Sprawozdanie

Projekt wirtualna kamera

Piotr Heinzelman, alb. 146703
15.X.2023

ZADANIE 1. Wirtualna kamera.

Wyobraźmy sobie że trzymamy aparat fotograficzny w dłoniach. Możemy zmienić położenie aparatu (podejść, przesunąć się w bok itd). Możemy przesunąć aparat. To znaczy możemy wykonać translację w układzie obserwatora: góra - dol, lewo-prawo, przód-tył. Możemy obrócić aparat wokół trzech osi układu współrz. związanego z obserwatorem (aparatem !). Możemy także zmienić ogniskową aparatu (zoom), to znaczy możemy zmienić kąt patrzenia. Przypominam, że czym innym jest zmiana kąta patrzenia, a czym innym podejście do obiektu (translacja). Zakładamy, że nasz aparat robi zawsze ostre zdjęcia. Zapominamy całkowicie o problemie ostrości. Proszę napisać program realizujący taką wirtualną kamerę. Na ekranie powinniśmy zobaczyć zdjęcie z aparatu, a odpowiednimi klawiszami mieć możliwość zmiany powyższych parametrów. Scena dowolna. Jak najprostsza (!!!), ale taka aby można było pokazać wszystkie zalety naszej wirtualnej kamery. (I wszystkie wymienione tutaj operacje !)

Dobrym przykładem sceny będzie np. zestaw prostopadłościanów imitujących widok ulicy z domami po lewej i prawej stronie. Złym przykładem będzie pojedyncza kula na scenie, bo trudno będzie pokazać np zbliżanie się i zmianę ogniskowej. Prostopadłościany mogą być rysowane krawędziowo bez eliminacji zasłaniania. W takim przypadku w tym projekcie trzeba narysować zbiór odcinków. Oczywiście o odpowiednich współrzędnych wynikających z operacji w przestrzeni trójwymiarowej i rzutowania.

Przedmiotem odbioru projektu jest program realizujący to zadanie.

załącznik:

pliki źródłowe - język Java v19
https://github.com/piotrHeinzelman/GrafikaKomp_ProgramowanieObiektowe

1. Instrukcja i sposób użycia

Program został napisany w języku Java, do uruchomienia niezbędne jest środowisko uruchomieniowe. Program wczyta dane z pliku tekstowego np:

```
#x,y,z=0.0,colorCode=R White|Black|Red|Green|Blue|Cyan|Magenta|grAy    # pierwsza linia nie jest brana pod
uwage. pomiedzy danymi Tabulator, kolejnosc prostopadloscianu: 1. dol-lewy-przod, 2. dol-prawy-przod \
3. dol-prawy-tyl 4. dol-lewy-tyl 5.gora...
10.0,0.0,100,0
30.0,0.0,100,0
30.0,0.0,80,0
10.0,0.0,80,0
10.0,20.0,100,0
30.0,20.0,100,0
30.0,20.0,80,0
10.0,20.0,80,0
```

Można również załadować plik z danymi wybierają z menu File->Open lub skrótem CTRL+O i pokazać plik z rozszerzeniem txt. Program działa w 2 trybach i tak - może wizualizować dane jako punkty w trzech wymiarach, choć precyzyjniej byłoby napisać we *współrzędnych jednorodnych normalizowanych* (automatycznie). sercem programu jest funkcja realizująca mnożenie macierzy M z punktem P czego wynikiem jest punkt P`.

Macierz M może być kilku rodzajów, może być to macierz translacji:

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

obrotu gdzie: s=sin(φ), c=cos (φ)

$$Ox = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c & -s & 0 \\ 0 & s & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Oy = \begin{bmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad Oz = \begin{bmatrix} c & -s & 0 & 0 \\ s & c & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

trzecia postać to macierz rzutująca na płaszczyznę „ekranu" o wartości d przechowywanej w pamięci:

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Pominąwszy działania przygotowania środowiska, konfiguracji programu, które w tym momencie są mało interesujące i w gruncie rzeczy dość trywialne, program wykonuje następujące działania:
Pierwsze - wczytanie punktów w pliku, po wczytaniu punkty przestrzeni 3d (x, y, z).

