SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

-INTERAKTIVNA RAČUNALNA GRAFIKA-

Dokumentacija laboratorijskih vježbi

B-inačica

Ivan Juren

Voditelj: prof. dr. sc. Željka Mihajlović

Zagreb, svibanj, 2020

Sadržaj

Opisi laboratorijskih vježbi dostupni na sljedećoj poveznici

--- http://www.zemris.fer.hr/predmeti/irg/labosi/B_Labosi.pdf ---

Prva laboratorijska vježba

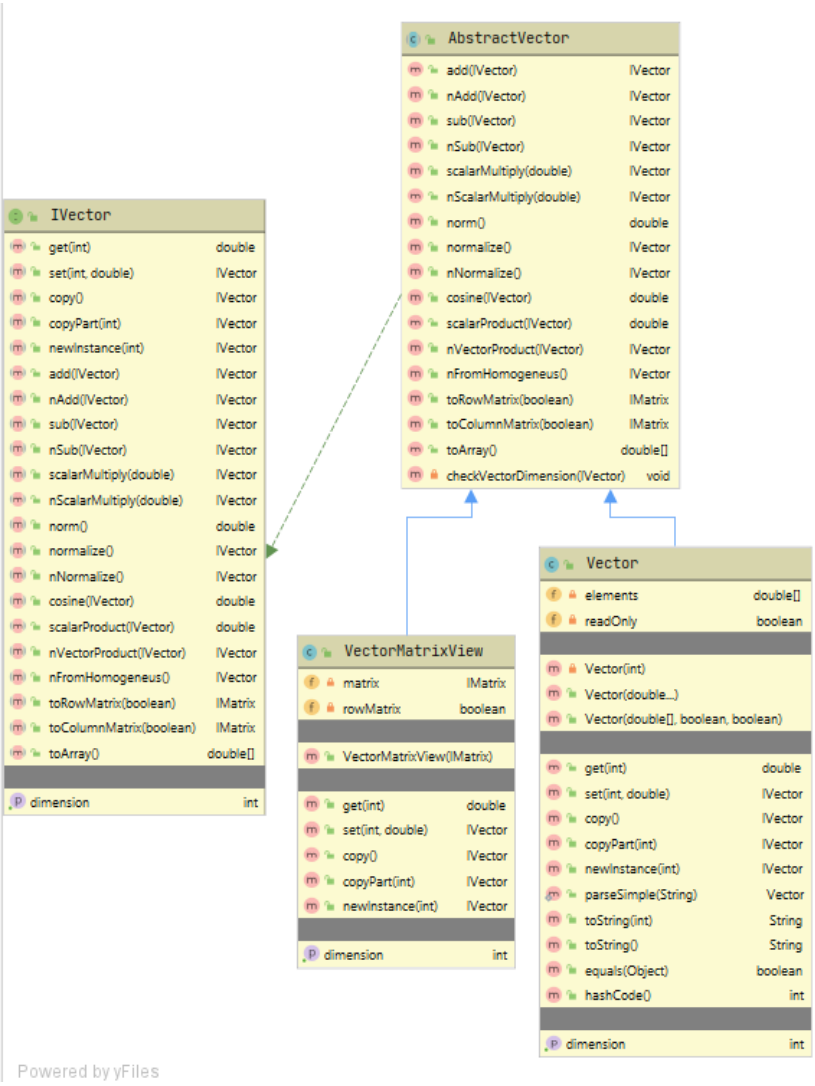
Izrada pomoćne biblioteke

U ovom zadatku razvijena je biblioteka za efikasno i jednostavno računanje s matricama i vektorima, te je omogućeno da matricu gledamo kao vektor i obratno ukoliko oblik strukture zadovoljava očekivani oblik (matrica sa jednim retkom ili jednim stupcem može se interpretirati kao vektor).

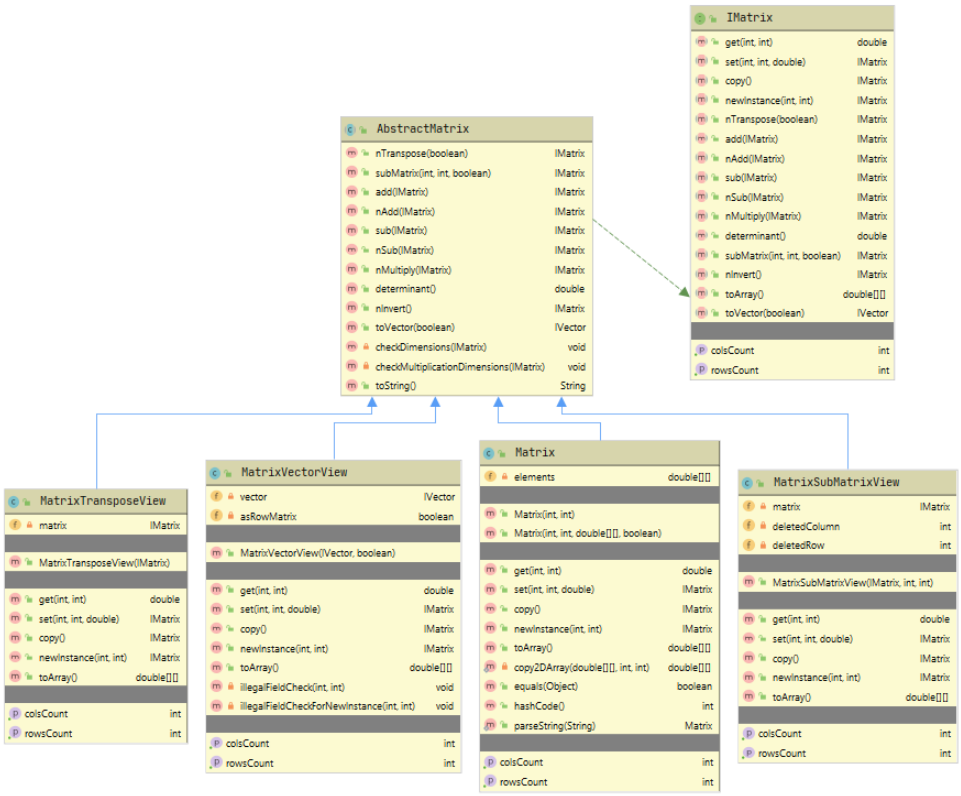
Vektore spremamo kao jednodimenzionalni niz brojeva double preciznosti, dok matrice spremamo kao dvodimenzionalni niz brojeva double preciznosti. Razvijeni pogledi (eng. View) nad matricama i vektorima zajedno s metodama propisanim ugovorima koje pružaju vektori i matrice omogućavaju sakrivanje implementacijskih detalja.

Razvijena biblioteka ima niz metoda od kojih mnoge imaju dvije inačice. Naprimjer, metoda za zbrajanje „add“ i metoda za zbrajanje „nAdd“. Metode naziva „nOperacija“ vraćaju novi vektor te ni na koji način ne modificiraju argumente nad kojima su pozvane.

U nastavku su prikazani dijagrami razreda biblioteke iz kojeg se mogu vidjeti ovisnosti i popis metoda koje svaki razred omogućava



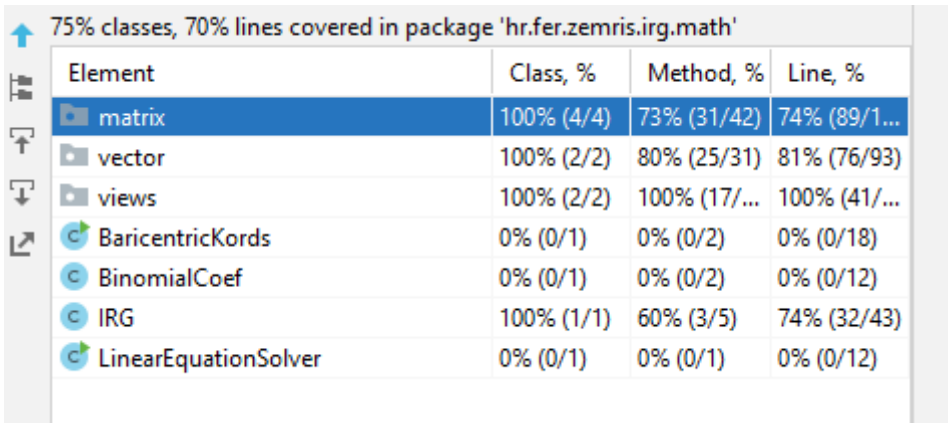
Slika 1 Dijagram razreda apstrakcije vektora



Slika 2 Dijagram razreda apstrakcije matrica

U sklopu ovog zadatka trebali smo napraviti i nekoliko pokaznih primjera kako se koristi ova biblioteka pa su iz tog razloga tu razvijeni razredi „BaricentricKords“ i „LinerEquationSolver“ koji prikazuju korištenje biblioteke.

Pošto je ovo dio labosa koji se kasnije kroz vježbe višestruko koristio, bilo je iznimno važno ovo napisati točno stoga postoje testovi koji testiraju implementaciju navedene biblioteke sa gotovo 80% pokrivenih slučajeva.



Slika 3 Pokrivenost testovima

Iz prethodne slike vidljivi su još neki razredi koje smo kasnije dodali u *„math“* paket.

