutnog smjera kretanja svake od točaka, na vrh svake nadodana je strelica koji pokazuje upravo taj smjer. Pogledajmo kako se ovakvi trokuti konstruiraju.

```
glm::vec3 d = triangle[i*3].update(chosen_equation, dt,
                                   equation_constant, delta);

glm::vec3 perpendicular = glm::cross(
       glm::vec3(triangle[i*3].x,triangle[i*3].y,triangle[i*3].z),
       glm::vec3(triangle[i*3].x+0.01f,triangle[i*3].y,triangle[i*3].z));

glm::vec3 unit_p = glm::normalize(perpendicular)/50.f;
glm::vec3 unit_d = glm::normalize(d)/20.f;

//2nd triangle point
triangle[i*3+1].x = triangle[i*3].x - unit_d.x + unit_p.x;
triangle[i*3+1].y = triangle[i*3].y - unit_d.y + unit_p.y;
triangle[i*3+1].z = triangle[i*3].z - unit_d.z + unit_p.z;

//3rd triangle point
triangle[i*3+2].x = triangle[i*3].x - unit_d.x - unit_p.x;
triangle[i*3+2].y = triangle[i*3].y - unit_d.y - unit_p.y;
triangle[i*3+2].z = triangle[i*3].z - unit_d.z - unit_p.z;
```

Kod 3.4 – Kalkulacija za vizualizaciju trenutnog smjera kretanja

Nakon što smo prvu točku trokuta osvježili koristeći istu funkciju kao i za osvježavanje svake od točaka, u varijablu `d` spremamo izračunati pomak za buduće izračune.
Kako bismo odredili preostale dvije točke potrebno je izračunati vektor okomit na smjer kretanja. To možemo dobiti kalkulacijom vektorskog produkta između vektora smjera kretanja i vektora smjera kretanja s malim nekolinearnim offsetom (u našem slučaju taj offset je `0.01f` po x osi). Ovakav izračun je moguć jer nam egzaktan kut trenutne rotacije same strelice nije bitan.

Normaliziramo vektore i dijelimo ih s odgovarajućim konstantama kako bi dobili strelice željene veličine. Sada je moguće konstruirati donje dvije točke strelice oduzimajući od njih normalizirani vektor pomaka, te zbrajajući / oduzimajući izračunati normalizirani okomiti vektor. Dobiveni rezultat je slijedeći:



Slika 3.4 - Vizualizacija smjera kretanja

3.5. Iscrtavanje dijela putanje točaka

Na slici 3.4 moguće je vidjeti da se osim točaka i njihovih smjerova kretanja iza njih iscrtavaju i već prođeni dijelovi putanje svake od točaka. Ovo dio je realiziran koristeći dodatan vektor u kojeg spremamo već prije izračunate točke klase `Point`. Potpun kod u nastavku.

```
trail[i].push_back((Point){x, y, z});
if(trail[i].size() > 50){
   trail[i].erase(trail[i].begin());
}
glBindVertexArray(vao3);
glBindBuffer(GL_ARRAY_BUFFER, vbo3);
```

```
glBufferSubData(GL_ARRAY_BUFFER,                                    0,
                trail[i].size() * sizeof(Point), trail[i].data());
for(int k =0;k<(int)trail[i].size()-1;k++){
    shader.setInt("trail_index", k+1);
    glDrawArrays(GL_LINES, k , 2);
 }
```

Kod 3.5 Spremanje i iscrtavanje točaka putanje atraktora

Nakon spremanja samih točaka u vektor provjeravamo je li veličina tog vektora veća od 50, te ako je brišemo točku koja je prva dodana u vektor kako bi limitirali duljinu iscrtanog `dijela` putanje. Preostali kod koristi OpenGL pozive za iscrtavanje samog vektora, te još u sjenčar za svaki segment putanje postavlja vrijednost uniformne varijable `trail_index`. Nešto više o ovom u idućem poglavlju.

3.6. Sjenčari

Sjenčare u svakom grafičkom programu koristimo za željene transformacije, bojanja, razne efekte itd. Pogledajmo za što su u ovoj vizualizaciji zaslužni tzv. sjenčari vrhova i fragmenata (vertex / fragment shader).