Kolejne punkty są rzutowane na płaszczyznę ekranu **P' = R*P**

```
Double _x = x*m[0][0] + y*m[1][0] + z*m[2][0] + w*m[3][0];
Double _y = x*m[0][1] + y*m[1][1] + z*m[2][1] + w*m[3][1];
Double _z = x*m[0][2] + y*m[1][2] + z*m[2][2] + w*m[3][2];
Double _w = x*m[0][3] + y*m[1][3] + z*m[2][3] + w*m[3][3];
```

oraz automatycznie zmieniane na wartości całkwite (wymóg pikselizacji, oraz normalizowane) :

```
return new Point3D( _x/_w , _y/_w , _z/_w , 1.0  );
```

w wyniku czego dostajemy listę pikseli.

Odpowiednie piksele łączymy liniami.

```
edges.add(new Edge(corners[0], corners[1]));
edges.add(new Edge(corners[1], corners[2]));
edges.add(new Edge(corners[2], corners[3]));
edges.add(new Edge(corners[3], corners[0]));

edges.add(new Edge(corners[0], corners[4]));
edges.add(new Edge(corners[1], corners[5]));
edges.add(new Edge(corners[2], corners[6]));
edges.add(new Edge(corners[3], corners[7]));

edges.add(new Edge(corners[4], corners[5]));
edges.add(new Edge(corners[5], corners[6]));
edges.add(new Edge(corners[6], corners[7]));
edges.add(new Edge(corners[7], corners[4]));
```

i rysujemy zwykłymi funkcjami obiektu JFrame.

Po wybraniu kierunku dokonuje się translacja punktów Punktów3d w pamięci do noewgo położenia, po naciśnięciu klawiszy strzałek na klawiaturze Macierzą przez którą przemnożymy punkty będzie macierz translacji, po naciścnięciu **CTRL+strzałki** macierzą będzie macież obrotu wokół OX i OZ, natomiast **ALT+strzałki prawo** lub **lewo** wygenerują macierz obrotu OY która będzie pomnożona przez punkty.

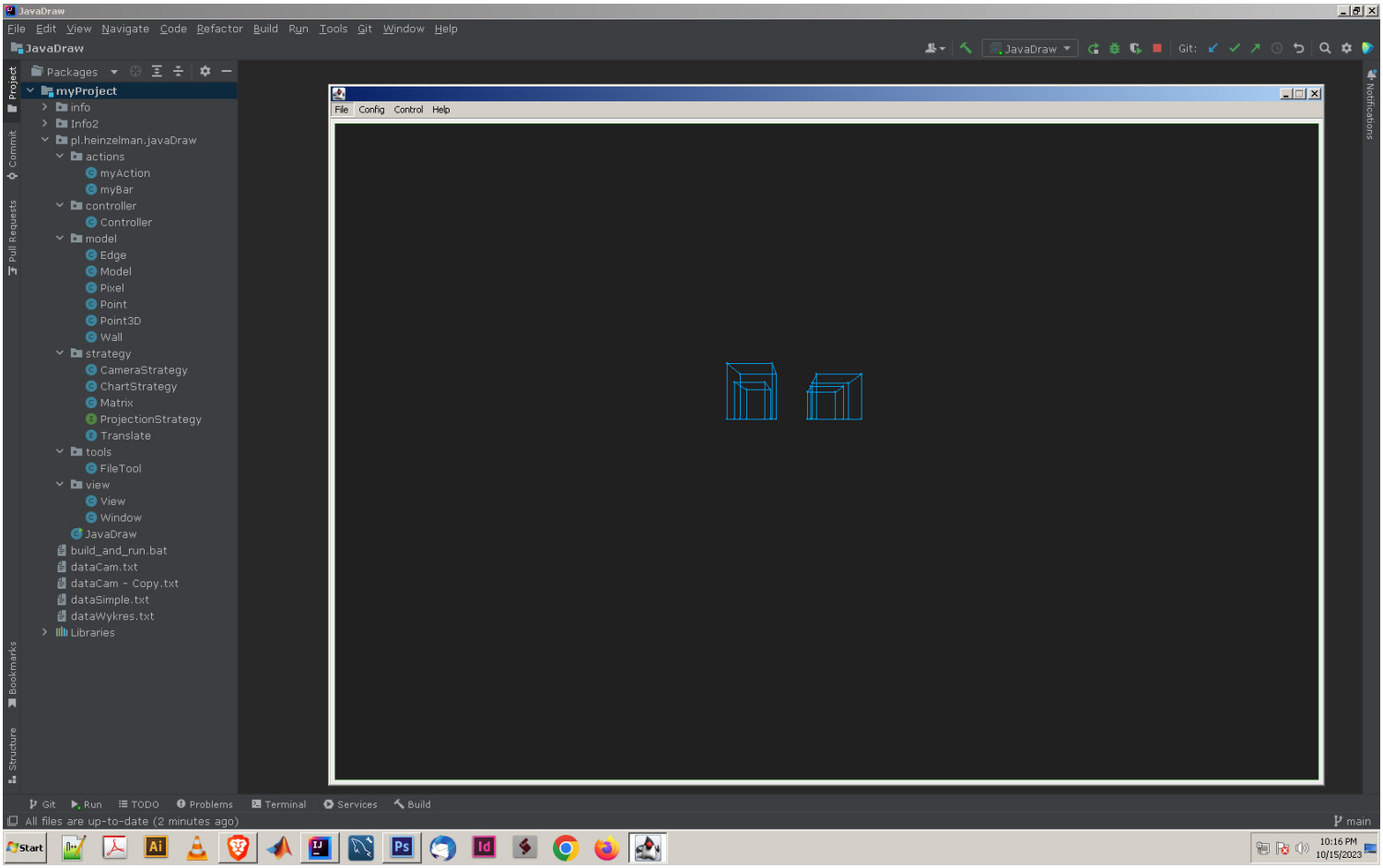
użycie „**CTRL +**” lub „**CTRL -**” powoduje zmianę parametru d, odpowiednio zmniejszenie lub zwiekszenie o 10%, co daje w wyniku wrażenie podobne do „powiększenia” lub „zmniejszenia”.