Prvi program u OpenGl-u

Cilj ovog zadatka bio je upoznati se sa osnovama openGL-a. Implementacija vježbe odrađena je u razredu „TrianglesDrawer“ koji omogućuje crtanje trokuta raznih boja. U Sklopu ove vježbe razvio sam i paket „color“ koji ima enumeraciju „Color“ (najčešće korištene boje) i “ColorSupplier“ čije instance se ponašaju poput „State mashine-a“ ili kao beskonačni iterator.

```
package hr.fer.zemris.irg.color;

import ...

public class ColorSupplier {

    private List<Color> colorList;
    private int currentColorIndex;

    public ColorSupplier() {
        this.colorList = Arrays.stream(values()).collect(Collectors.toList());
        this.currentColorIndex = 0;
    }

    public Color setNext() { return this.set(1); }

    public Color setPrevious() { return this.set(-1); }

    private Color set(int step) {
        currentColorIndex = (currentColorIndex + colorList.size() + step) % colorList.size();
        return colorList.get(currentColorIndex);
    }

    public Color set(Color color) {
        int index = colorList.indexOf(color);
        currentColorIndex = index != -1 ? index : currentColorIndex;
        return color;
    }

    public Color get() { return colorList.get(currentColorIndex); }
}
```

Slika 4 Color supplier

Crtanje linija na rasterskim prikazajnim jedinicama

Cilj ove laboratorijske vježbe bio je napraviti crtanje linija Bresenhamovim algoritmom i odsijecanje linija Cohen–Sutherland algoritmom.

```
public static Optional<Line> segment(Line line, Point lowerLeftBorder, Point upperRightBorder) {

    int c0 = checkTBRL(line.getStart(), lowerLeftBorder.getX(), lowerLeftBorder.getY(), upperRightBorder.getX(), upperRightBorder.getY());
    int c1 = checkTBRL(line.getEnd(), lowerLeftBorder.getX(), lowerLeftBorder.getY(), upperRightBorder.getX(), upperRightBorder.getY());

    if ((c0 | c1) == 0) {
        return Optional.of(line);
    }

    if ((c0 & c1) != 0) {
        return Optional.empty();
    }

    return Optional.of(new Line(
            calculateNewPc0(line.getStart().copy(), line.getEnd().copy(), lowerLeftBorder, upperRightBorder),
            calculateNewPc0(line.getEnd().copy(), line.getStart().copy(), lowerLeftBorder, upperRightBorder),
            line.getColor()));
}
```

Slika 5 Nalaženje segmenta linije Cohen-Sutherland algoritmom

Nakon što se pronađe segment koji je potrebno iscrtati, pozvali smo vlastitu implementaciju Bresenhamovog algoritma koji liniju crta jednu po jednu rastersku jedinicu. Linije su raspoređene u 4 skupine te se poziva jedna od dvije metode sa pozitivnim (as-is) ili invertiranim (od kraja prema početku) smjerom crtanja linije.

```
public static void drawLine(GL2 gl2, Point pointStart, Point pointEnd, Color color) {
    gl2.glBegin(GL_POINTS);
    gl2.glColor3f(color.getR(), color.getG(), color.getB());

    if (Math.abs(pointEnd.getY() - pointStart.getY()) < Math.abs(pointEnd.getX() - pointStart.getX())) {
        if (pointStart.getX() > pointEnd.getX())
            plotLineForEachX(gl2, pointEnd.getX(), pointEnd.getY(), pointStart.getX(), pointStart.getY());
        else
            plotLineForEachX(gl2, pointStart.getX(), pointStart.getY(), pointEnd.getX(), pointEnd.getY());
    } else {
        if (pointStart.getY() > pointEnd.getY())
            plotLineForEachY(gl2, pointEnd.getX(), pointEnd.getY(), pointStart.getX(), pointStart.getY());
        else
            plotLineForEachY(gl2, pointStart.getX(), pointStart.getY(), pointEnd.getX(), pointEnd.getY());
    }
    gl2.glEnd();
}
```

Slika 6 Rješavanje problema određivanja osi koja će sigurno imati točno jednu rastersku jedinicu za svaku vrijednost

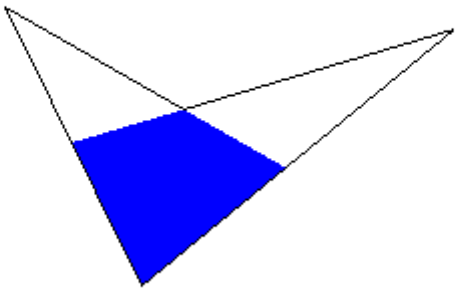
```
private static void plotLineForEachX(GL2 gl2, int x0, int y0, int x1, int y1) {
    int dx = x1 - x0;
    int dy = y1 - y0;
    int yi = 1;
    if (dy < 0) {
        yi = -1;
        dy = -dy;
    }
    int d = 2 * dy - dx;
    int y = y0;

    for (int x = x0; x < x1; x++) {
        gl2.glVertex2i(x, y);
        if (d > 0) {
            y = y + yi;
            d = d - 2 * dx;
        }
        d = d + 2 * dy;
    }
}
```

Slika 7 Implementacija Bresenhamovog algoritma koji za svaki pomak na x koordinati crta točno jednu rastersku jedinicu

Druga laboratorijska vježba

Crtanje i popunjavanje poligona

Cilj ove vježbe bio je implementacija crtanja i popunjavanja konveksnog poligona te određivanje konveksnosti unutar poligona. Za implementaciju ove vježbe ekstenzivno je korištena razvijena biblioteka. Osim osnovnog zadatka (koji se može naći na linku ispod sadržaja) dodao sam funkcionalnost da se prilikom na tipku „P“ za svaku točku ekrana provjeri nalazi li se ona unutar konkavnog poligona. Svaka takva točka obojana je plavom bojom. Rasterske jedinice bojane algoritmom iz udžbenika obojane su crnim. Kao što se vidi na sljedećim primjerima.



Slika 8 Točke za koje provjera jesu li unutar konkavnog poligona daje pozitivan rezultat

Slika 9 Konveksni poligon obojan algoritmom za konkavne poligone

U sklopu vježbe implementirana je i provjera hoće li poligon ostat konkavan i ako dodamo još jednu točku. Drugi dio vježbe bio je bojanje poligona koji radi kao scan-line algoritam. Za svaku y koordinatu provučemo pravac i nađemo koja je točka sjecišta najdesnija s lijeve strane i najlijevije desno sjecište te povučemo ravnu liniju između te dvije točke.

3D tijela

U sklopu ove vježbe implementirana ljuska koja ima mogućnost učitati tijelo, normalizirati ga, ispisati normalizirano tijelo, te određivati odnos unesenih točaka i učitanog tijela. Pošto u samoj vježbi nije bilo dovoljno dobro specificirano treba li ovo napraviti samo za konkavna tijela, implementirao sam da radi i za konkavna i konveksna tijela. Način na koji sam to napravio je sljedeći. Uzeo sam točku za koju treba provjeriti njen odnos. Iz te točke ispucao zraku u nekom smjeru (nije bitno kamo) tražio i brojao presjecišta sa plaštom tijela. Ukoliko je broj proboda neparan (gledamo samo jedan smjer ispucavanja zrake) tada zaključujemo da je točka bila unutar tijela. Ukoliko je broj proboda paran tada zaključujemo da je zraka ušla i izašla iz tijela.

```
> Task :ObjectModelerShell.main()
Loaded file:kocka.obj
Plane{a=0.0, b=-1.0, c=0.0, d=-0.0}
Plane{a=0.0, b=-1.0, c=0.0, d=-0.0}
Plane{a=1.0, b=0.0, c=0.0, d=-1.0}
Plane{a=1.0, b=0.0, c=-0.0, d=-1.0}
Plane{a=0.0, b=1.0, c=-0.0, d=-1.0}
Plane{a=0.0, b=1.0, c=0.0, d=-1.0}
Plane{a=-1.0, b=0.0, c=0.0, d=-0.0}
Plane{a=-1.0, b=0.0, c=0.0, d=-0.0}
Plane{a=0.0, b=0.0, c=-1.0, d=-0.0}
Plane{a=0.0, b=0.0, c=-1.0, d=-0.0}
Plane{a=0.0, b=0.0, c=1.0, d=-1.0}
Plane{a=0.0, b=0.0, c=1.0, d=-1.0}
Insert commands: (quit, normiraj, 1 2 3)

normiraj
v -1.0 -1.0 -1.0
v -1.0 -1.0 1.0
v 1.0 -1.0 -1.0
v 1.0 -1.0 1.0
v 1.0 1.0 -1.0
v 1.0 1.0 1.0
v -1.0 1.0 -1.0
v -1.0 1.0 1.0
f 1 3 2
f 3 4 2
f 3 5 4
f 5 6 4
f 5 7 6
f 7 8 6
f 7 1 8
f 1 2 8
f 1 5 3
f 1 7 5
f 2 4 6
f 2 6 8

0.3 0.4 0.5
Tocka {x=0.3, y=0.4, z=0.5} je UNUTAR tijela.
```

Slika 10 Korištenje ljuske

Problem prilikom računanja probodišta javio se kada bi zraka prošla kroz vrh ili brid. Ako bi zraka prošla kroz brid tada bi to probodište detektirao dvaput (za dva poligona), a ako bi prošla kroz vrh tada bi ju detektirao onoliko puta koliko poligona koristi taj vrh.