3.6.1. Sjenčar vrhova

Sjenčar vrhova iz memorije učitava pozicije svake od točaka i nad njima provodi odgovarajuće transformacije koristeći matrice pogleda i projekcije (Kod 3.6.1) čiji je izračun objašnjen u poglavlju 3.2.

```
#version 330 core
layout (location = 0) in vec3 pos;

uniform mat4 view;
uniform mat4 projection;

void main(){
```

```
    gl_Position =  projection * view * vec4(pos, 1.0f) ;
}
```

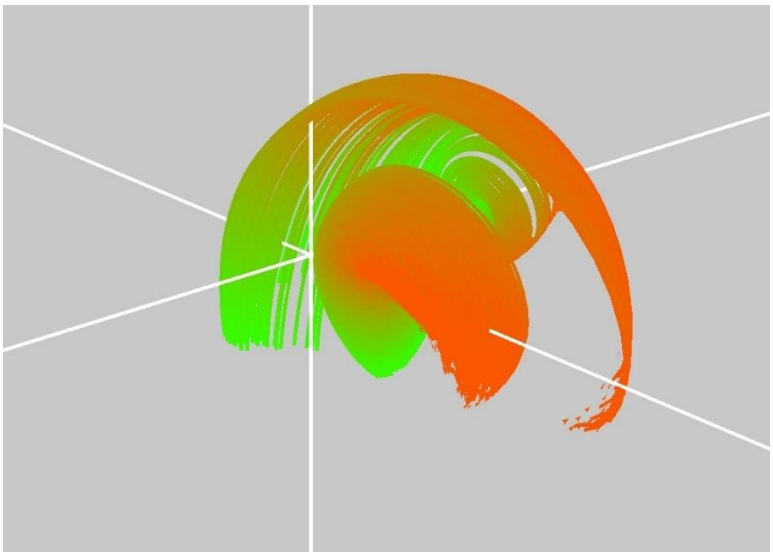
Kod 3.6.1 Sjenčar vrhova

3.6.2. Sjenčar fragmenata

Sjenčar fragmenata između ostaloga zaslužan je za pravilnu interpolaciju između početne i krajnje boje krivulje koja se dobiva iscrtavanjem dijela putanje (Slika 3.6). Samo bojanje ove krivulje je realizirano koristeći OpenGL primitivu GL_LINES između svake dvije susjedne točke (Kod 3.5) čije su vrijednosti boja određene interpolacijom boja zadanih kroz korisničko sučelje. Interpolacija boja je omogućena koristeći prije spomenutu uniformnu varijablu `trail_index` koja služi kao faktor interpolacije i glsl funkciju `mix()` (čije je korištenje prikazao u kodu 3.6.2).

```
#version 330 core
uniform bool coord_sys, coord_net;
uniform vec4 start_color, end_color;
uniform int trail_index;

out vec4 final_color;

void main(){
    if(coord_net)
        final_color = vec4(0.4f, 0.4f, 0.4f, 0.1f);
    else if(coord_sys)
        final_color  = vec4(1.0f, 1.0f, 1.0f, 1.0f);
    else
        final_color = mix(end_color, start_color, trail_index/50.f);
}
```

Kod 3.6.2 Sjenčar fragmenata



Slika 3.6 Vizualizacija Sprott kaotičnog atraktora

3.7. Korisničko sučelje

Korisničko sučelje (Slika 3.7) aplikacije sastoji se od 3 prozora: Svojstva, Atraktori i Konstante.

U prozoru Svojstva prikazana su neka od bitnijih svojstava aplikacije, a ona su sljedeća:

- Timestamp – klizač za postavljanje brzine vizualizacije
- Point number – klizač za odabir broja vizualiziranih točaka atraktora
- Head & Tail color – odabrane boje početka i kraja krivulje u formatu RGBA
- Camera Theta angle – indikator horizontalnog kuta kamere s obzirom na ishodište
- Camera Y axis – indikator vertikalne pozicije kamere (u intervalu [0-1])
- Show coordinate system net – tipka za prikaz koordinatnog sustava
- Reset simulation – tipka za vraćanje vizualizacije u inicijalno stanje

Što se tiče prozora Atraktori on se sastoji od liste 10 atraktora koje korisnik proizvoljno odabire.