Wynik zostanie zrzutowany na płaszczyznę ekranu tak jak poprzednio i wyświetlony.

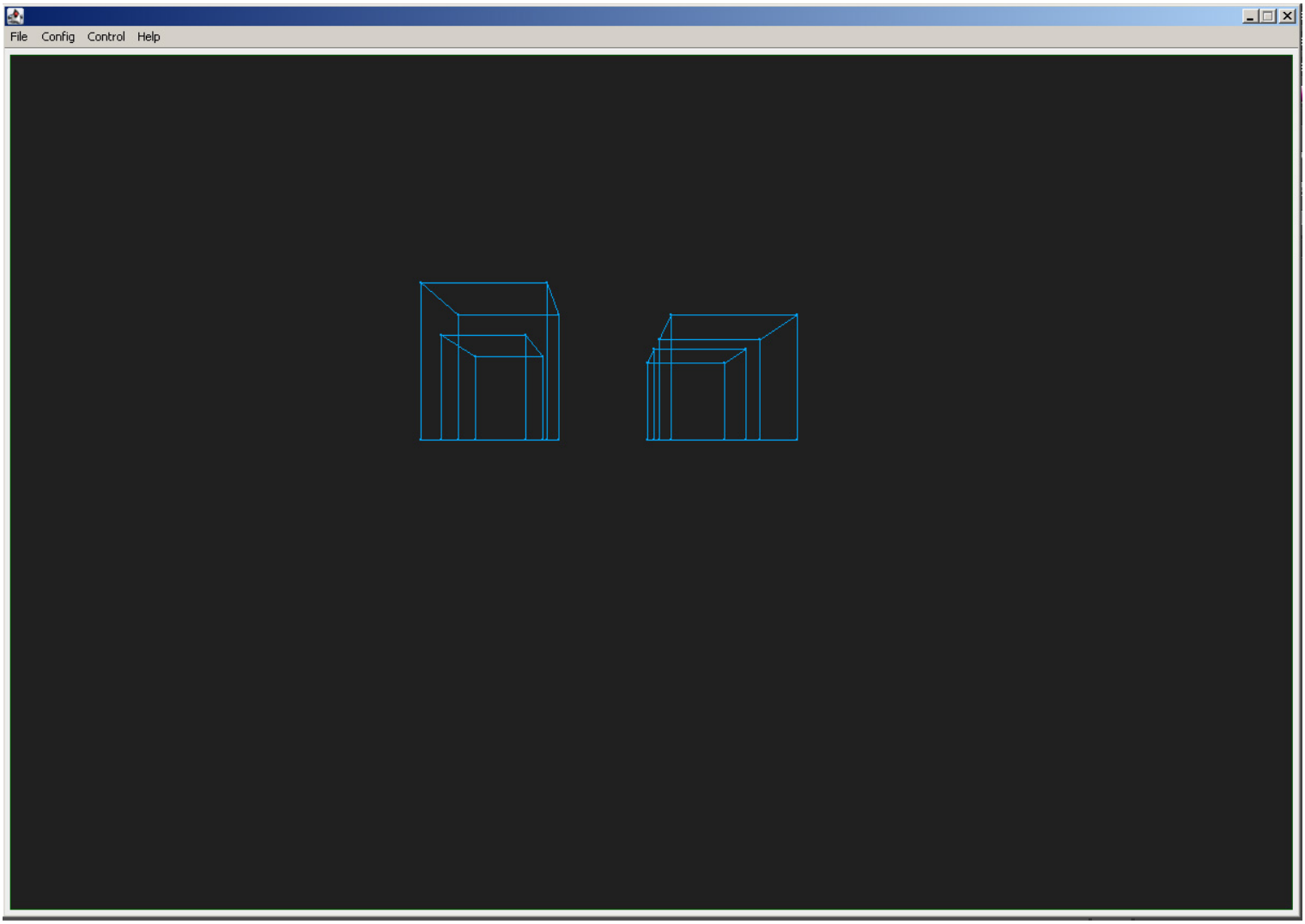
Najistotniejszym elementem jest użycie tylko jednej funkcji translacji, co powoduje że program jest bardzo elegancki, a funkcje liczące są odseparowane od pozostałych i zdefinoiwane w strategii Kamera.