Da bi riješio problem duplog brojanja uveo sam pamćenje probodišta, no tu se javlja problem numeričke nepreciznosti. Zbog navedenih problema uvedena je heuristika. Kada se pronađe probodište provjeri se postoji li probodište koje je blizu jer ako postoji to je vjerojatno ta ista točka.

```
public Boolean calculatePosition(Vertex3D point) {
    List<IVector> intersectionsList = new ArrayList<>();
    int intersections = 0;
    Vector p0 = new Vector(point.getCords(), readOnly: true, referenceArray: false);
    Vector unit = new Vector( ...elements: 1, 0, 0);

    for (Face3D face : faces) {

        IVector norm = face.calculateCoefficients(vertices).toVector().copyPart( numberOfElements: 3);
        FaceCoeficient fc = face.calculateCoefficients(vertices);
        double s = -(fc.getA() * point.getX() + fc.getB() * point.getY() + fc.getC() * point.getZ() + fc.getD()) / (norm.scalarProduct(unit));
        if (s < 0) continue;
        IVector intersectionPoint = p0.nAdd(unit.nScalarMultiply(s));
        if (intersectionsList.stream().anyMatch(e -> e.nSub(intersectionPoint).norm() < 1e-6))
            continue;

        intersectionsList.add(intersectionPoint);
        Boolean onAFaceResult = onAFace(intersectionPoint, face);
        if (onAFaceResult == TRUE) intersections++;

        if (onAFaceResult == null) {
            if (onAFace(p0, face) == TRUE || onAFace(p0, face) == null) return null;
            else intersections++;
        }
    }
    return intersections % 2 == 1;

}
```

Slika 11 Provjera postoji li već navedena točka

Da bi provjerio nalazi li se točka u ravnini poligona koristim uvrštavanje u jednadžbu ravnine, a da bi provjerio nalazi li se unutar poligona koristim baricentrične koordinate.

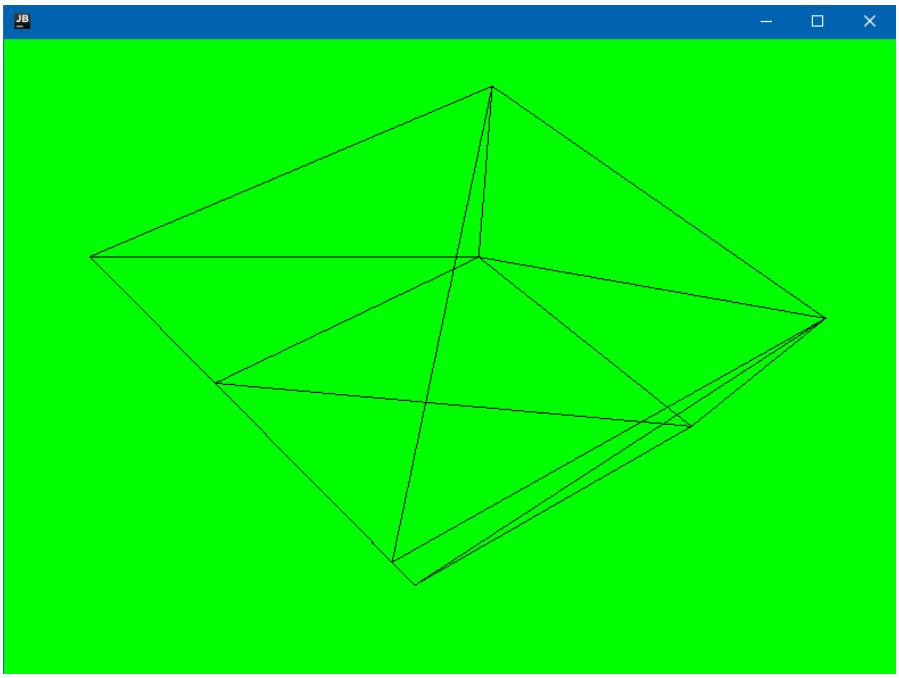
```
private Boolean onAFace(IVector point, Face3D face) {

    var fc = face.calculateCoefficients(vertices);
    if (Math.abs(fc.getA() * point.get(0) + fc.getB() * point.get(1) + fc.getC() * point.get(2) + fc.getD()) > 0.01)
        return false;

    Vertex3D a = vertices.get(face.getX0() - 1);
    Vertex3D b = vertices.get(face.getX1() - 1);
    Vertex3D c = vertices.get(face.getX2() - 1);

    Vector result = BaricentricKords.calculateBaricentricCords(new Vector(a.getCords()), new Vector(b.getCords()), new Vector(c.getCords()), point);

    for (int i = 0; i < 3; i++)
        if (result.get(i) > 1 || result.get(i) < 0) return false;

    if (result.get(0) > 0 && result.get(0) < 1
            && result.get(1) > 0 && result.get(1) < 1
            && result.get(2) > 0 && result.get(2) < 1)
        return true;

    return null;
}
```

Slika 12 Korištenje jednadžbe ravnine i baricentričnih koordinata

Treća laboratorijska vježba

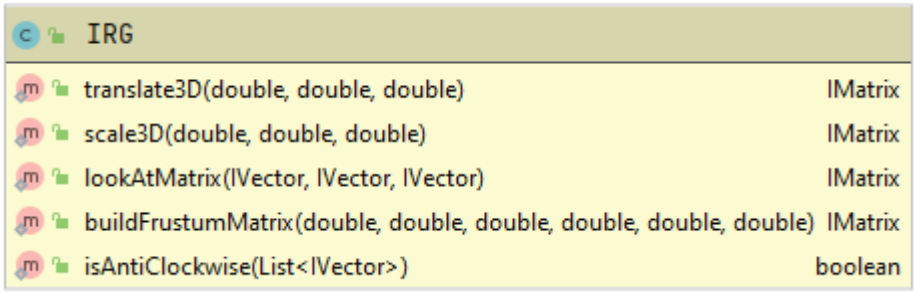
Projekcije

Cilj ove vježbe bio je proučiti što su to projekcije, koje vrste postoje i kako se radi perspektivna projekcija. U sklopu ove vježbe prošili smo „math“ biblioteku sa novim razredom „IRG“ koji zna raditi odrežene transformacije. Program se oslanjao na implementaciju ljuske iz prethodnog zadatka čiji je zadatak bio učitati datoteku.



Slika 13 Prikaz modela kocke razvijenim projekcijama

Implementacije ovih zadataka nalaze se u „projections“ paketu. ProjectionDrawer koristi glu.lookAt() naredbu da bi odredio gledište i očište. ProjectionDrawer2 koristi glu.gluPerspektive() dok ProjectionDrawer3 ima implementira te iste funkcionalnosti unutar IRG razreda.



Slika 14 Sadržaj razreda IRG

Uklanjanje skrivenih poligona

Uklanjanje skrivenih poligona implementirano je na tri načina, te je prije tog dijela vježbe bilo potrebno implementirati odbacivanje stražnjih poligona z-spremnikom.

Algoritam 1. Poligon je prednji ako se očište nalazi iznad ravnine u kojoj leži poligon. Poligon je stražnji ako se očište nalazi ispod ravnine u kojoj leži poligon. Neka je jednadžba ravnine u kojoj leži poligon jednaka ax+by+cz+d = 0. Slijedi da je poligon prednji ako je a·eyeX+b·eyeY+c·eyeZ+d > 0.

Algoritam 2. Neka je s c označen centar poligona (tj. trokuta): c = (V1+V2+V3)/3 . Neka je s e označen vektor iz centra poligona prema promatraču: e = eye − c. Konačno, neka je n normala na ravninu u kojoj leži poligon. Poligon je prednji ako je apsolutna vrijednost kuta između vektora n i e manja od 90◦ , tj. ako je kut iz (−90 , 90).