U prozoru Konstante moguće je pronaći listu klizača za odabir vrijednosti konstanta određenog atraktora.

Slika 3.7 - Korisničko sučelje

4. Rezultati

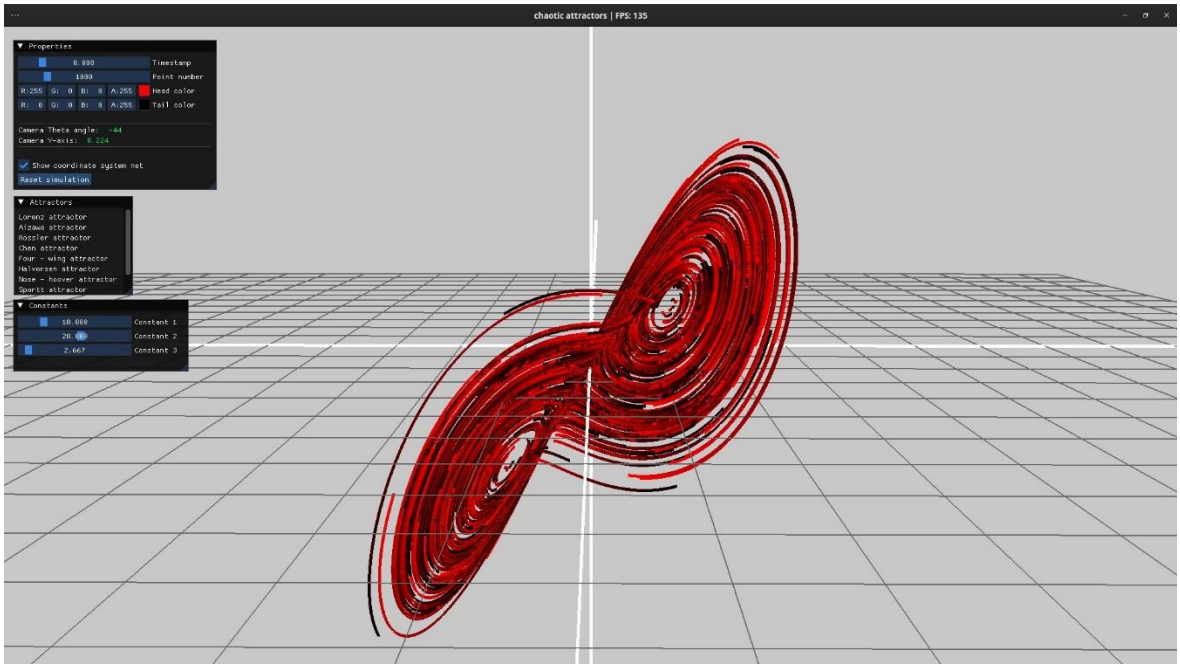
U ovom poglavlju demonstrirat ćemo dobivene rezultate za neke od postavljenih parametara vizualizacije te će se diskutirati o mogućim nadogradnjama/poboljšanjima aplikacije.

U svrhe demonstracije rezultata korištena je sljedeća oprema:

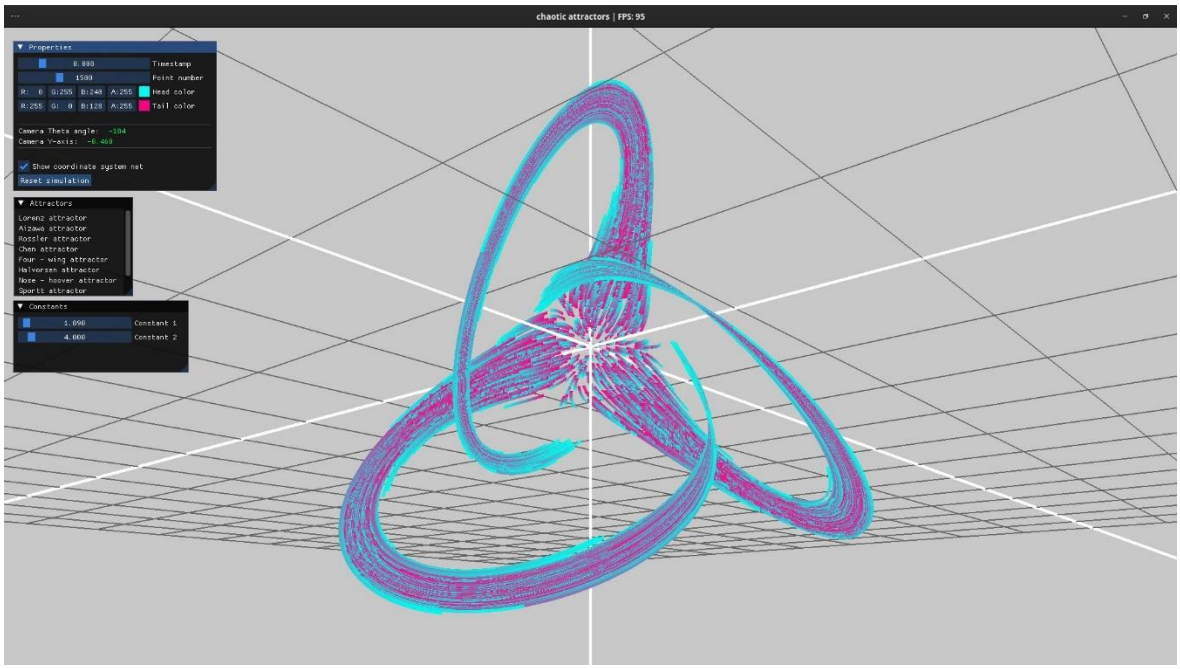
- CPU: AMD Ryzen 5 2600 @ 3.400 Ghz
- RAM: 16.0 GB Dual-Channel @ 1596 Mhz
- GPU: 4095 MB NVIDIA GeForce GTX 1050 Ti
- OS: Linux EndeavourOS (6.8.9-arch1-2)

4.1. Odnos FPS-a i broja vizualiziranih točaka

Jedini parametar kojeg korisnik može postaviti unutar aplikacije, a da primjetno utječe na njene performanse, je broj vizualiziranih točaka. Za očekivati je da će se vizualizacija atraktora usporiti što više točaka želimo iscrtati. Pogledajmo kakav je točno odnos FPS-a i odabranog broja vizualiziranih točaka (Slike 4.1.1 i 4.1.2).