ponadto program może użyć innej strategii wyświetlania - strategii wykres i zrzutować punkty 2d wraz z osią na płaszczyznę ekranu, ale to już inna historia.
Dodatkowo można w programie zmieniać grubość linii obrysu oraz kolor linii wybierając parametry z menu.

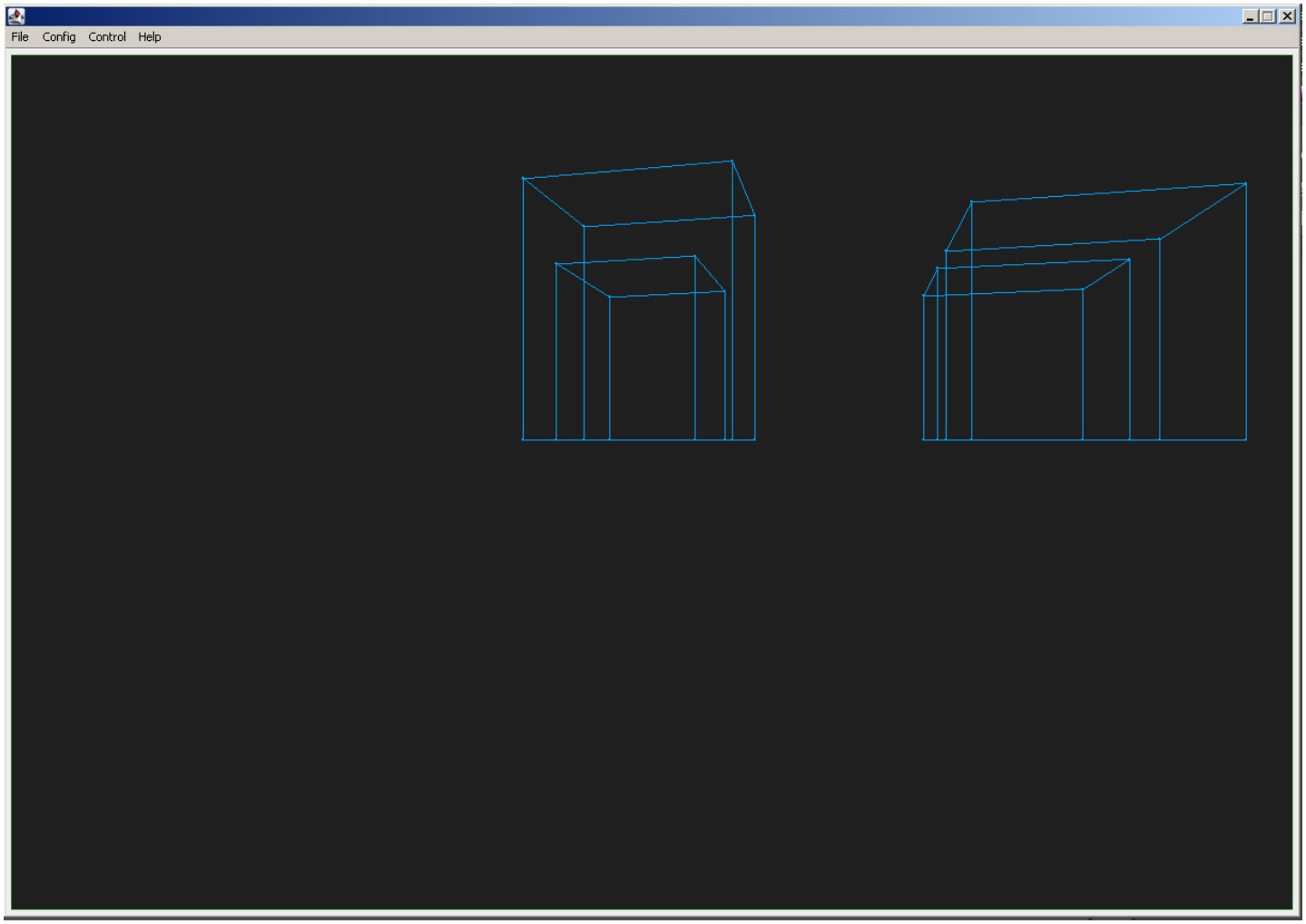
F1, F2, F3 - wyświetlenie/ukrycie wierzchołków/krawędzi/ścian.