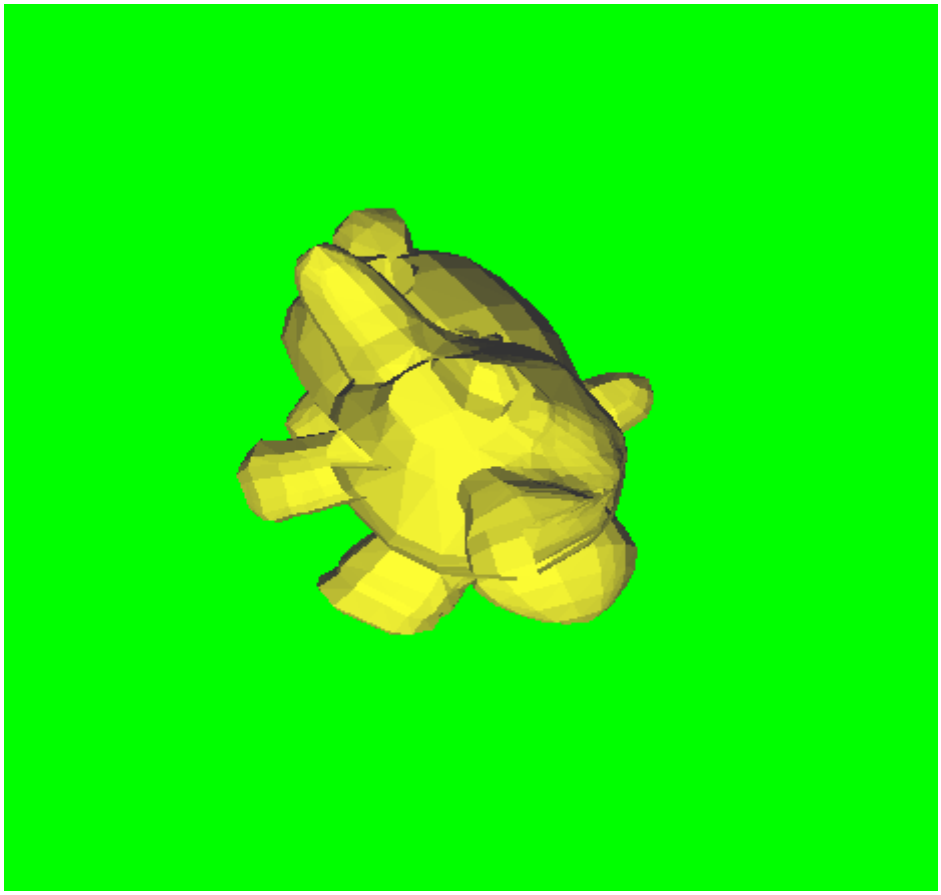
Algoritam 3. Neka su točke trokuta V1, V2 te V3 iz 3D sustava scene preslikane u 2D sustav projekcije kao V'1 , V'2 i V'3. Poligon je prednji ako je smjer obilaska točaka V'1 , V'2 i V'3 suprotan smjeru obilaska kazaljki na satu.

```
private void addKeyBoardListeners() {
    glCanvas.addKeyListener((KeyAdapter) keyPressed(e) → {
            switch (e.getKeyCode()) {
                case VK_R:
                    calculateLookVariables(increment);
                    break;
                case VK_L:
                    calculateLookVariables(-increment);
                    break;
                case VK_NUMPAD1:
                    preProjectionTester = objectModel::algorithm1;
                    postProjectionTester = points -> true;
                    break;
                case VK_NUMPAD2:
                    preProjectionTester = objectModel::algorithm2;
                    postProjectionTester = points -> true;
                    break;
                case VK_NUMPAD3:
                    preProjectionTester = (face, eye) -> true;
                    postProjectionTester = IRG::isAntiClockwise;
                    break;
                case VK_NUMPAD4:
                    preProjectionTester = (face, eye) -> true;
                    postProjectionTester = points -> true;
                    break;
                case VK_ESCAPE:
                    initLookVariables();
                    break;
                default:
                    System.out.println("Unsupported key");
            }
            glCanvas.display();
    });
}
```

Slika 15 Dio koda koji prikazuje korištenje strategija za odabir algoritma odbacivanja

Problem koji se pokazao u ta tri slučaja su poligoni koje ne bi trebali moći vidjeti no oni su prednji. Te poligone nismo uspjeli maknuti iz scene ovim primitivnim algoritmima.

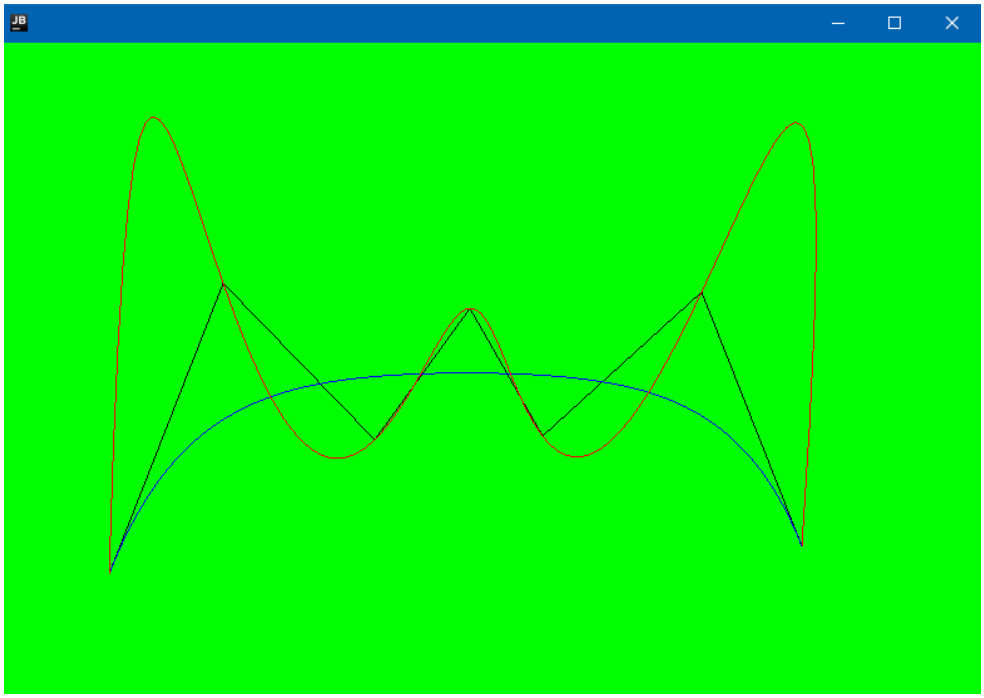
Slika 16 Referentni prikaz objekta



Slika 17 Prikaz objekta gdje su odbačeni stražnji poligoni sa ova tri algoritma bez uporabe z-spremnika

Bezierova krivuja

U ovoj vježbi implementiran je program koji za kontrolni poligon (crno) iscrtava interpolacijsku (crveno) i aproksimacijsku (plavo) krivulju



Slika 18 Bezierove krivulje

```
public void draw_curve(List<IVector> points, int divs, GL2 gl2, Color color) {
    int n = points.size() - 1;
    var factors = binomialCoef.computeFactors(n);

    gl2.glBegin(GL_LINE_STRIP);
    gl2.glColor3f(color.getR(), color.getG(), color.getB());

    for (int i = 0; i <= divs; i++) {
        double t = 1.0 / divs * i;
        double x = 0, y = 0, b;

        for (int j = 0; j <= n; j++) {

            if (j == 0) b = Math.pow(1 - t, n);
            else if (j == n) b = factors.get(j) * Math.pow(t, n);
            else b = factors.get(j) * Math.pow(t, j) * Math.pow(1 - t, n - j);

            x += b * points.get(j).get(0);
            y += b * points.get(j).get(1);
        }
        gl2.glVertex2d(x, y);
    }
    gl2.glEnd();
}
```

Slika 19 Dio koda koji crta bezijerovu krivulju pomoću Bernsteinovih težinskih vrijednosti za listu točaka (kontrolni poligon).

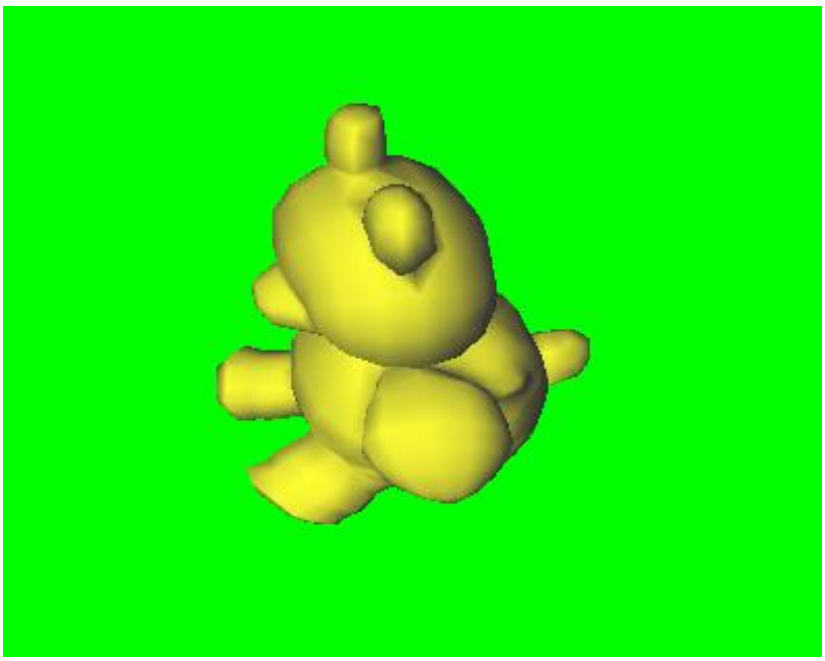
```
public class BinomialCoef {

    HashMap<Integer, List<Integer>> cache = new HashMap<>();

    public BinomialCoef(int n) { for (int i = 0; i < n; i++) computeFactors(i); }

    public List<Integer> computeFactors(int n) {
        if (cache.containsKey(n)) return cache.get(n);

        List<Integer> factors = new ArrayList<>();
        int a = 1;

        for (int i = 1; i <= n + 1; i++) {
            factors.add(a);
            a = a * (n - i + 1) / i;
        }
        cache.put(n, List.of(factors.toArray(new Integer[0])));
        return factors;
    }

}
```