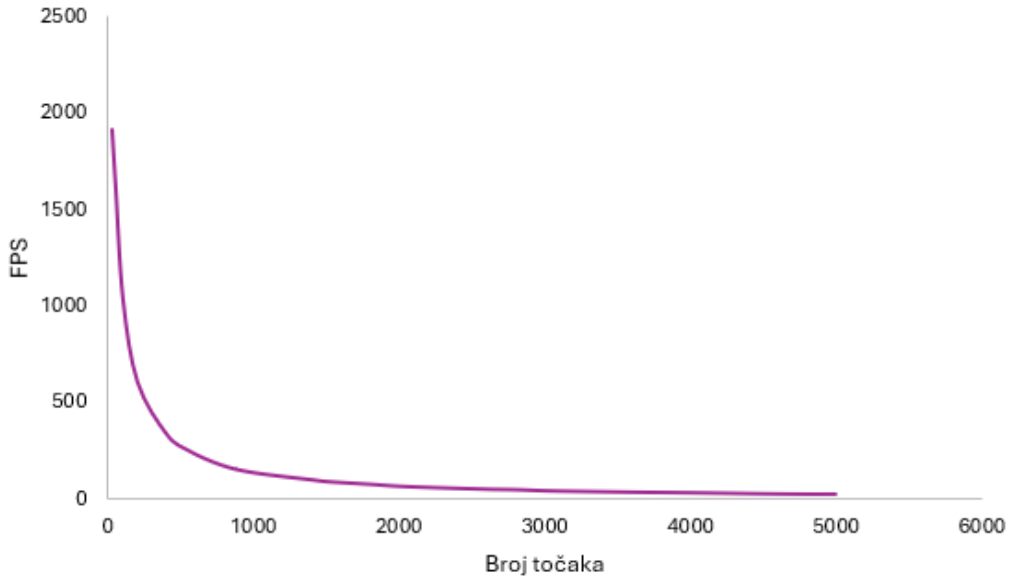


Slika 4.1.1 – Vizualizacija atraktora s brojem postavljenih točaka na 1000



Slika 4.1.2 – Vizualizacija atraktora s brojem postavljenih točaka na 1500

Kao što je i očekivano, vidljiv je pad FPS-a pri postavljanju većeg broja vizualiziranih točaka. U svrhe boljeg shvaćanja ovog odnosa napravimo graf s većim brojem uzoraka (Slika 4.1.3).



Slika 4.1.3 – Graf odnosa FPS-a i broja vizualiziranih točaka

4.2. Problemi i nedostatci vizualizacije

Jedan od problema vizualizacije koji se javlja kod nekih atraktora je ta da kada se timestamp postavi na veću vrijednost preko klizača, točke s vremenom skrenu s putanje i „odlete“ u beskonačnost. Naime, osvježavanjem pozicije točke s prevelikom timestamp vrijednošću točka više ne leži na putanji na kojoj bi ležala kada bi taj timestamp bio manji, te sama jednadžba atraktora postaje nevaljana. Za ovaj problem jedno od rješenja bi bilo da se za svaki od atraktora podesi odgovarajući interval vrijednosti na klizaču kako se ne bi pravila prevelika numerička greška pri osvježavanju pozicije. Osim ovakvog pristupa moguće bi bilo koristiti nešto složenije tehnike računanja diferencijalnih jednadžbi kako bi greška aproksimacije funkcije bila što manja. No kada bi se one implementirale postavlja se pitanje kako bi se to odrazilo na performanse vizualizacije.

Što se tiče nedostataka, u aplikaciji je pokriveno sve što je inicijalno bio i plan pa i više. Jedan od funkcionalnosti (nedostataka) koja bi se mogla dodati u budućnosti je kretanje kamere mišom, čime bi se mogla omogućiti veća preciznost pri pogledu na atraktor iz raznih smjerova.

4.3. Moguće nadogradnje i poboljšanja

Jedna od mogućih nadogradnja vizualizacije je pružanje korisniku veće razine slobode implementirajući dodatne efekte s promjenjivim parametrima koristeći neke od raznih tehnika sjenčanja. Također, unutar aplikacije bilo bi moguće implementirati uzimanje slika zaslona pritiskom na neku od tipaka na tipkovnici.

S optimizacijske strane aplikacije bi se mogli implementirati nešto veći pothvati koristeći API-eve i biblioteke namijenjene za paralelizaciju aplikacija kao što su OpenMP i CUDA. Naravno, ovakve implementacije zahtijevaju ulaganje dodatnog vremena pošto nisu trivijalne, no nekada u budućnosti bi bilo zanimljivo vidjeti koliko se zapravo aplikacija može optimizirati koristeći ovakve tehnologije.

Zaključak

Vizualizacija kaotičnih atraktora predstavlja ključan korak u razumijevanju i interpretaciji složenih dinamičkih sustava. Kroz ovaj rad, pokazan je jedan od načina za grafički prikaz kaotičnih atraktora, demonstrirajući njihovu sposobnost da otkriju skriveni red u naizgled nasumičnim ponašanjima.

Kroz vizualizaciju, koncepti kao što su Lorenzov, Rösslerov i Sprottov atraktor postaju intuitivniji, omogućujući dublje razumijevanje dinamike sustava. Vizualni prikazi ovih atraktora ne samo da olakšavaju identifikaciju osnovnih obrazaca i struktura, već također pomažu u otkrivanju osjetljivosti na početne uvjete i inherentne nepredvidivosti koje su karakteristične za kaotične sustave.