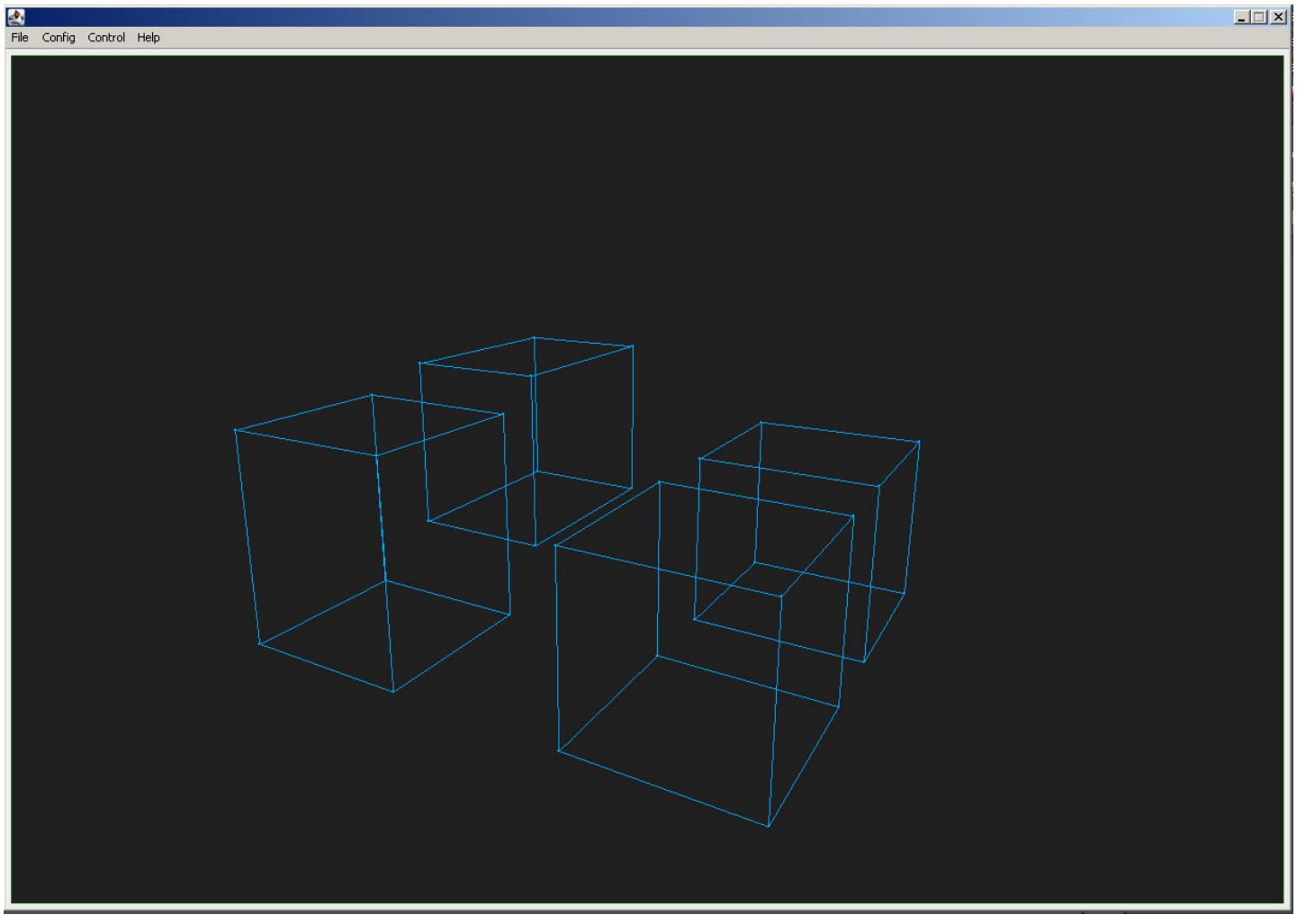
start (po załadowaniu danych)