Slika 20 Dio koda koji se koristi za memoizaciju binomnih koeficijenata da ih efikasnije izračuna

Četvrta laboratorijska vježba

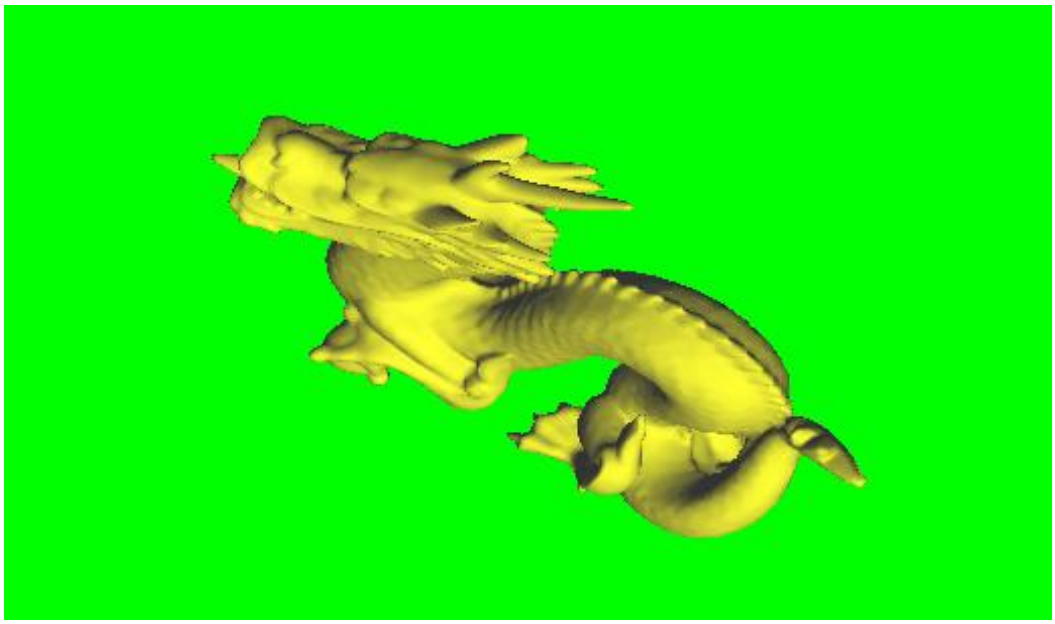
Sjenčanje

Ideja ove vježbe bilo je proučiti Gouraudovo i Konstantno sjenčanje te implementirati oboje uz referentnu/paralelnu implementaciju pomoću OpenGl-a.

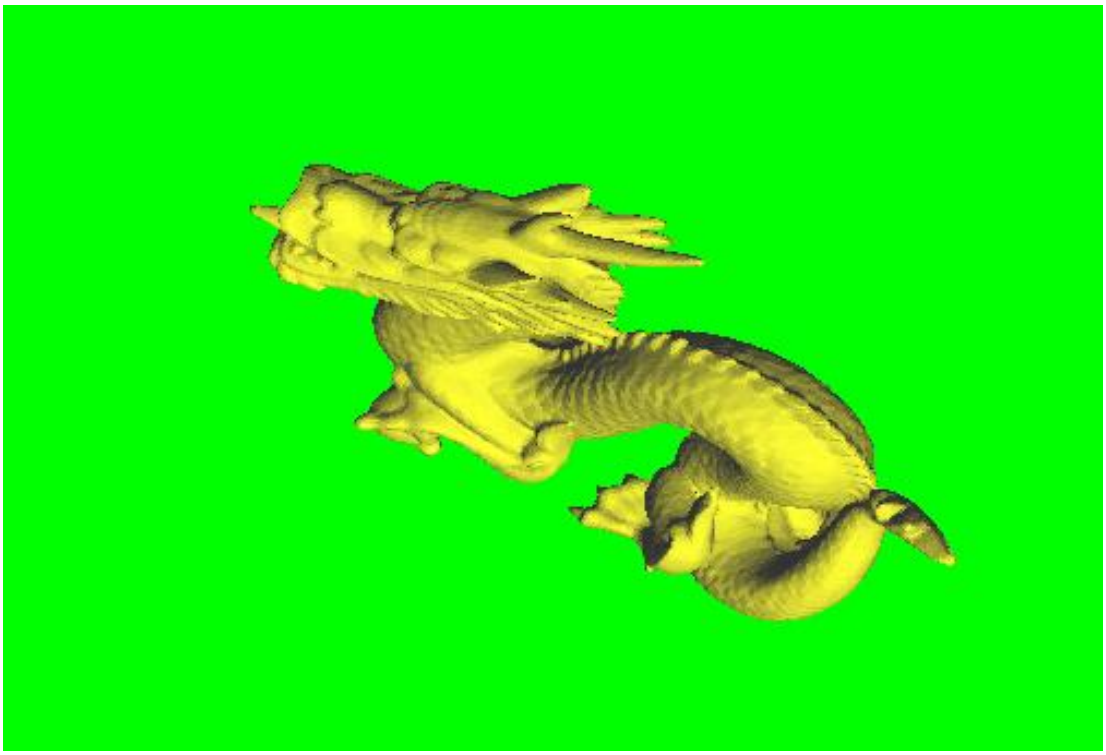


Slika 21 Objekt uz Gouraudovo sjenčanje

Slika 22 Objekt uz Konstantno sjenčanje



Slika 23 Objekt uz Gouraudovo sjenčanje



Slika 24 Objekt uz konstantno sjenčanje

Kod implementacije ove vježbe bilo je zanimljivo za primijetiti da što je objekt detaljniji da konstantno sjenčanje izgleda sve više kao Gouraudovo sjenčanje zato što je mnogo poligona pa gotovo svaka rasterska jedinica postaje poligon za sebe. U tom slučaju Gouraudovo sjenčanje samo „omekša“ sliku tako da ona izgleda dosta „mekano“. Razlika između da dva sjenčanja je u tome što za konstantno koristimo jednu normalu (normala iz središta poligona) te sjenčanje interpoliramo samo po toj jednoj vrijednosti za cijeli poligon. Za Gouraudovo sjenčanje računamo poseban vektor za svaki vrh. Svakih od ih vektora se računa kao prosjek normala susjednih poligona.

```
public double[] getCentralForFace(Face3D face) {
    double[] sum = new double[]{0, 0, 0};

    for (Vertex3D v : getVerticesOfFace(face)) {
        var cords = v.getCords();
        sum[0] += cords[0];
        sum[1] += cords[1];
        sum[2] += cords[2];
    }

    for (int i = 0; i < 3; i++) sum[i] /= 3;
    return sum;
}
```

Slika 25 Dio koda koji računa prosječnu normalu s obzirom na normale vrhova

Fraktali

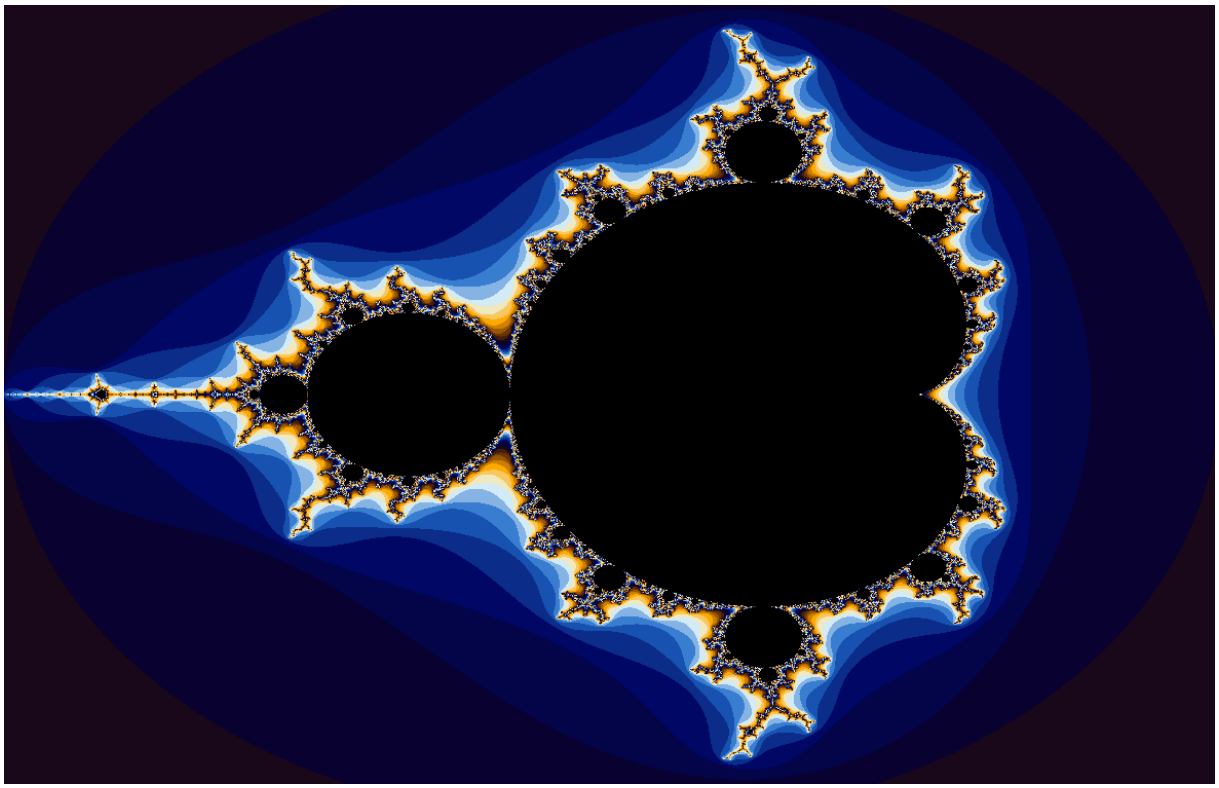
Mandelbrotov fraktal

U ovoj vježbi implementirano je jednodretveno i višedretveno rješenje. Kod jednodretvenog rješenja, program prolazi jednu po jednu rastersku jedinicu i izračunava koliko je potrebno da ona divergira. Kod višedretvenog rješenja, ekran je podijeljen na osam puta broj virtualnih jezgri procesora vertikalnih traka. Svaku tu traku modeliram kao jedan „posao“ koji onda odrađuje jedna dretva. Sve dretve svoje rezultate zapisuju u niz koji predstavlja brzinu divergencije pojedine točke.

