UNIVERSITATEA DIN BUCUREŞTI

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

Lucrare de licenţă

GENERATOR DE FULGI DE ZĂPADĂ UNICI FOLOSIND OPENGL

Absolvent

STRÎMBEANU LUANA DIANA

Coordonator ştiinţific

Conf. Dr. STUPARIU MIHAI SORIN

Bucureşti, septembrie 2022

Rezumat

Grafica pe calculator este un domeniu care observă o dezvoltare rapidă, atât pe partea software cât și pe cea hardware, cu precădere în industriile gaming și CGI care folosesc grafici cu nivel ridicat de detaliu. Acest fapt duce la o cerere cât mai mare de optimizare, pe cât posibil, a utilizării memoriei folosite. Soluția ar putea fi generarea procedurală a cât mai multor asset-uri noi din asset-uri deja existente. Această practică este cel mai des întâlnită în generarea peisagistică (teren, nori, copaci, frunze etc.). Aplicația pe care o propune această lucrare este un generator procedural de fulgi unici în OpenGL utilizând primitive grafice cât mai disjuncte, prin desenarea tuturor componentelor direct din cod pe placa grafică, fără utilizarea asset-urilor externe. Rezultatul aplicației ar trebui să fie un fulg de zăpadă unic desenat în fereastră. Forma fulgului de zăpadă trebuie să fie diferită la fiecare execuție a programului. S-a ales fulgul de zăpadă deoarece se încadrează în generarea peisagistică și are o geometrie ușor de înțeles și replicat.

Abstract

Computer graphics is a field seeing rapid development, both on the software and hardware side, especially in the gaming and CGI industries that use highly detailed graphics. This leads to a greater demand to optimize, as much as possible, the memory usage. The solution could be the procedural generation of as many new assets as possible from already existing assets. This practice is most common in landscape generation (terrain, clouds, trees, leaves, etc.). The application that this paper proposes is a unique procedural snowflake generator with OpenGL using graphic primitives as distinctive as possible, by drawing all components directly from inside the code onto the graphics card, without using external assets. The result of the application should be a single snowflake drawn in the window. The shape of the snowflake must be different at each execution of the program. The snowflake was chosen because it fits into landscape generation and has a geometry that is easy to understand and replicate.

Cuprins

Capitolul 1

Introducere

Am decis realizarea unei lucrări practice în domeniul graficii pe calculator cu scopul explorării subdomeniului graficii generate procedural de calculator.

Tema lucrării este generarea procedurală a fulgilor de zăpadă unici folosind API-ul OpenGL, utilizând primitive grafice cât mai disjuncte, prin desenarea tuturor componentelor direct din cod pe placa grafică, fără utilizarea asset-urilor externe. Scopul programului este să deseneze mereu alt fulg de zăpadă când este executat.

Motivația lucrării a provenit din faptul că generarea procedurală este un subiect tot mai de interes în Grafica pe Calculator, cu precădere în ce privește aplicațiile de simulare, cele mai populare astfel de medii fiind jocurile video și Realitatea Virtuală.

Industria jocurilor video este în dezvoltare continuă, fiind una dintre cele mai importante din entertainment în anii recenți, întrecând chiar și industriile muzicii și a filmelor[1]. Industria Realității Virtuale a primit un mare avânt odată cu rebrand-uirea companiei Facebook în Meta.

De aceea, și așteptările cresc. Jocurile devin din ce în ce mai complexe cu fiecare an care trece, dezvoltatorii fiind pionierii tehnologiilor noi pentru a oferi experiențe cât mai unice. Poate una dintre componentele unui joc care suferă schimbările cele mai mari generație de generație este grafica. Diferențele vizuale dintre jocurile de acum zece sau chiar cinci ani față de jocurile de acum nu sunt numai neneglijabile, dar chiar substanțiale.

Fig. 1.1 – Elder Scrolls V (2011), sursa: [10]

Fig 1.2 – The Witcher 3 (2015), sursa: [11]

Totodată, trăim în epoca ecranelor 4K HD care pot reda un nivel de detaliu al imaginilor foarte mare. Un procent de 75% dintre pasionații de jocuri pe calculator spun că grafica unui joc le influențează deciziile de cumpărare[2]. Acest trend al graficii ultrarealiste cu nivel mare de detaliu are însă și un preț destul de mare, și anume acela al memoriei utilizate. Dacă în 2010 jocurile ocupau doar 2-5GB din memoria fizică, acum au ajuns la a ocupa și la de o sută de ori mai mult de atât.

De aceea, dezvoltatorii de jocuri aleg să reutilizeze multe dintre asset-uri, însă această practică ajunge treptat să creeze o experiență de monotonie și previzibilitate.

Soluția apărută a fost grafica generată procedural. Avantajele generării procedurale includ dimensiuni mai mici ale fișierelor, cantități mai mari de conținut și caracter aleatoriu pentru un joc mai puțin anticipabil.

Generarea procedurală se folosește în jocuri și aplicații de simulare în preponderent în ceea ce privește generarea mediului înconjurător. Elementele de bază ale peisajelor create de generatoarele de peisaje includ terenul(Fig 1.3), apa, frunzișul și norii.

Fig. 1.3 – Teren generat procedural, sursa: [12]

Cel mai popular joc la ora actuală este Minecraft [3] – un joc publicat în 2011 care a trecut testul timpului și și-a menținut relevanța timp de un deceniu. Minecraft este un joc 3D open-world format din „blocks” a cărui lume este complet generată procedural (Fig. 1.4), fiind în esență infinită, dar bazându-se însă pe o multitudine de reguli care oferă unicitate environment-ului. Jucătorii pot manipula aceste blocks și a crea la rândul lor environments (Fig 1.5).

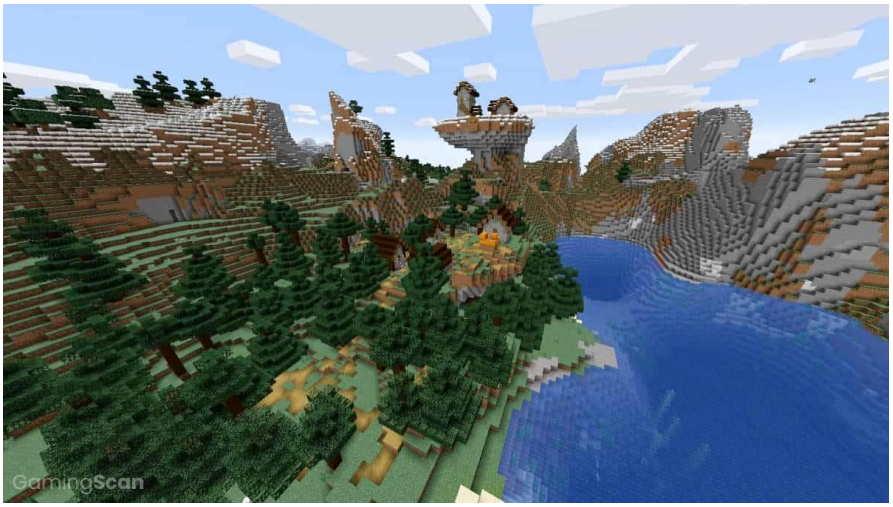

Fig. 1.4 – Lume generată
procedural în Minecraft, sursa: [13]

Fig. 1.5 – Environment creat de
un jucător Minecraft, sursa: [14]

Am vrut să experimentez la rândul meu cu generarea procedurală a graficii. Am ales generarea procedurală a fulgilor de zăpadă deoarece se încadrează în generarea de peisaj și nu este ceva ce am întâlnit implementat des în practică, deci poate fi explorat mai mult, în special în contextul actual când se pune accent pe „detaliu”. Totodată, aceștia au o geometrie destul de bine stabilită, cu reguli destul de clare.

Capitolul 2

Preliminarii

2.1 Tipuri de fulgi de zăpadă în natură

Vom folosi o clasificare[4] făcută de Dr. Kenneth G. Libbrecht, profesor de fizică la California Institute of Technology.

1. Prisme Simple

O prismă hexagonală este forma geometrică cea mai elementară a unui fulg de zăpadă. Prismele simple sunt de obicei atât de mici încât abia pot fi văzute cu ochiul liber. Fațetele fulgului sunt rareori perfect plate, fiind de obicei decorate cu diverse șanțuri, creste sau alte caracteristici.

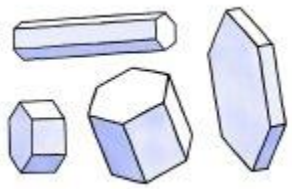

Fig. 2.1 – forme teoretice diverse pe care le pot avea prismele simple, sursa: [4]

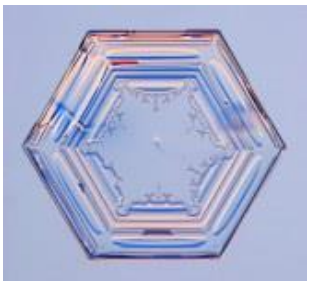

Fig. 2.2 – prismă simplă în natură, sursa: [4]

2. Plăci Stelare

Acești fulgi de zăpadă comuni reprezintă cristale subțiri, asemănătoare plăcilor, cu șase brațe largi care formează o formă de stea. Fețele lor sunt adesea decorate cu însemnări elaborate și simetrice.