Zaključno, vizualizacija kaotičnih atraktora igra vitalnu ulogu u demistifikaciji kaosa. Ona omogućava bolje razumijevanje i analizu složenih sustava, otkrivajući ljepotu i strukturu skrivenih u kaosu. Ovo razumijevanje ne samo da obogaćuje naše teoretsko znanje, već također ima praktične implikacije koje mogu unaprijediti tehnologiju, znanost i obrazovanje. Kao takva, vizualizacija ostaje neophodan alat u kontinuiranom istraživanju i primjeni kaotičnih dinamičkih sustava.

Literatura

[1] Juan Carlos Ponce Campuzano, Strange Attractors, (2018, kolovoz). Poveznica: https://www.dynamicmath.xyz/strange-attractors/ ; pristupljeno 15.02.2024

[2] Josh Kastorf, The Lorenz Attractor Explained, (2019, siječanj). Poveznica: https://www.youtube.com/watch?v=VjP90rwpBwU ; pristupljeno: 12.01.2024

[3] Frank Force, Lorenz Attractor, (2019, kolovoz). Poveznica: https://frankforce.com/wp-content/uploads/2019/08/lorenz-1-1.png ; pristupljeno 25.05.2024

[4] Joey de Vries, Learn OpenGL, (2014, lipanj). Poveznica: https://learnopengl.com/ ; pristupljeno: 20.04.2024

[5] Unknown, OpenGL Mathematics, (2014, Listopad). Poveznica:

https://glm.g-truc.net/0.9.2/api/a00245.html ; pristupljeno:20.04.2024.

[6] Unknown, Dear ImGui, (2014, kolovoz). Poveznica: https://github.com/ocornut/imgui ;

pristupljeno: 29.04.2024.

[7] Unknown, GLFW, (2022, srpanj). Poveznica: https://www.glfw.org/ ; pristupljeno:

20.04.2024

[8] Unknown, Euler's numerical method, (2019, siječanj). Poveznica: https://www.uah.edu/images/people/faculty/howellkb/DEText-Ch9.pdf ; pristupljeno: 05.05.2024

Sažetak

Vizualizacija atraktora

Pojavili se oni u meteorologiji, dinamici fluida, neuroznanosti ili samo u računalu prikazani pomoću bitova i bajtova, neporecivo je da je sama pojava kaotičnih atraktora misteriozna činjenica. Kroz ovaj rad demonstrirani su neki od osnovnih pojmova i svojstava kaotičnih atraktora, kao i jednadžbe koje ih opisuju. Koristeći OpenGL i već spomenute biblioteke vizualizirano je 10 kaotičnih atraktora, među kojima je i dobro znani Lorenzov atraktor. U ovome radu priložene su neke od slika vizualizacije, no smatram kako same slike pružaju uvid u samo fragment egzotičnosti iste. Iz ovoga razloga projekt je stavljen na GitHub kako bi možda i sami mogli pokrenuti ovu vizualizaciju i sagledati dinamiku atraktora iz Vama željene perspektive.

Ključne riječi: vizualizacija atraktora, kaotični atraktori, Lorenzov atraktor, OpenGL

Summary

Visualization of attractors

Whether they appear in meteorology, fluid dynamics, neuroscience, or simply in a computer using bits and bytes, it is undeniable that the very phenomenon of chaotic attractors is a mysterious fact. This paper demonstrates some of the basic concepts and properties of chaotic attractors, as well as the equations that describe them. Using OpenGL and already mentioned libraries, 10 chaotic attractors have been visualized, including the well-known Lorenz attractor. Some of the visualization images are included in this paper, but I believe that the images themselves provide only a glimpse of their exotic nature. For this reason, the project has been uploaded to GitHub so that you could maybe run the visualization yourself and observe the dynamics of the attractors from your desired perspective.

Keywords: visualization of attractors, chaotic attractors, Lorenz attractor, OpenGL

Privitak

Projekt je moguće pronaći na poveznici:

(**https://github.com/drito256/chaotic_attractors/**)