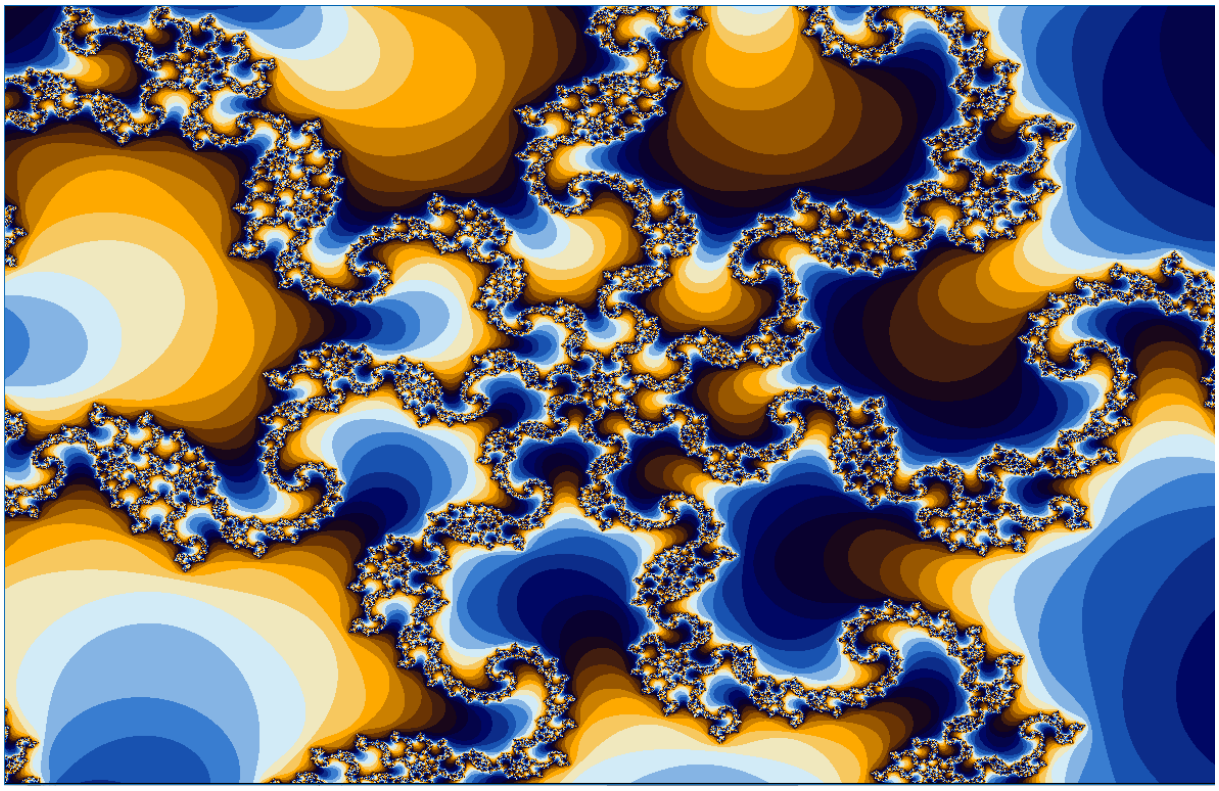
Nakon implementacije višedretvenosti, naišao sam na problem uskog grla između procesora i radne memorije zbog velikog broja objekata koji se stvarao prilikom računanja. Taj dio sam riješio izbjegavajući korištenje objekata kompleksnih brojeva i zamjenjujući ih primitivnim tipovima.

Nakon završetka logike posvetio sam se bojanju pa sam napravio malo drugačije bojanje koje je prikazano na slikama ispod. Bojanje koristi paletu od 16 boja te se boje ponavljaju svakih 16 dubina. Npr. Jedna nijansa plave se ponovi svakih 16n+1 puta.

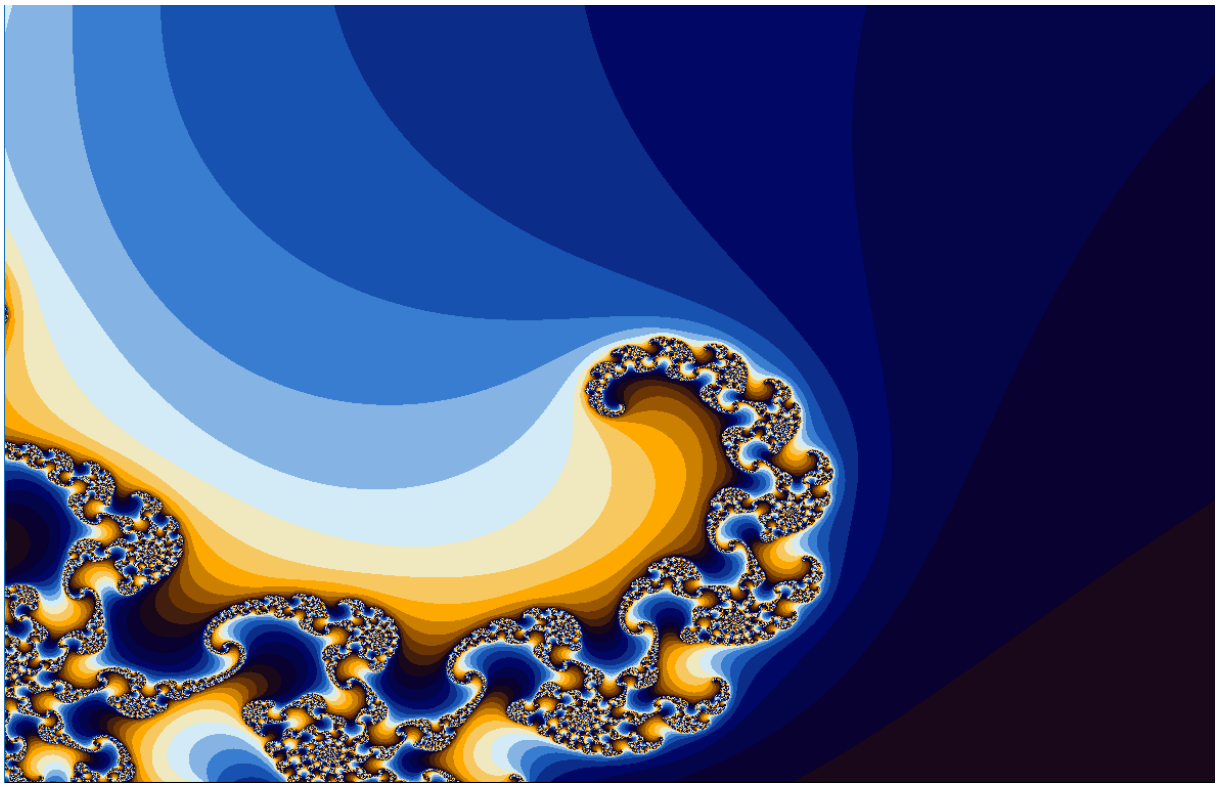
Osim toga ostvareni su i trivijalne implementacije bojanja (konvergira-crno, divergira-bijelo) i bojanje zadano u udžbeniku. Te implementacije napravio sam oblikovnim obrascem strategija koje se aktiviraju pritiskom na tipku – obrazac promatrač.