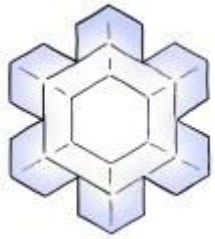

Fig. 2.3 – forma geometrică, sursa: [4]

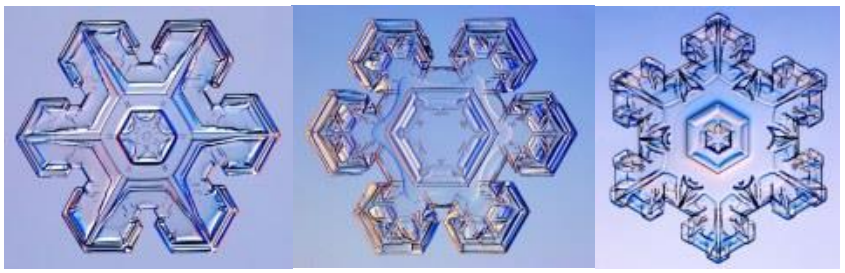

Fig. 2.4 – plăci stelare în natură, sursa: [4]

3. Plăci sectorizate

Plăcile stelare prezintă adesea creste distinctive care indică colțurile dintre fațetele prismelor adiacente. Când aceste creste sunt deosebit de proeminente, cristalele sunt numite plăci sectorizate.

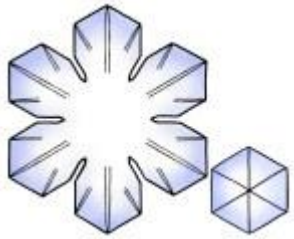

Fig. 2.5 – forma geometrică, sursa: [4]

Fig. 2.6 – plăci sectorizate în natură, sursa: [4]

4. Dendrite Selare

După cum le zice și numele, dendritele stelare sunt cristale de zăpadă asemănătoare plăcilor, care au ramuri și ramificații laterale. Acestea sunt cristale destul de mari, de obicei de 2-4 mm în diametru, care sunt ușor de văzut cu ochiul liber.

Fig. 2.7 – placă sectorizată în natură, sursa: [4]

5. Dendrite Ferigă

Când ramurile cristalelor stelare au multe ramuri laterale, le numim dendrite stelare ferigă. Acestea sunt cele mai mari cristale de zăpadă, adesea căzând pe pământ cu diametre de 5 mm sau mai mult.

Fig. 2.8 – forma geometrică, sursa: [4]

Fig. 2.9 – dendrită ferigă în natură, sursa: [4]

6. Coloane

Coloanele hexagonale se formează adesea cu regiuni goale conice la capete. Secțiunile goale sunt simetrice.

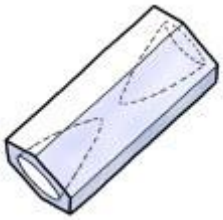

Fig. 2.10 – forma geometrică, sursa: [4]

Fig. 2.11 – fulg de tip coloană în natură, sursa: [4]

7. Ace

Acele sunt cristale de gheață subțiri, columnare, care cresc atunci când temperatura este în jur de -5 C.

Fig. 2.12 – forma geometrică, sursa: [4]

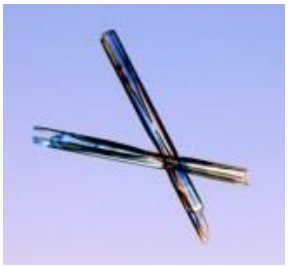

Fig. 2.13 – fulg de tip ac în natură, sursa: [4]

8. Combinări dintre formele de bază prezentate mai sus

Acestea includ: coloane care au la fiecare capăt câte o placă stelară, plăci duble, plăci suprapuse, cristale triangulare, fulgi cu douăsprezece ramuri, rozete, dendrite spațiale.

Fig. 2.14 – coloane acoperite, sursa: [4]

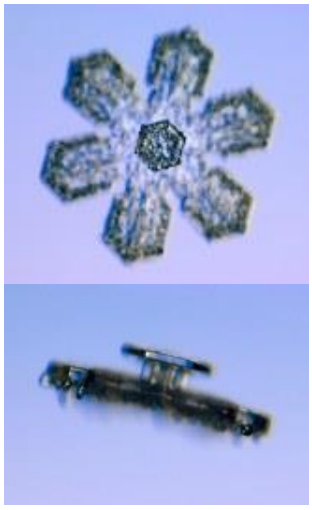

Fig. 2.15 – plăci duble, sursa: [4]

Fig. 2.16 – plăci suprapuse, sursa: [4]

Fig. 2.17 – cristal triangular, sursa: [4]

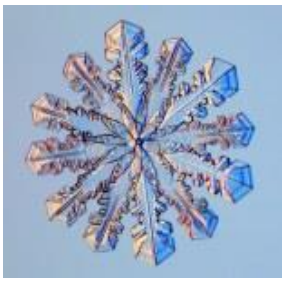

Fig. 2.18 – fulgi cu 12 ramuri, sursa: [4]

Fig. 2.19 – rozetă, sursa: [4]

Fig. 2.20 – dendrită spațială, sursa: [4]

Pentru aplicația pe care urmează să o creăm vom construi doar primele cinci forme prezentate deoarece sunt regulate și au relevanță estetică.

2.2 Tehnologii folosite

Pentru programul de bază al aplicației se vor folosi elemente din codul predat la Laboratorul 8 al cursului de Grafică pe Calculator.

2.2.1 OpenGL

OpenGL („Open Graphics Library”) este o interfață software pentru hardware-ul grafic. Interfața constă dintr-un set de câteva sute de proceduri și funcții care permit unui programator să specifice obiectele și operațiunile implicate în producerea de imagini grafice de înaltă calitate, în special imagini color ale obiectelor tridimensionale. [6]

De ce OpenGL?

Am ales folosirea OpenGL deoarece este un API cross-platform. spre deosebire de alternative ca DirectX care se poate folosi doar pe Windows, sau Metal care este dedicat ecosistemului Apple. Totodată, OpenGL este mai ușor de înțeles, învățat și utilizat decât DirectX, iar ca performanță diferența dintre cele două API-uri este neglijabilă.

Cum funcționează OpenGL?

Deoarece OpenGL este doar un API, implementarea librăriilor rămâne în sarcina dezvoltatorilor hardware. Pentru ca o placă video să aibă suport pentru OpenGL, aceasta trebuie să aibă un framebuffer. OpenGL va face randarea în acest framebuffer.

OpenGL este definit ca o „state machine”. Diferitele apeluri API schimbă starea OpenGL, interogează o parte din acea stare sau determină OpenGL să folosească starea curentă pentru a reda ceva.

Obiectele sunt întotdeauna containere pentru state. Fiecare tip particular de obiect este definit de starea particulară pe care o conține. Un obiect OpenGL este o modalitate de a încapsula un anumit grup de stări și de a le schimba pe toate într-un singur apel de funcție.

Obiectele OpenGL sunt:

1. Obiecte Buffer: Vertex Buffer (ține informații despre vârfuri și se ocupă de randarea acestora), Pixel Buffer (se ocupă de transferul pixelilor de la user la OpenGL și invers), Shader Storage Buffer (stochează și obține date de la GLSL – OpenGL Shading Language), Uniform Buffer (stochează date uniforme ale unui program pentru shaders).
2. Vertex Array Objects – un Vertex Array Object (VAO) este un obiect OpenGL care stochează tot state-ul necesar pentru a furniza date despre vârfuri. Acesta stochează

formatul datelor de vârf, precum și al obiectelor Buffer care furnizează array-uri de date despre vârfuri.

3. Texturi
4. Obiecte Framebuffer
5. Obiecte pentru operații asincrone
6. Obiecte ce sincronizează activitatea dintre GPU și aplicație
7. Obiecte pentru program și shaders

În aplicația noastră ne vom folosi de obiecte Buffer și Vertex Array.

Pipeline-ul de randare OpenGL este inițiat atunci când se efectuează o operație de randare. Operațiunile de randare necesită prezența unui Vertex Array Object și a unui Program Object care furnizează shader-ele.

Odată inițiat, pipline-ul funcționează în următoarea ordine:

1. Procesarea vârfurilor:
   1.1. Fiecare vârf extras din Vertex Array va fi preluat de un Vertex Shader. Fiecare vârf din stream este procesat la rândul său într-un vârf de output.
   1.2. Etape opționale de teselare a Primitivelor. Teselarea reprezintă împărțirea seturilor de date de poligoane care prezintă obiecte dintr-o scenă în structuri adecvate pentru randare. În special pentru randarea în timp real, datele sunt teselate în triunghiuri.
   1.3. Procesarea opțională a Geometry Shader. Geometry Shader primește o singură primitivă și întoarce zero sau mai multe primitive.

2. Vertex Post-Processing
   2.1. Transform Feedback-ul. Transform Feedback este procesul de captare a Primitivelor generate de etapele de procesare a vârfurilor, înregistrând datele de la acele primitive în Buffer Objects. Acest lucru permite păstrarea rendering state post-transformare a unui obiect și retrimiterea acestor date de mai multe ori.
   2.2. Asamblarea Primitivelor. Aceasta este etapa în care Primitivele sunt împărțite într-o secvență de primitive de bază individuale. După unele procesări minore, acestea sunt transmise la rasterizare pentru a fi randate.
   2.3. Clipping-ul primitivelor, perspective divide, transformarea viewport-ului relativ la spațiul ferestrei.

3. Rasterizarea. Acesta este procesul prin care fiecare Primitivă individuală este impărțită în elemente discrete numite Fragmente. Aceasta este etapa în care vârfurile sunt transformate în pixeli.

4. Procesări realizate de Fragment Shader (ex.: procesarea culorilor, depth value)

5. Per-Sample Processing, adică procesul în care fragmentele procesate de Fragment Shader sunt transmise mai multor Buffers incluzând, dar fără a se limita la: Scissor Test (verificarea dacă Fragmentele se află într-un Cropping Region), Stencil Test (acționează ca Scissor Test, doar că în loc de o regiune dreptunghiulară avem orice fel de regiune neregulată. Se folosește pentru umbre), Depth Test (verificarea dacă un fragment apare sau nu pe ecran ex.: dacă o primitivă se află în spatele altei primitive, aceasta nu este desenată), Amestecare (combinarea de culori), Operații Logice, Măști. [5] [6]

2.2.2 Librăria FreeGLUT/GLUT

GLUT se ocupă de toate sarcinile specifice sistemului necesare pentru crearea ferestrelor, inițializarea contextelor OpenGL și gestionarea evenimentelor de intrare (ex.: mouse/tastatură), pentru a spori portabilitatea programelor OpenGL. [7]

GLUT a fost scrisă de Mark J. Kilgard, autorul *OpenGL Programming for the X Window System and The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*.