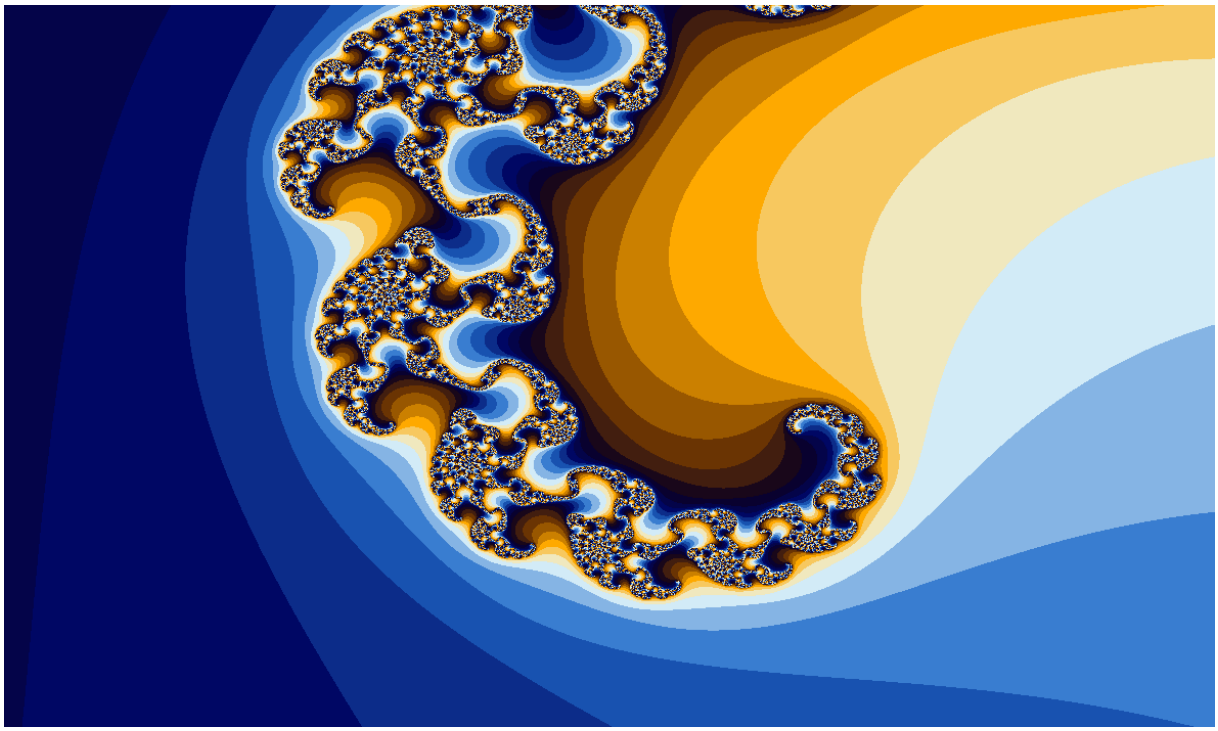
Slika 26 Mandelbrotov fraktal



Slika 27 Mandelbrotov fraktal - detalj 1



Slika 28 Mandelbrotov fraktal - detalj 2



Slika 29 Mandelbrotov fraktal - detalj 3

L-Sustavi

Unutar implementacije L-Sustava koristio sam više oblikovnih obrazaca a sam L sustav definiran je iz datoteke koja se učitava prilikom pokretanja. Dodatno sam ugradio više promatrača na pritisak tipki koji dinamički mijenjaju prikazan L-sustav ili povećavaju dubinu do koje možemo ići.

Svaka komanda oblikovana je „Command“ oblikovnim obrascem.

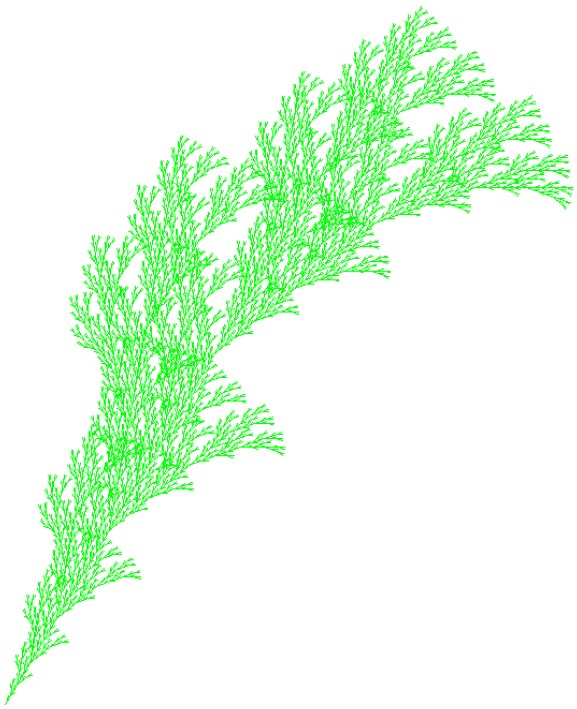
```java
/**
 * Class used as a implementation of {@link Command} interface.
 * <p>
 * Has constructor that takes one double as an argument and remembers it because
 * it is later used in {@link #execute(Context, Painter)} method to rotate
 * current angle
 *
 * @author juren
 */
public class RotateCommand implements Command {

    /**
     * variable that saves double representing angle that is later used in
     * {@link #execute(Context, Painter)}
     */
    private final double angle;

    /**
     * Constructor that gets {@link #factor} and saves it.
     *
     * @param angle used later in {@link #execute(Context, Painter)} method
     */
    public RotateCommand(double angle) { this.angle = angle; }

    @Override
    public void execute(Context ctx, Painter painter) {
        TurtleState curstate = ctx.getCurrentState();
        curstate.getDirection().rotate(Math.toRadians(this.angle));
    }

}
```

Slika 30 Prikaz obrasca naredbe kojim su ostvarene podržane naredbe

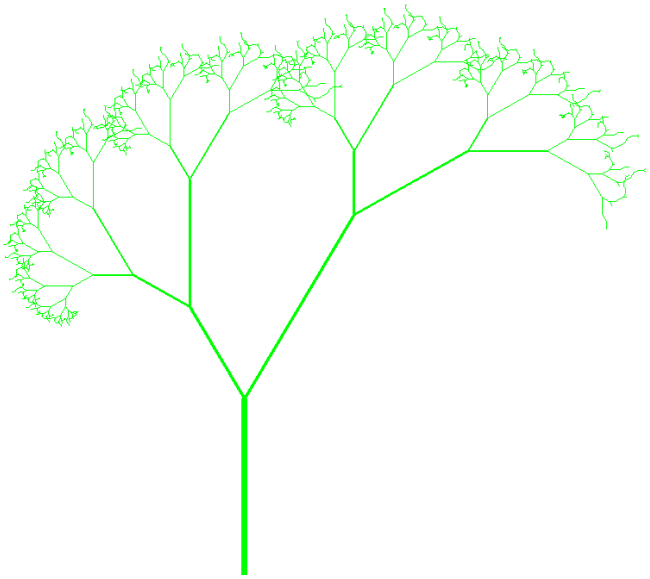
Dok za učitavanja iz datoteke koristimo obrazac graditelj. Datoteke kojima definiramo L-sustave imaju dio koji određuje početne uvjete, produkcije i aksiom.

```
origin                  0.38 0.0
angle                   90
unitLength              0.28
unitLengthDegreeScaler 1.0 / 1.003

command F draw 1
command + rotate 30
command - rotate -30
command s scale 0.6
command w pensize 0.6
command [ push
command ] pop
command G color 00FF00

axiom GFX

production Y FX+FY-FX
production X sw[-FY]+FX
```

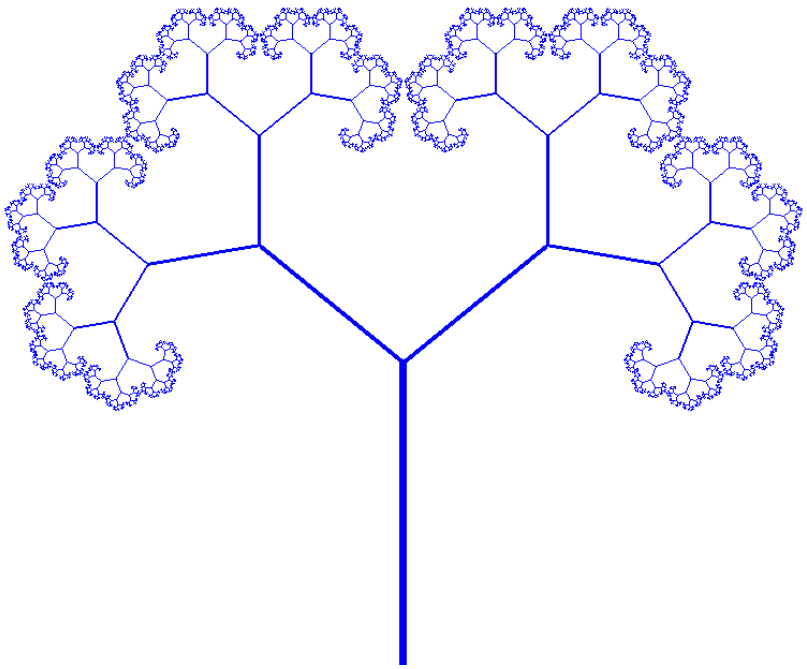
Slika 31 Promjer datoteke koja definira L-sustav



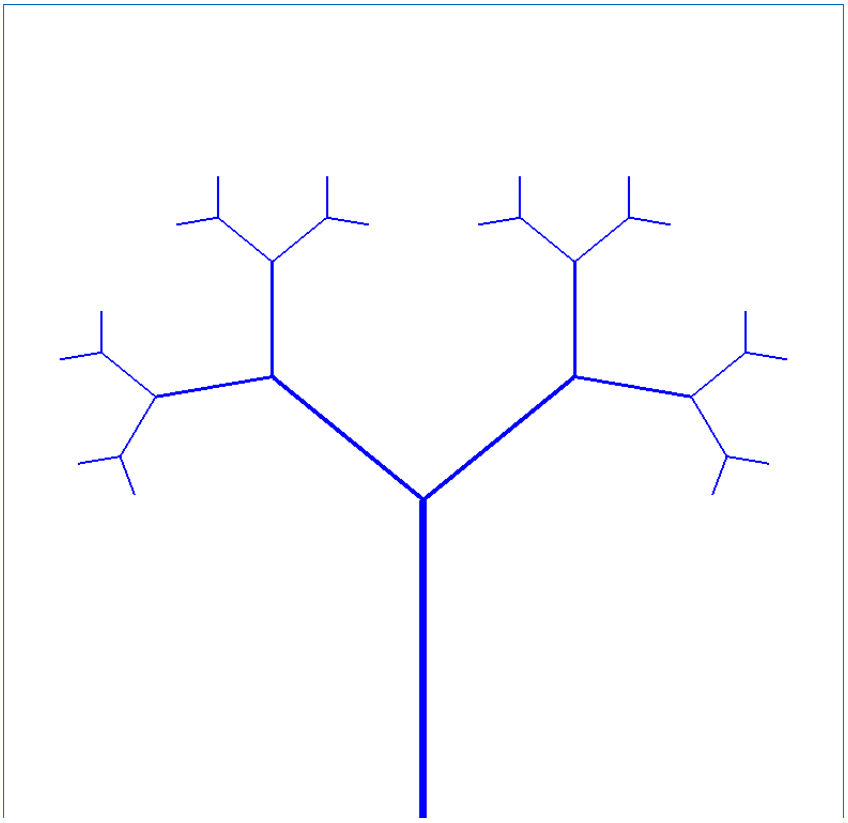
Slika 32 L-sustav - plant



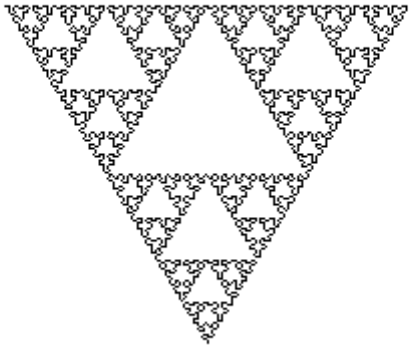
Slika 33 L-sustav - plant



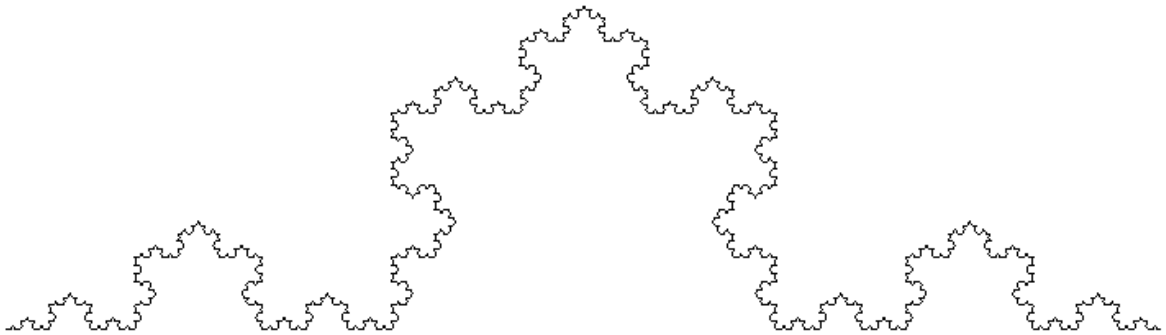
Slika 34 L-sustav - plant - dubina 10



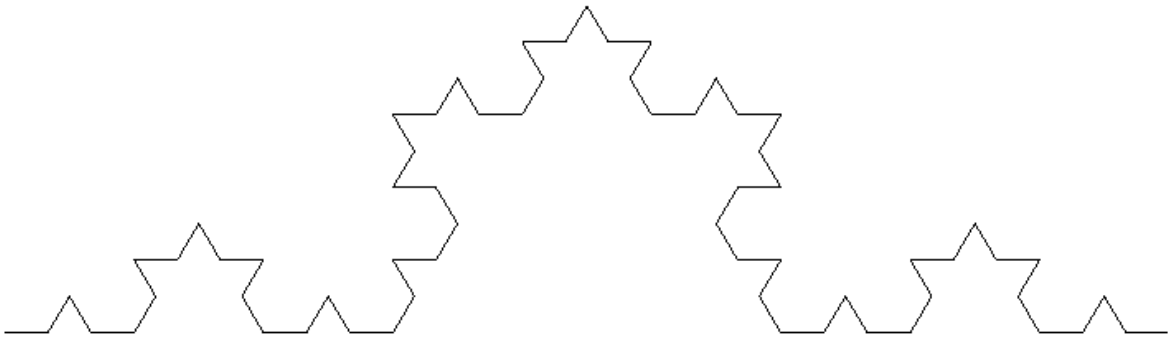
Slika 35 L-sustav - plant - dubina 3



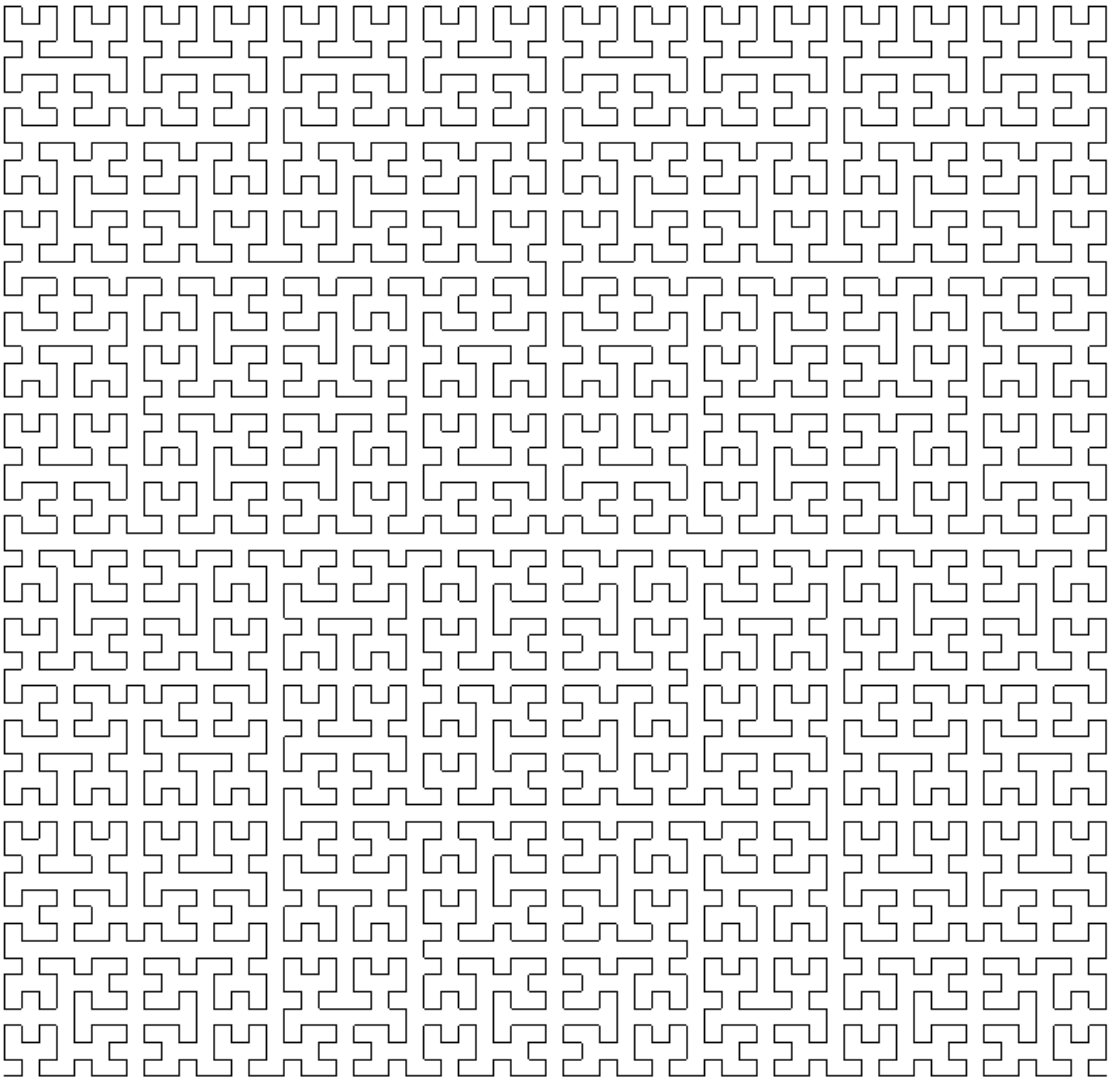
Slika 36 L-sustav - trokut SierpinskiGasket – dubina 8



Slika 37 Kochova krivulja – dubina 6



Slika 38 Kochova krivulja – dubina 3



Slika 39 L-sustavi - Hilbertova krivulja