Deoarece nu mai este oferit suport pentru GLUT în prezent iar librăria a devenit deprecată, a apărut ca alternativă biblioteca FreeGLUT care intenționează să înlocuiască complet GLUT și are doar diferențe minore față de acesta. FreeGLUT a fost dezvoltată de Paweł W. Olszta. [7]

În aplicația pe care o vom dezvolta, vom folosi această librărie pentru a crea o fereastra și a procesa evenimente de mouse/tastatură.

2.2.3 Librăria GLEW

Biblioteca OpenGL Extension Wrangler (GLEW) este o bibliotecă open-source și cross-platform de încărcare a extensiilor C/C++. GLEW oferă mecanisme eficiente run-time pentru a determina ce extensii OpenGL sunt acceptate pe platforma țintă. GLEW a fost dezvoltată de Milan Ikits and Marcelo Magallon. [8]

2.2.4 Librăria GLM

OpenGL Mathematics (GLM) este o bibliotecă C++ header-only de matematică pentru software de grafică bazat pe specificațiile OpenGL Shading Language (GLSL).

GLM oferă clase și funcții concepute și implementate cu aceleași convenții de denumire și funcționalități ca GLSL. Acest proiect nu se limitează la caracteristicile GLSL. Mai degrabă este un sistem de extensie, bazat pe convențiile de extensie GLSL, și oferă capabilități extinse: transformări de matrice, cuaternioni, numere aleatorii, zgomot etc.

Această bibliotecă funcționează cu OpenGL, dar asigură și interoperabilitatea cu alte biblioteci third-party și SDK. Este un bun candidat pentru randarea software (raytracing / rasterizare), procesarea imaginilor, simulări fizice și orice context de dezvoltare care necesită o bibliotecă de matematică simplă și convenabilă.

GLM este scris în C++98, dar poate utiliza paradigme de C++11 atunci când este acceptat de compilator. Este o bibliotecă independentă de platformă, fără dependințe.[9]

Capitolul 3
Generatorul de fulgi de zăpadă

Aplicația va consta într-o fereastră creată cu biblioteca FreeGLUT, în interior căreia vom desena forma unui fulg de zăpadă utilizând pipeline-ul OpenGL. Vom crea primele cinci tipuri de fulgi de zăpadă enumerate în secțiunea de Preliminarii și anume: prisme simple, plăci stelare, plăci sectorizate, dendrite stelare și dendrite ferigă.

3.1 Instanțiere fereastră

Pentru a instanția o fereastră OpenGL vom folosi următoarele următorul cod generic preluat din documentația FreeGLUT [7]:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
glutInitWindowPosition(100, 100);
glutInitWindowSize(1200, 900);
glutCreateWindow("Generare fulg de zapada unic");
glewInit();
Initialize();
glutDisplayFunc(RenderFunction);
glutIdleFunc(RenderFunction);
glutKeyboardFunc(processNormalKeys);
glutSpecialFunc(processSpecialKeys);
glutCloseFunc(Cleanup);
glutMainLoop();
```

Funcțiile `glutInit()` și `glewInit()` se folosesc pentru a instanția librăriile FreeGLUT, respectiv GLEW.

Funcția `glutInitDisplayMode()` se folosește pentru a seta modul de afișare inițial. Alegem ca parametri `GLUT_RGB` care ne va permite setarea culorilor în RGB, alegem `GLUT_DEPTH` pentru a avea un buffer de adâncime ce ne permite utilizarea geometriei 3D și alegem `GLUT_DOUBLE` care ne permite utilizarea a doi buffers pentru randare. Prin utilizarea a doi buffers pentru randare ne asigurăm că nu desenăm direct pe ecran – fapt ce ar putea rezulta în desenări incomplete –, ci desenarea se face într-un buffer iar randarea(afișarea) se face la finalul desenării, în alt buffer.

Funcțiile `glutInitWindowPosition()` și `glutInitWindowSize()` setează poziția și

mărimea ferestrei, după cum sugerează și numele.

Funcția `glutCreateWindow()` creează fereastra în care vom desena și primește ca parametru numele ferestrei.

Funcțiile `glutDisplayFunc()` si `glutIdleFunc()` se ocupă de afișarea desenării în fereastra pe care am creat-o și primesc ca parametru funcția în care se află codul pentru desenarea propriu-zisă. Aceasta este `RenderFunction()` pe care o vom analiza mai jos.

Funcțiile `glutKeyboardFunc()` și `glutSpecialFunc()` se ocupă de procesarea comenzilor de tastatură și mouse. Ele primesc ca parametrii funcțiile pe care le vom folosi pentru prinderea și prelucrarea event-urilor de mouse și tastatură, adică `processNormalKeys()` pentru tastatură și `processSpecialKeys()` pentru mouse.

Funcția `glutCloseFunc()` se apelează înainte ca fereastra să fie închisă și distrusă. Îi dăm ca parametru funcția `Cleanup()` în care vom distruge tot ce am inițializat manual.

Funcția `glutMainLoop()` se ocupă de loop-ul principal de desenare. Această funcție nu returnează niciodată.

3.2 Inițializare

Funcția `Initialize()` se va ocupa de inițializarea shader-elor și a Vertex Buffer Object.

Vom folosi un shader de fragmente generic [10]:

```
#version 400
in vec3 ex_Color;
out vec3 out_Color;

void main(void)
{
  out_Color=ex_Color;
}
```

În shader-ul de vârfuri [10] avem:

```
#version 400
layout(location=0) in vec4 in_Position;
layout(location=1) in vec3 in_Color;
out vec4 gl_Position;
out vec3 ex_Color;
uniform mat4 viewShader;
uniform mat4 projectionShader;

void main(void)
{
    gl_Position = projectionShader*viewShader*in_Position;
    ex_Color=in_Color;
}
```

Astfel, matricele de proiecție și vizualizare vor fi trimise de către funcția de randare.

Tot în funcția `Initialize()` trimitem și locațiile acestor matrici:

```
viewLocation = glGetUniformLocation(ProgramId, "viewShader");
projLocation = glGetUniformLocation(ProgramId, "projectionShader");
```

Vertex Buffer Object se ocupă cu trimiterea informațiilor despre vârfuri la placa video. Înainte să creăm obiectul, trebuie mai întâi să generăm random câteva informații despre primitivele pe care le vom folosi.

Vom avea două componente de bază:

1. Hexagonul regulat (Fig. 3.1)

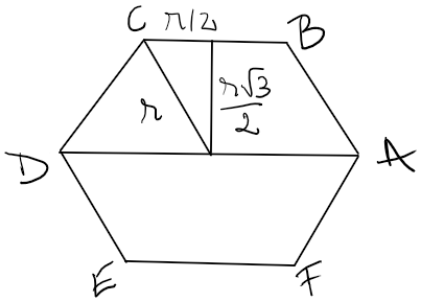

Fig. 3.1 – Hexagonul regulat cu distanțele relevante reprezentate

Alegem această formă pentru baza fulgului de zăpadă, dar și pentru extruziuni.

Fie O(ox, oy) punctul de origine al hexagonului. Fiind un hexagon regulat, acesta este circumscris unui cerc. Este suficient să generăm raza r, iar restul coordonatelor vârfurilor se vor afla ușor.

Astfel, avem punctele *A(ox + r, oy), B(ox + r/2, oy + √3r/2), C(ox – r/2, oy + √3r/2), D(ox – r, oy), E(ox - r/2, oy - √3r/2), F(ox + r/2, oy - √3r/2).*

Generăm raza hexagonului după formula simplă: *coreRadius = 5 + (rand() % 100).*

2. Pentagonul neregulat (Fig. 3.2, Fig 3.3)

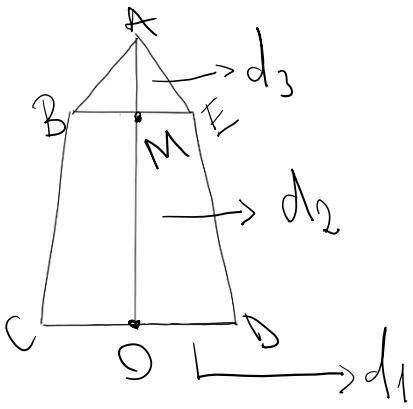

Fig. 3.2 – Pentagon cu d3 < d1

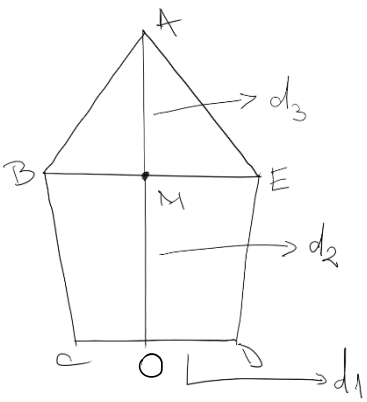

Fig. 3.3 - Pentagon cu d3 > d1

Alegem această formă pentru extruziunile fulgilor de zăpadă. Nefiind vorba de un pentagon regulat, trebuie să generăm mai multe dimensiuni. Acestea sunt:

d1 – baza pentagonului, dată de segmentul CD

d2 – înălțimea trapezului BCDE, dată de segmentul OM

d3 – pentru simplitate, am ales ca triunghiul ABE să aibă AM = BE, unde BE este baza triunghiului și AM înălțimea acestuia.

Punctele O și M sunt mijloacele laturilor CD, respectiv BE.

Vom alege ca punct de origine al pentagonului punctul O(ox, oy). Astfel se obțin punctele: *A(ox, oy + d2 + d3), B(ox – d3/2, oy + d2), C(ox – d1/2, oy), D(ox + d1/2, oy), E(ox + d3/2, oy + d2).*

Pentru generarea d1, d2, d3 am ales formulele:

d1 = 10 + (rand() % (coreRadius / 4))

d2 = coreRadius + (rand() % 150)

d3 = 5 + (rand() % (int)(coreRadius / 4))

Formulele sunt relative la raza hexagonului de bază, deoarece se vor atașa la vârfurile acestuia. Distanța dintre două vârfuri ale hexagonului este coreRadius. Pentru a nu avea suprapunere, trebuie ca distanța dintre două extruziuni să fie minim coreRadius/2. Am ales totuși coreRadius/4 pentru o estetică mai bună. Am ales să nu pun nicio restricție distanțelor d1, d2, d3 relativ la celelalte distanțe din pentagon.

Cu informațiile obținute vom construi funcția void `CreateVBO()`. Aici vom avea vectorii `GLfloat Vertices[]` și `GLubyte Indices[]`.

În `Vertices[]`, vom stoca un vârf sub forma: {x, y, z, dir, r, g, b} unde x, y, z sunt coordonatele vârfului, r, g, b sunt canalele red, green, blue pentru culoarea vârfurilor, iar dir este 1.0f dacă avem un punct (ceea ce este cazul) sau 0.0f dacă avem o direcție (nu se aplică). Deci, dir va fi mereu 1.0f, iar culoarea vârfurilor am ales-o arbitrar să fie alb. Culoarea fundalului va fi setată negru folosind funcția `glClearColor(0.0f, 0.0f, 0.0f, 0.0f)` în interiorul funcției `Initialize()`. Vom avea și o coordonată z pentru poziția punctelor. Dacă extindem punctul de origine la *O(ox, oy, oz)*, alegem vârful *A(ax, ay, oz)* pentru spatele poligonului și *A(ax, ay, oz + thickness)* pentru fața poligonului. Parametrul thickness ar putea fi generat random la rândul său, dar pentru că acest lucru nu ar fi denotat mari diferențe estetice, i-am dat valoarea arbitrară 3.0f pentru simplitate.

În `Indices[]` vom stoca triangularea celor două poligoane după cum urmează:

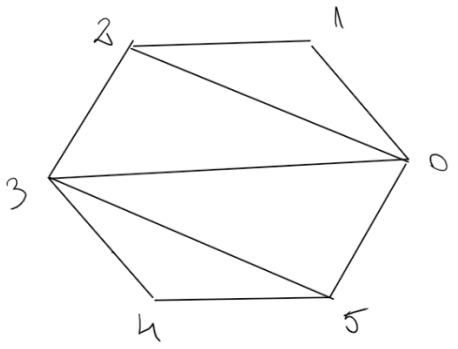

Fig. 3.4 – spatele hexagonului 3D

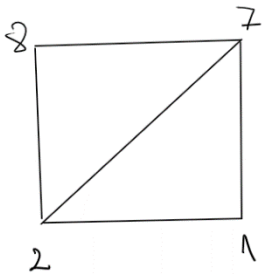

Fig. 3.5 – fața superioară a hexagonului 3D

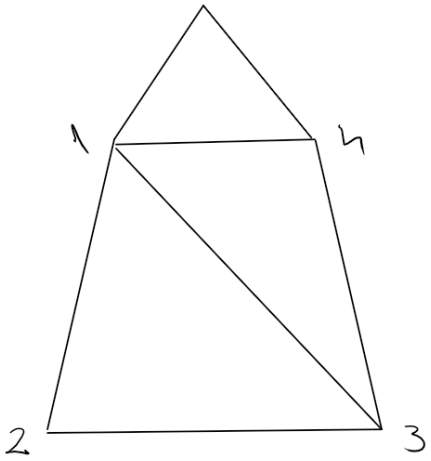

Fig. 3.6 – spatele pentagonului 3D

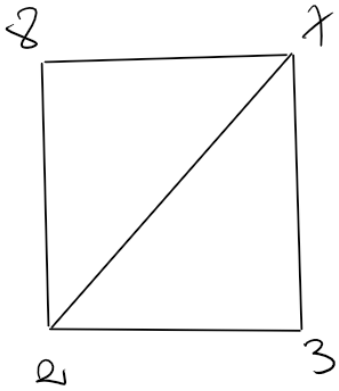

Fig. 3.7 – baza pentagonului 3D

Se extind aceste reguli de triangulare si la celelalte fețe și obținem:

```
GLubyte Indices[] =
{
	//Hexagon 3D

	//spate
	0, 1, 2,	2, 3, 0,
	3, 4, 5,	5, 0, 3,

	//lateral
	2, 3, 8,	8, 9, 3,
	3, 4, 9,	9, 10, 4,

	//bottom
	4, 5, 10,	10, 11, 5,

	//lateral
	1, 0, 6,	6, 7, 1,
	0, 5, 6,	6, 11, 5,

	//top
	1, 2, 7,	7, 8, 2,

	//fata
	6, 7, 8,	8, 9, 6,
	9, 10, 11,	11, 6, 9,
```

```
        //Pentagon 3D

        //spate
        12, 13, 16,         13, 14, 15,
        15, 16, 13,

        //lateral
        12, 13, 17,         17, 18, 13,
        13, 14, 18,         18, 19, 14,

        //bottom
        14, 15, 19,         19, 20, 15,

        //lateral
        16, 15, 20,         20, 21, 16,
        12, 16, 21,         21, 17, 12,

        //fata
        17, 18, 21,         18, 19, 20,
        20, 21, 18

    };
```

Unde vârfurile 0-5 sunt pentru spatele hexagonului (Fig. 3.4), 6-11 pentru fața hexagonului, 12-16 pentru spatele pentagonului (Fig. 3.6) și 17-21 pentru fața pentagonului. Triangularea s-a făcut doar in sens trigonometric deoarece ne interesează doar fața unui triunghi, nu și spatele.

Vertex Array Object (VAO) va conține toate Vertex Buffer Objects pe care le vom folosi și va stoca informațiile pentru un obiect complet care vor fi trimise ulterior la placa video. Generăm Vertex Array Object astfel:

```
    glGenBuffers(1, &VboId); // atribute
    glGenBuffers(1, &EboId); // indici

    glBindBuffer(GL_ARRAY_BUFFER, VboId);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vertices), Vertices, GL_STATIC_DRAW);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(Indices), Indices, GL_STATIC_DRAW);

    glEnableVertexAttribArray(0); // atributul 0 = pozitie
    glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 7 * sizeof(GLfloat), (GLvoid*)0);
    glEnableVertexAttribArray(1); // atributul 1 = culoare
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 7 * sizeof(GLfloat), (GLvoid*)(4
* sizeof(GLfloat)));
```

Pe scurt, creăm un buffer pentru vârfuri și unul pentru indici, legăm de acestea vectorii `Vertices[]` și `Indices[]` pe care i-am creat mai devreme, iar pentru vârfuri mai facem și împărțirea între poziții și culori.

Ne vom folosi doar de aceste două primitive pe care le-am generat (hexagonul și pentagonul) pentru a crea forma fulgului de zăpadă, urmând doar să creăm instanțe cu transformări diferite pentru componentele fulgului.

Tot în `Initialize()` va trebui să generăm și aceste transformări.

Vom avea 5 tipuri de fulgi:

```
enum SNOWFLAKE_TYPE {
	SIMPLE_PRISM,
	STELLAR_PLATE,
	STELLAR_DENDRITE,
	SECTORED_PLATES,
	FERN_DENDRITES
};
```

SIMPLE_PRISM va reprezenta un hexagon simplu, de bază. Trebuie să generăm doar raza hexagonului, lucru de care ne ocupăm deja când generăm coreRadius. Vom ține și o variabilă numită noOfExtrusions pentru cazurile următoare. In acest caz, *noOfExtrusions = 0*.

STELLAR_PLATE (Fig. 3.8) va fi format dintr-un hexagon de bază și extruziuni hexagonale.

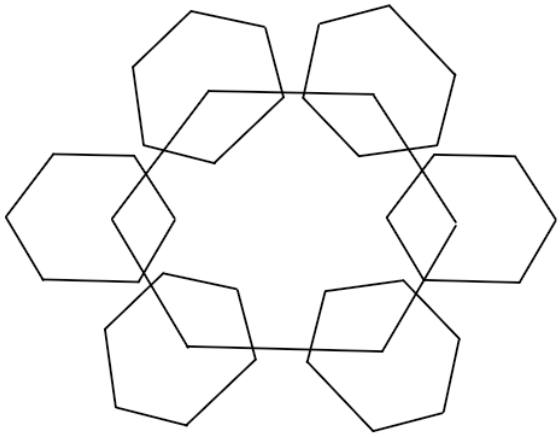

Fig. 3.8 – forma geometrică a STELLAR_PLATE

În acest caz *noOfExtrusions = 1* și va trebui să mai generăm o rază pentru hexagoanele de pe extruziuni, care va fi mai mică decât coreRadius.

STELLAR_DENDRITE (Fig. 3.9) va extinde STELLAR_PLATE, adăugând extruziuni extruziunilor existente, creându-se o formă de fractal.

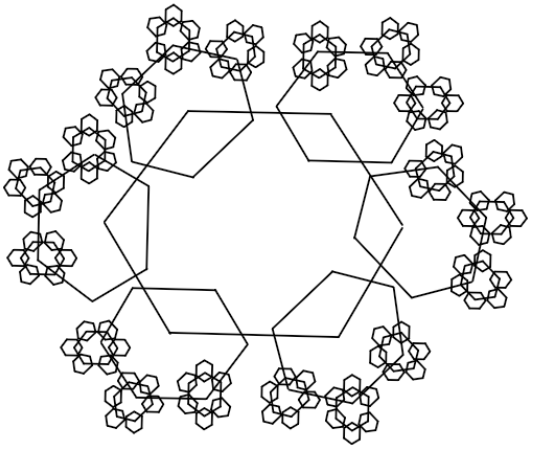

Fig. 3.9 – forma geometrică a STELLAR_DENDRITE

În acest caz vom folosi formula *noOfExtrusions = 1 + rand() % 7*, deci vom avea maxim 7 nivele de extruziune. Am ales un număr mic deoarece vom folosi o funcție recursivă cu complexitate exponențială pentru generare, dar oricum extruziunile vor deveni din ce in ce mai mici si nu vor mai putea fi vizibile.

Generarea extruziunilor se face astfel:

```
float minHexRadius = coreRadius;
for (int ii = 0; ii < noOfExtrusions; ii++) {
	if ((int)(minHexRadius / 4) > 0) {
		int radius = minHexRadius / 4 + (rand() % (int)(minHexRadius / 4));
		minHexRadius = radius;
		stellarDendriteSizes.push_back(radius);
	}
	else {
		noOfExtrusions = ii;
		break;
	}
}
```

Ținem o variabilă minHexRadius care reține raza extruziunii curente. Aceasta este inițializată cu valoarea coreRadius a razei hexagonului de bază. Pentru fiecare extruziune, vom calcula raza dupa formula:

radius = minHexRadius / 4 + (rand() % (minHexRadius / 4))

Astfel, fiecare nouă extruziune are raza cel puțin un sfert și cel mult jumătate din raza extruziunii pe care se formează. Deoarece extruziunile se prind de un hexagon si distanța dintre două puncte pe hexagon este r (raza hexagonului), înseamnă că raza extruziunilor nu trebuie să depășească r/2 ca să nu existe intersecții. Am ales să ofer și o dimensiune minimă a extruziunilor ca să fie echilibrate și să arate mai coerent din punct de vedere estetic. Vom ține razele extruziunilor astfel generate în vectorul `stellarDendriteSizes[]`.

SECTORED_PLATE (Fig. 3.10) introduce primitiva de tip pentagon în structura existentă.

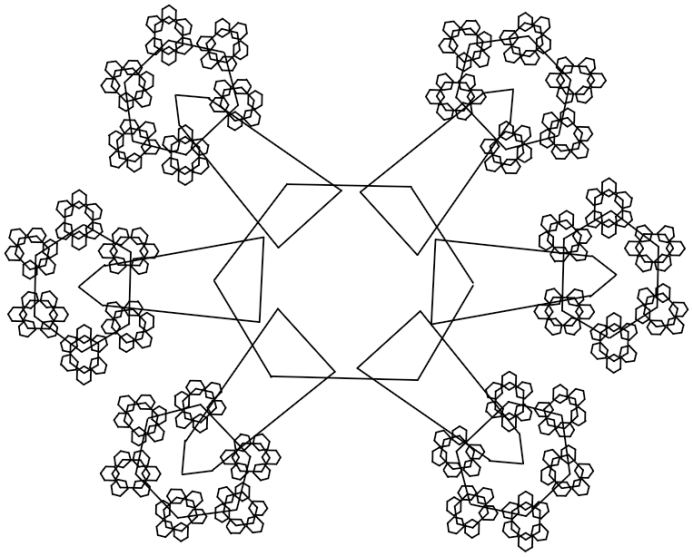

Fig. 3.10 – forma geometrică a STELLAR_PLATE

Vom folosi noOfExtrusions generat pentru tipul STELLAR_DENDRITE. Pentru extruziunea de tip pentagon am generat deja d1, d2 și d3, deci nu mai trebuie să generăm nimic.

FERN_DENDRITE utilizează doar primitiva de tip pentagon pentru a genera extruziuni.

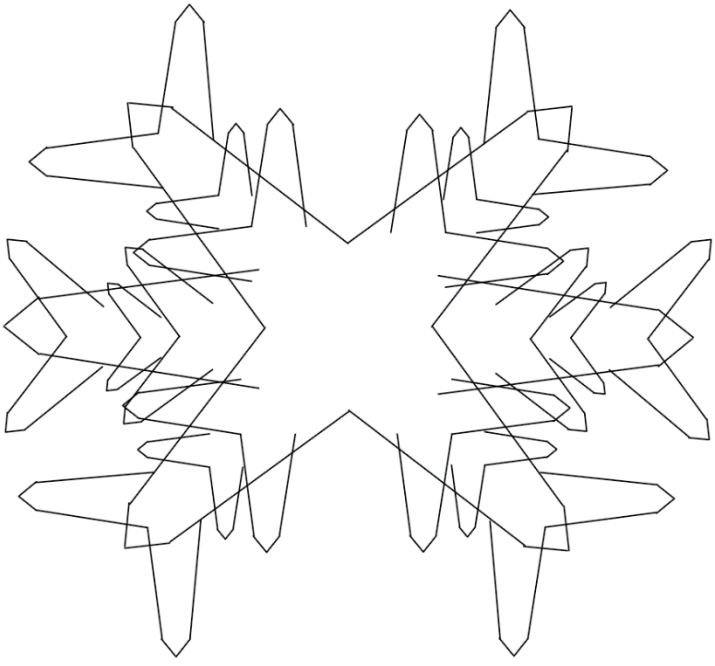

Fig. 3.11 – forma geometrică a FERN_DENDRITE

Pentru acest caz vom avea ca și bază șase poligoane principale de tip pentagon pe care le vom numi ramuri, aflate la unghi de 60° între ele, circumscriind un cerc de rază d2+d3. Ramurile vor avea un punct comun de origine, O. Pe ramuri vom genera extruziuni

iterativ în loc de a le genera recursiv ca în cazul STELLAR_DENDRITE.

Va trebui să tratăm diferit generarea extruziunilor. Nu vom mai genera un număr fix de extruziuni ci, în schimb, vom genera mărimile extruziunilor și distanțele dintre ele până când rămânem fără loc pe ramura pe care se formează.

Generarea extruziunilor se face astfel:

```
int sum = coreRadius;
while (sum < d2) {
	int dist = 10 + rand() % (int)((d2 - sum) / 2);
	sum += dist;
	fernDendritesDist.push_back(sum);
	float scale = 5 + rand() % 50;
	fernDendritesSize.push_back(scale / 100.0f);
	float angle = (rand() % (int)(2 * PI * 1e3)) * 1.0f/1e3;
	fernDendritesAngle.push_back(angle);
}
```

Variabila sum va ține locul ultimei extruziuni plasate. Distanța dintre două extruziuni se generează după formula *dist = 10 + rand() % ((d2 - sum) / 2)* unde d2 - sum reprezintă cât loc mai avem pe ramură. Am decis să împart d2 - sum la 2 pentru a avea un oarece echilibru. Pentru mărimea extruziunilor am ales să folosesc un procent care va fi relativ la mărimea ramurii de bază. Deci, după formulă, o extruziune nu poate fi mai mare de 55% din mărimea ramurii de bază. Am ales să generez si unghiul dintre ramură și extruziunea curentă cu o marjă de 3 zecimale. Unghiul poate avea orice valoare între 0 și 2π.

Vectorii `fernDendritesDist[]`, `fernDendritesSize[]`, `fernDendritesAngle[]` vor acționa ca un transform stack pentru translație, scalare și respectiv rotație.

3.3 Desenare

Desenarea se va realiza în interiorul funcției RenderFunction().

3.3.1 Pregătirea matricelor de vizualizare si proiecție

Începem prin a transmite Vertex Array Object de care vom avea nevoie ulterior pentru desenarea poligoanelor:

```
glBindVertexArray(VaoId);
glBindBuffer(GL_ARRAY_BUFFER, VboId);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EboId);
```

Alegem poziția observatorului:

```
Obsx = Refx + dist * cos(alpha) * cos(beta);
Obsy = Refy + dist * cos(alpha) * sin(beta);
Obsz = Refz + dist * sin(alpha);
```

Unde alpha și beta vor fi generate de inputul tastatură/mouse pentru survolarea figurii. Ref este punctul de referință care a fost ales în punctul de origine al vederii. Variabila dist reprezintă distanța față de referință la care vrem să ne poziționăm. Putem construi matricea de vizualizare și să o trimitem la shader:

```
glm::vec3 Obs = glm::vec3(Obsx, Obsy, Obsz);
glm::vec3 PctRef = glm::vec3(Refx, Refy, Refz); // pozitia punctului de referinta
glm::vec3 Vert = glm::vec3(Vx, Vy, Vz); // verticala din planul de vizualizare
view = glm::lookAt(Obs, PctRef, Vert);
glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &view[0][0]);
```

Vom folosi proiecția în perspectivă și generăm matricea aferentă, după care o trimitem shader-ului:

```
projection = glm::infinitePerspective(fov, GLfloat(width) / GLfloat(height),
znear);
glUniformMatrix4fv(projLocation, 1, GL_FALSE, &projection[0][0]);
```

3.3.2 Desenare SIMPLE_PRISM

Pe cazul cel mai simplu, SIMPLE_PRISM, va trebui doar să desenăm hexagonul pe care l-am generat în inițializare.

```
glDrawElements(GL_TRIANGLES, 60, GL_UNSIGNED_BYTE, 0);
```

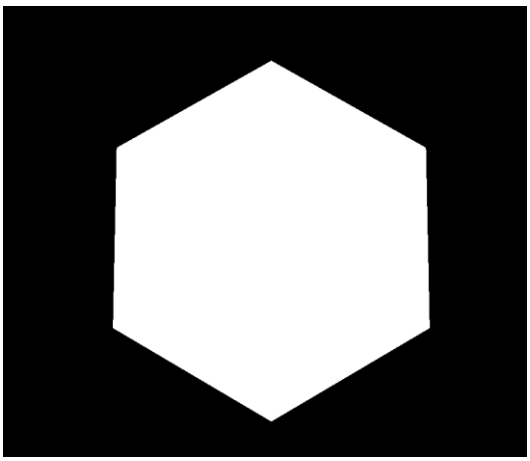

Fig. 3.12 – forma obținută de aplicație pentru cazul SIMPLE_PRISM

3.3.3 Desenare STELLAR_PLATE

Creștem puțin nivelul de complexitate cu generarea primului nivel de extruziune.

Fie structura:

```
struct point {
	float x, y, z;
}
```

Fie funcția:

```
void makeHexagon(vector<point>& vertices, float radius, point origin)
```

Aceasta primește ca parametru un vector `vertices[]`, o variabilă radius și o variabilă origin de tip point. Funcția va genera vârfurile unui hexagon cu raza radius și originea în origin și le va întoarce prin vectorul `vertices[]`.

Pentru a genera extruziunile, vom utiliza hexagonul de bază și îl vom translata și scala. Valorile obținute în `vertices[]` reprezintă translațiile pe care trebuie să le aplicăm extruziunilor.

Pentru scalare vom folosi formula *scale = size / coreRadius*, unde size este raza extruziunilor pe care am generat-o în inițializare.

Astfel, cu o instrucțiune de tip for vom itera prin toate cele 6 vârfuri și vom aplica scalarea și translația aferente. La fiecare pas aplicăm și o rotație de PI/3 față de rotația precedentă. După ce înmulțim matricea de vizualizare cu transformările obținute, o transmitem shader-ului și chemăm apelul de desenare.

```
	float size = stellarDendriteSizes[0];
	float scale = size / coreRadius;
	vector<point> verticesOffset;
	makeHexagon(verticesOffset, coreRadius, origin);
	for (int ij = 0; ij < 6; ij++) {
		point crtOffset = verticesOffset[ij];
		glm::mat4 matrTransl = glm::translate(glm::mat4(1.0f),
glm::vec3(crtOffset.x, crtOffset.y, crtOffset.z));

		glm::mat4 resizeMatrix = glm::scale(glm::mat4(1.0f), glm::vec3(scale,
scale, 0.75f));
		glm::mat4 rotMatrix = glm::rotate(glm::mat4(1.0f), rotAngle, glm::vec3(0.0,
0.0, 1.0));
		rotAngle += PI / 3;

		view = glm::lookAt(Obs, PctRef, Vert) * matrTransl * resizeMatrix *
rotMatrix;
		glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &view[0][0]);

		glDrawElements(GL_TRIANGLES, 60, GL_UNSIGNED_BYTE, 0);
		glEnd();
	}
```

Figura obținută astfel:

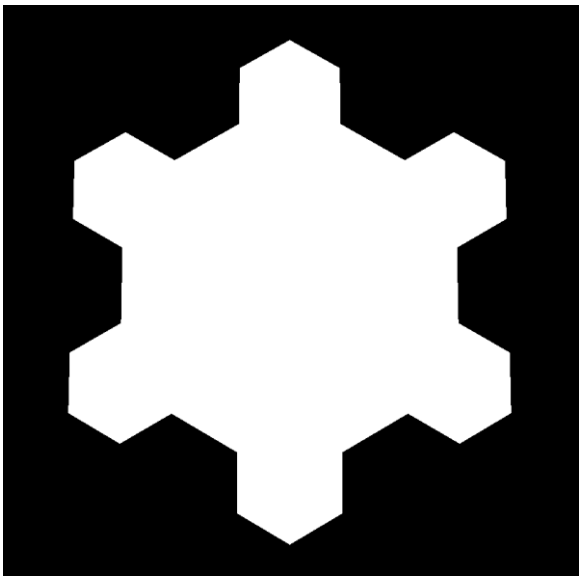

Fig. 3.13 – forma obținută de aplicație
pentru cazul STELLAR_PLATE

3.3.4 Desenare STELLAR_DENDRITE

Vom mai crește complexitatea figurii prin adăugarea recurentă de extruziuni.

Fie structura:

```
struct transform {
	point translate;
	float rotationAngle;
	float scale;
};
```

Trebuie modificată puțin abordarea problemei. Vom ține un vector de tip vector<transform> `trfArray[]`. Vrem să folosim o funcție recursivă pentru a desena extruziunile, dar contextul de desenare (VAO, VBO) se află în `RenderFunction()`, așa că în funcția noastră recursivă vom genera doar transformările pe care urmează să le aplicăm, și le vom stoca în vectorul `trfArray[]`.

Fie funcția:

```
void createExtrusions(int level, point origin, float hexRadius, float prevRadius,
vector<transform>& trfVector)
{
	float scale = hexRadius / coreRadius;
	float rotAngle = 0;
	vector<point> verticesOffset;
	makeHexagon(verticesOffset, prevRadius, origin);
```

```
            for (int ij = 0; ij < 6; ij++) {
                point crtOffset = verticesOffset[ij];
                transform trf;

                trf.translate = crtOffset;
                trf.scale = scale;
                trf.rotationAngle = rotAngle;
                trfVector.push_back(trf);

                rotAngle += PI / 3 / level;

                if (level < noOfExtrusions) {
                    createExtrusions(level + 1, crtOffset,
                stellarDendriteSizes[level], hexRadius, trfVector);
                }
            }
        }
```

unde level este nivelul curent de extruziune, origin și prevRadius sunt originea, respectiv raza hexagonului căruia vrem să îi aplicăm extruziuni, hexRadius este raza extruziunii și trfVector este stiva de transformări pe care le vom genera.

Scalarea extruziunii curente se face relativ la hexagonul de bază de la inițializare. Generăm translațiile relativ la originea și raza hexagonului căruia vrem să îi adăugăm extruziuni. Iterăm prin vârfurile obținute și adăugăm transformările în vectorul de transformări. Rotația se realizează la un unghi de PI / 3 / level pentru că, deși toate hexagoanele pe care le vom genera au unghiurile dintre vârfuri de 60°, rotația este relativă la hexagonul de bază și nu la hexagonul de referință.

Ca să trecem la următorul nivel de extruziune, apelăm:

```
createExtrusions(level + 1, crtOffset, stellarDendriteSizes[level], hexRadius, trfVector);
```

Astfel, vom adăuga extruziuni extruziunii pe care doar ce am creat-o. Noua origine va fi crtOffset, adică vârful pe care se află extruziunea curentă, noua prevRadius va fi raza curentă (hexRadius), raza noilor extruziuni va fi stellarDendriteSizes[level] deoarece level va fi indexat de la 1, pe când vectorul `stellarDendriteSizes[]` este indexat de la 0.

În interiorul `RenderFunction()` apelăm funcția de creare a extruziunilor astfel:

```
createExtrusions(1, origin, stellarDendriteSizes[0], coreRadius, trfArray);
```

La finalul apelului, mai rămâne decât să iterăm prin trfArray și să creăm matricele de transformare ca mai sus, să le aplicăm matricei de vizualizare, să trimitem shader-ului și să chemăm apelul de desenare.

Figurile obtinute:

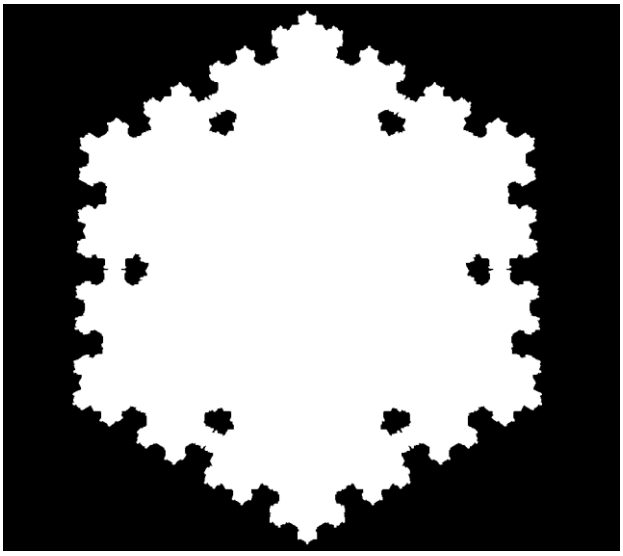

Fig. 3.14 – o formă obținută de aplicație pentru cazul STELLAR_DENDRITE

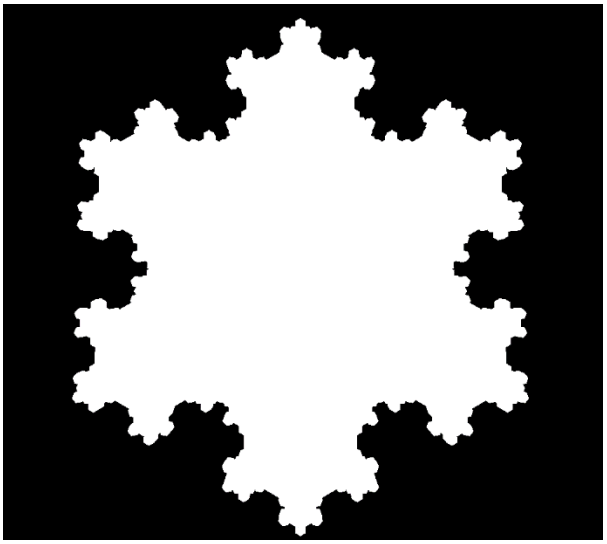

Fig. 3.15 – o formă obținută de aplicație pentru cazul STELLAR_DENDRITE

3.3.5 Desenare SECTORED_PLATES

Pentru generarea acestui tip de fulg păstrăm tot pipeline-ul anterior de desenare al extruziunilor. Tot ce trebuie să facem este să adăugăm „ramurile” de tip primitivă pentagon. Le generăm direct în `RenderFunction()`. Creăm 6 ramuri pe care le rotim la 60° una de cealaltă.

```
	float rotAngle = 0;

	for (int ij = 0; ij < 6; ij++) {
		codCol = 0;
		glUniform1i(codColLocation, codCol);

		glm::mat4 rotMatrix;
		rotMatrix = glm::rotate(glm::mat4(1.0f), (rotAngle), glm::vec3(0.0, 0.0,
1.0));
		rotAngle += PI / 3;

		view = glm::lookAt(Obs, PctRef, Vert) * rotMatrix;
		glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &view[0][0]);

		glDrawElements(GL_TRIANGLES, 48, GL_UNSIGNED_BYTE, (void*)(60));
		glEnd();

	}
```

Trebuie modificat și apelul funcției `createExtrusions()` din `RenderFunction()` în `createExtrusions (1, origin, stellarDendriteSizes[0], d2, trfArray)`. Am înlocuit coreRadius cu d2 deoarece dorim sa începem desenarea extruziunilor în vârful ramurilor.

Obținem figurile:

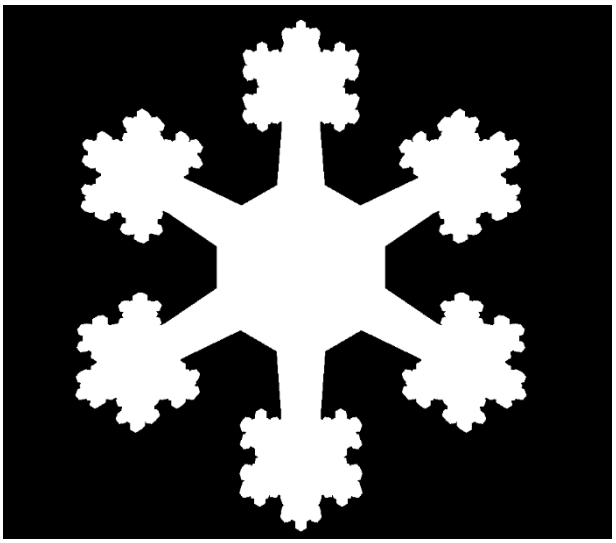

Fig. 3.16 – o formă obținută de aplicație pentru cazul SECTORED_PLATES

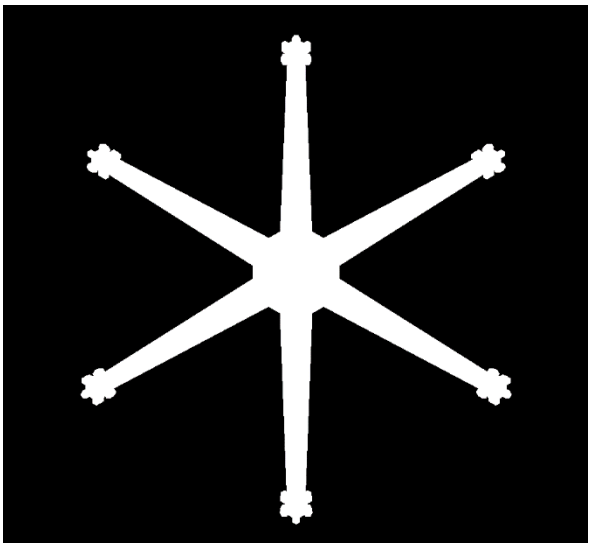

Fig. 3.17 – o formă obținută de aplicație pentru cazul SECTORED_PLATES

3.3.6 Desenare FERN_DENDRITES

Tipul FERN_DENDRITES se bazează pe o generare iterativă și folosește doar primitiva de tip pentagon, deci nu vom mai avea nevoie de codul legat de generarea hexagoanelor.

Păstrăm codul pentru generarea ramurilor. În interiorul său, vom genera extruziuni pentru fiecare ramură. După ce am desenat ramura curentă, îi vom adăuga ramificații prin intermediul funcției:

```
void createFernExtrusions(point origin, float alpha, float rotAngle, vector<transform>& trfVector)
```

Spre deosebire de `createExtrusions()` apelat anterior, `createFernExtrusions()` ține un vector de transformări doar pentru ramificațiile de pe ramura curentă, nu pentru toată structura.

Funcția `createFernExtrusions()` primește: un origin de care vom avea nevoie doar pentru aflarea coordonatei z, unghiul alpha de rotație la care se află ramura pe care ne aflăm față de primitiva pentagonală originală, variabila side care ne spune dacă generăm ramificațiile pe partea stângă sau dreaptă a ramurii, și vectorul de transformări trfVector.

```
void createFernExtrusions(point origin, float alpha, int side, vector<transform>&
trfVector)
{
	for (int ii = 0; ii < fernDendritesSize.size(); ii++) {
		transform trf;

		/*

			|\
			| \
		     a |  \  c
			|   \
			|____\alpha
			  b
		*/

		float c = fernDendritesDist[ii];
		float a = c * cos(alpha);
		float b = c * sin(alpha);
		point transl(a, b, origin.z);
		trf.translate = transl;

		trf.scale = fernDendriteSize[ii];

		if(side == 0)
			trf.rotationAngle = alpha - fernDendritesAngle[ii];
		else
			trf.rotationAngle = alpha - (PI - fernDendritesAngle[ii]);

		trfVector.push_back(trfL);
	}
}
```

Apelul `fernDendritesSize.size()` ne informează câte ramificații am generat. Iterăm prin aceste transformări pe care le-am generat în inițializare.

Vom nota cu c distanța de la punctul de origine la locația pe ramură a ramificației curente. Ca să aflăm punctul de pe ramură în care trebuie translatată ramificația, va trebui să ducem proiecțiile a și b pe primitiva de referință. Cunoaștem unghiul alpha și distanța c și trebuie să aflăm proiecțiile a și b utilizând formulele trigonometrice adecvate pentru a afla translația.

Scalarea se preia direct din `fernDendriteSize[]`.

Pentru rotație, vom folosi formula *alpha - fernDendritesAngle[ii]* ca să aplicăm ramificația pe partea stângă a ramurii și *alpha - (PI - fernDendritesAngle[ii])* pentru ramificația simetrică de pe partea dreaptă. Rotațiile trebuie, desigur, să fie relative la alpha, adică rotația ramurii de bază.

Apelul funcției createFernExtrusions() arată așa:

```
createFernExtrusions(origin, rotAngle, 0, trfArray);
createFernExtrusions(origin, rotAngle, 1, trfArray);
```

Pentru desenarea efectivă folosim forma de până acum:

```
for (int ii = 0; ii < trfArray.size(); ii++) {
	transform crtTrf = trfArray[ii];
	glm::mat4 matrTransl = glm::translate(glm::mat4(1.0f),
glm::vec3(crtTrf.translate.x, crtTrf.translate.y, 0.0f));
	glm::mat4 resizeMatrix = glm::scale(glm::mat4(1.0f),
glm::vec3(crtTrf.scale, crtTrf.scale, 1.0f));
	glm::mat4 rotMatrix = glm::rotate(glm::mat4(1.0f),
crtTrf.rotationAngle, glm::vec3(0.0, 0.0, 1.0));

	view = glm::lookAt(Obs, PctRef, Vert) * matrTransl * rotMatrix *
resizeMatrix;
	glUniformMatrix4fv(viewLocation, 1, GL_FALSE, &view[0][0]);

	glDrawElements(GL_TRIANGLES, 48, GL_UNSIGNED_BYTE, (void*)(60));
	glEnd();
}
```

Se obțin formele:

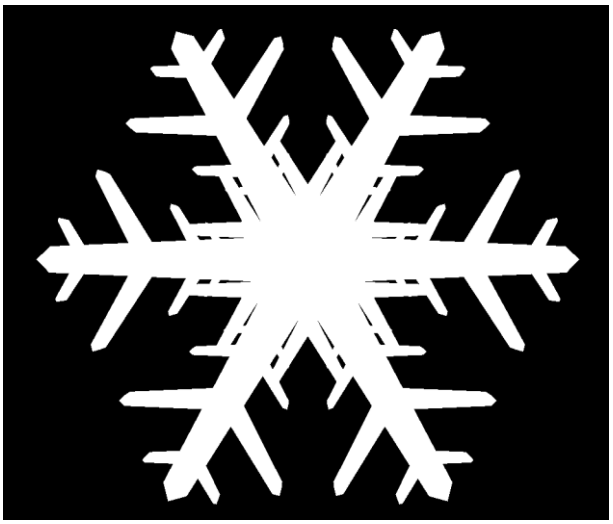

Fig 3.17 – o formă obținută de aplicație pentru cazul FERN_DENDRITES

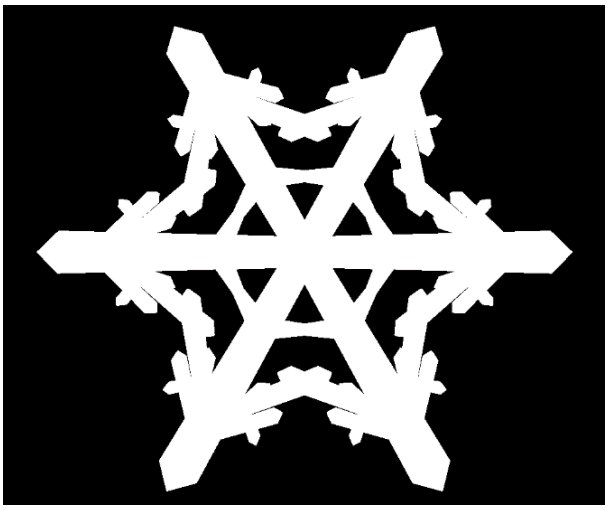

Fig 3.18 – o formă obținută de aplicație pentru cazul FERN_DENDRITES

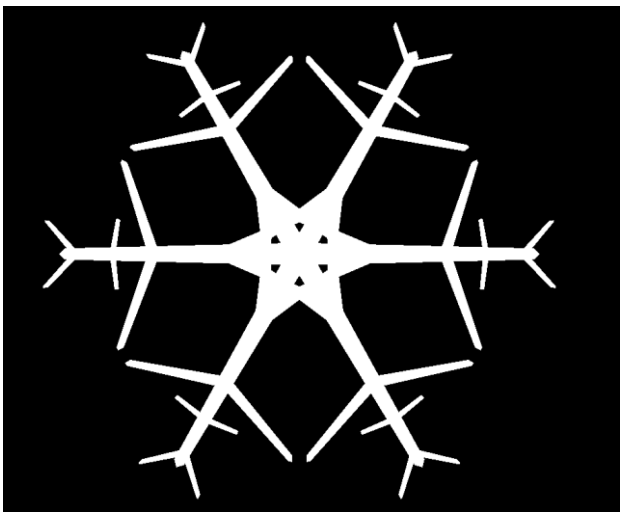

Fig 3.19 – o formă obținută de aplicație pentru cazul FERN_DENDRITES

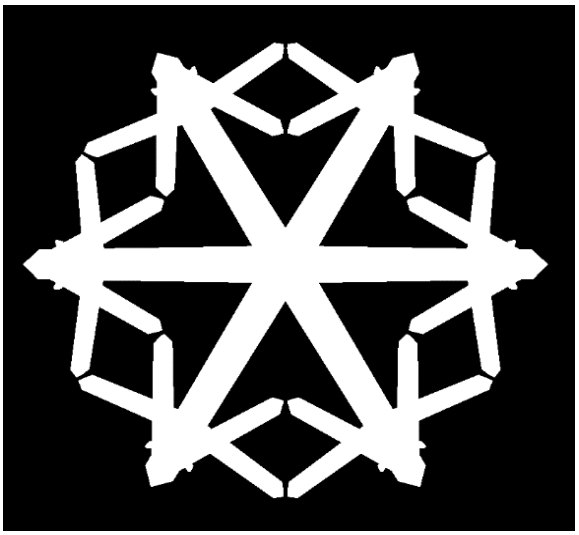

Fig 3.20 – o formă obținută de aplicație pentru cazul FERN_DENDRITES

3.3.7 Survolare

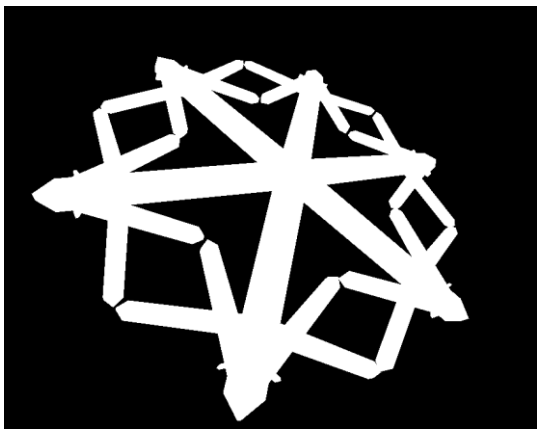

Fig 3.20 – un fulg de tip FERN_DENDRITE după ce i-au fost aplicate transformări din event-uri de tastatură

Pentru o mai bună vizualizare a formelor obținute a fost realizat și un mic mecanism de survolare și zoom.

Pentru zoom s-au procesat tastele „+” și „-” și un increment de 5.0.

```
void processNormalKeys(unsigned char key, int x, int y)
{
	switch (key) {
	case '-':
		dist -= 5.0;
		break;
	case '+':
		dist += 5.0;
		break;
	}
}
```

Pentru survolare în jurul formei (orbit) s-au procesat tastele arrow up, down, left, right. Valorile alpha și beta sunt în radiani și reprezintă unghiuri de rotație într-un interval de *[-PI/2, PI/2]*.

```
void processSpecialKeys(int key, int xx, int yy)
{
	switch (key)
	{
	case GLUT_KEY_LEFT:
		beta -= 0.01;
		break;
	case GLUT_KEY_RIGHT:
		beta += 0.01;
		break;
	case GLUT_KEY_UP:
		alpha += incr_alpha1;
		if (abs(alpha - PI / 2) < 0.05){
```

```
                        incr_alpha1 = 0.f;
                }
                else{
                        incr_alpha1 = 0.01f;
                }
                break;
        case GLUT_KEY_DOWN:
                alpha -= incr_alpha2;
                if (abs(alpha + PI / 2) < 0.05){
                        incr_alpha2 = 0.f;
                }
                else{
                        incr_alpha2 = 0.01f;
                }
                break;
        }
}
```

3.3.8 Finalizări și Cleanup

În primul rând, pentru a activa desenarea, trebuie ca la finalul funcției `RenderFunction()` să nu omitem chemarea funcției `glutSwapBuffers()` ca să ne asigurăm că buffer-ul pe care l-am încărcat cu informații despre fulgul nostru de zăpadă este trimis către desenare pe placa video. După aceasta, trebuie apelat `glutFlush()` care forțează execuția tuturor comenzilor GL și curăță buffer-ii.

Pentru distrugerea shader-elor chemăm funcția `glDeleteProgram(ProgramId)`, unde ProgramId este id-ul programului la care am atașat shader-ele. Atașarea shader-elor se face în inițializare cu ajutorul funcției `LoadShaders()` care este o funcție generică de încărcare a shader-elor adaptată dintr-un tutorial.

Pentru distrugerea Vertex Buffer Object chemăm funcțiile:

```
glDisableVertexAttribArray(1);
glDisableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glDeleteBuffers(1, &VboId);
glDeleteBuffers(1, &EboId);
```

care dezinițializează buffer-ii de vârfuri și indici, și dezactivează atributele de poziție și culoare ale buffer-ului de vârfuri. Acesta este un cod generic de dezinițializare prezent în documentația FreeGLUT [7].

Restul variabilelor folosite au fost preponderent locale, sau din librăria STL vector.h care se bazează pe memorie alocată dinamic și își face cleanup independent.

Capitolul 4

Concluzii

Consider că scopul aplicației a fost atins. Formele produse sunt suficient de complexe cât să fie ușor diferențiabile și unice în felul lor și să nu se ajungă la fenomenul de „procedural oatmeal”, termen propus de Kate Compton, care spune că deși este posibil să se genereze matematic mii de boluri de fulgi de ovăz folosind generarea procedurală, acestea vor fi percepute ca fiind aceleași de către utilizator și nu vor avea noțiunea de unicitate pe care ar trebui să o urmărească un sistem procedural.

Aplicația mi-a oferit o experiență de învățare pozitivă și o mai bună înțelegere a pipeline-ului OpenGL, precum și reîmprospătarea noțiunilor de geometrie analitică.

În continuare, întrevăd două posibilități de extindere a aplicației:

1. Combinarea/suprapunerea formelor obținute pentru a crea forme și mai complexe și a da mai multă valoare axei z.
2. Generarea unei scene cu zeci/sute/mii de obiecte generate procedural și compararea performanței sale cu o scenă în care fulgii sunt reprezentați de mesh-uri disjuncte, externe.

Ca o extindere mai mult de „estetică” a codului, aș încerca și să structurez aplicația după concepte de programare orientată pe obiecte. Acest lucru nu a fost însă o prioritate în timpul implementării, scopul meu fiind doar de a realiza un „proof of concept”. Abia când am început să adaug structuri pentru a îmi ușura munca am realizat că un cod mai compartimentat și reutilizabil m-ar fi ajutat mai mult în anumite aspecte ale implementării, și chiar devine necesar dacă doresc să iau în considerare rutele de extindere a aplicației propuse mai sus.

Bibliografie

[1] Ben Gilbert, „Video-game industry revenues grew so much during the pandemic that they reportedly exceeded sports and film combined”, Business Insider, Dec. 23, 2020 https://www.businessinsider.com/video-game-industry-revenues-exceed-sports-and-film-combined-idc-2020-12 (Ultima accesare 26.08.2022)

[2] Will Usher, „75% Of Gamers Say That Graphics Do Matter When Purchasing A Game”, Cinema Blend, Iun. 12, 2014, https://www.cinemablend.com/games/75-Gamers-Say-Graphics-Do-Matter-Purchasing-Game-64659.html (Ultima accesare 26.08.2022)

[3] Phil James, „Most Played & Most Popular Games In The World (2022)”, Gamer Tweak, Aug 2, 2022, https://gamertweak.com/most-played-popular-games/ (Ultima accesare 26.08.2022)

[4] Kenneth G. Libbrecht, „A Guide to Snowflakes”, Feb. 1, 1999, https://www.its.caltech.edu/~atomic/snowcrystals/class/class-old.htm#:~:text=A%20hexagonal%20prism%20is%20the,)%2C%20or%20anything%20in%20between. (Ultima accesare 11.07.2022)

[5] Documentație oficială OpenGL, https://www.khronos.org/opengl/ (Ultima accesare 29.08.2022)

[6] Kurt Akeley, Mark Segal „The OpenGL R Graphics System: A Specification (Version 4.0 (Core Profile) - March 11, 2010)” Mar. 11, 2010

[7] Documentație oficială FreeGLUT, http://freeglut.sourceforge.net/ (Ultima accesare 02.09.2022)

[8] Documentație oficială GLEW, https://www.opengl.org/sdk/libs/GLEW/ (Ultima accesare 02.09.2022)

[9] Documentație oficială GLM, https://glm.g-truc.net/0.9.9/index.html (Ultima accesare 02.09.2022)

[10] Joey de Vries, „Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion”, Jun. 17, 2020

Bibliografie

Referințe foto:

[10] https://gamerant.com/skyrim-clip-real-lydia-stuck-on-rocks/ (Ultima accesare 02.09.2022)

[11] https://www.origin.com/can/en-us/store/witcher/the-witcher-wild-hunt (Ultima accesare 02.09.2022)

[12] https://dribbble.com/shots/11201619-Procedural-Landscape-Model (Ultima accesare 02.09.2022)

[13] https://www.gamingscan.com/find-minecraft-world-seed/ (Ultima accesare 02.09.2022)

[14] https://www.minecraft.net/en-us/article/new-maps-available-minecraft-rtx-windows-10-beta (Ultima accesare 02.09.2022)