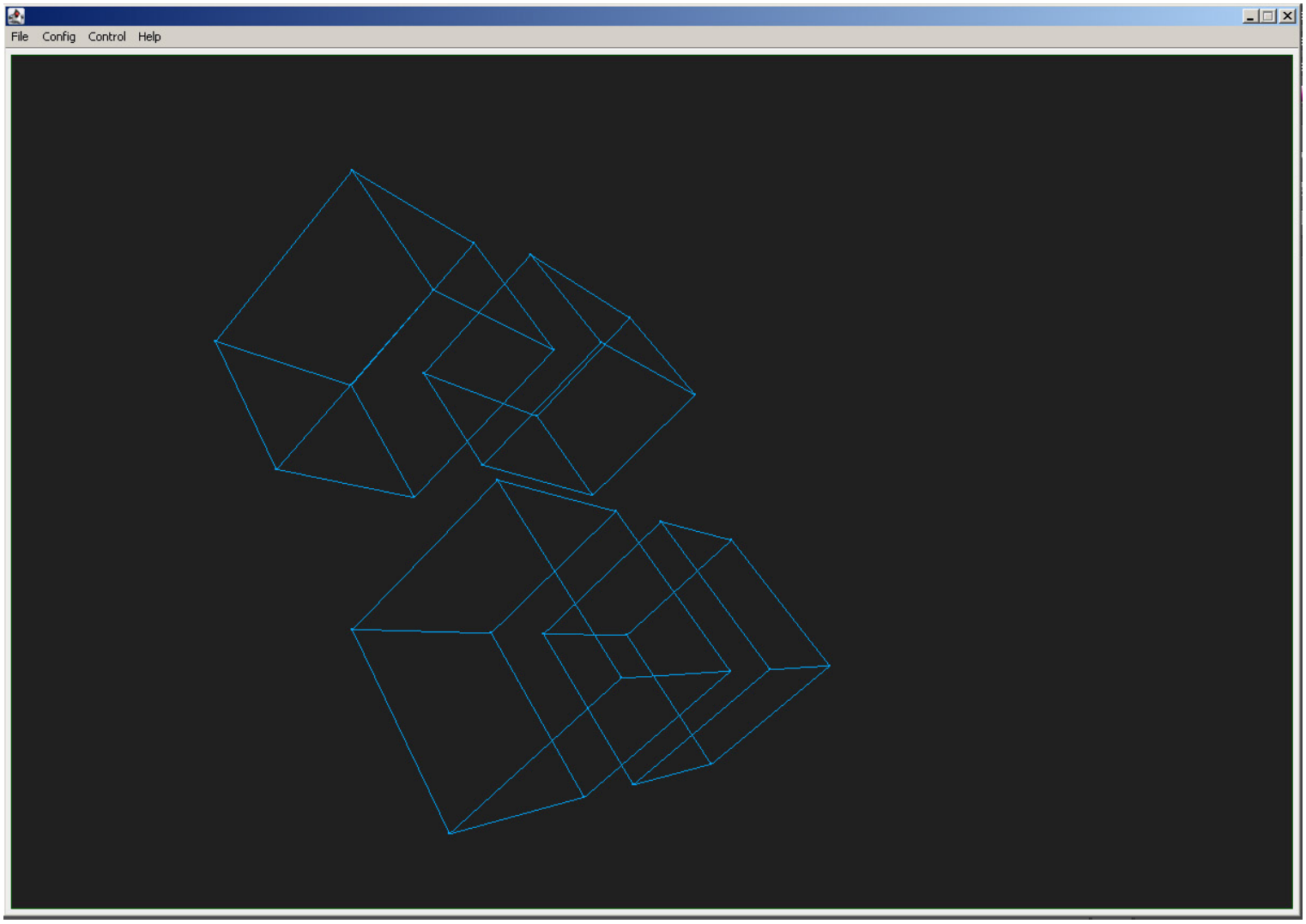
control i +



->



control i strzałki góra i prawo



alt i trzałka w prawo

ZADANIE 2. **Implementacja algorytmu eliminacji elementów zasłoniętych**.

Scena zawiera zbiór obiektów wielościennych (więcej niż 2 obiekty).

Algorytm dowolny, ale

NIE Ray Tracing/Ray Casting (lub inna odmiana RT)

NIE Z-bufor.

Przedmiotem odbioru jest program realizujący to zadanie, plus zestaw obrazków pokazujących poprawność pracy algorytmu dla charakterystycznych danych (zależnych od wybranego algorytmu).

2. Opis

W projekcie zaimplementowałem algorytm malarski, który polega na malowaniu wszystkich elementów sceny w kolejności od najdalszego do najbliższego, algorytm ten jest prosty w implementacji natomiast jest niewydajny jeśli chodzi o szebkość działania. Do celów akademickich nadaje się bardzo dobrze.

Aby z niego skorzystać należy posortować elementy sceny - zakładam że tylko ściany - w kolejności od najdalszego do najbliższego. Bazujemy na punktach przestrzeni (przed rzutowaniem) nie na punktach zrzutowanych (pikselach lub odpowiednikach).

Do sortowania używam może nie najłatwiejszego - ale eleganckiego rozwiązania czyli drzewa BSP - Binary Split Partitioning.

Proces przebiega następująco
- wybieramy jedną ścianę ze zbioru i umieszczamy na szczycie drzewa.
- pozostałe ściany dzielimy na 2 grupy:
  a) jeśli ściany leżą po „górnej" stronie ściany - przechodzą do prawej części poddrzewa,
  b) analogicznie, jeśli leżą po „spodniej" stronie ściany - przechodzą do lewej części poddrzewa.
  c) jeśli ściana leży częściowo po „górnej" i „spodniej" części drzewa - dzielimy ją płaszczyzną na 2 ściany, które leżą jedna po „górnej" a druga po „spodniej" stronie, i umieszczamy je we właściwym poddrzewie.
- następnie przystępujemy do dzielenia grup obu poddrzew - dla każdej osobno - wybieramy jedną ścianę i w stosunku do niej dzielimy wszystkie ściany poddrzewa.

Jeśli obserwator znajduje się po stronie „górnej" pierwszej ściany - ściany czytamy wspak, jeśli po spodniej ściany odczytujemy w kolejności in-order.

Aby zrealizować matematycznie to zadanie wykorzystałem:
- metodę **obliczająca płaszczyznę** na której leży ściana;
- metodę **obliczającą wektor normalny** do płaszczyzny;
- metodę **obliczającą po której stronie leży punkt względem płaszczyzny**, czyli obliczającą współczynnik przez jaki należy pomnożyć wektor normalny zaczepiony na płaszczyźnie aby koniec znalazł się w badanym punkcie. Jeśli współczynnik jest dodatni punkt leży nad, a jeśli ujemny - leży pod.
- metodę **obliczająca punkt wspólny odcinka i płaszczyzny** - niezbędny do podziału ściany na 2 części, czyli „górną" i „spodnią"
- strukturę **drzewa BSP** na którym można przechować zbiór ścian przed ich renderowaniem (rzutowaniem).

poniżej ciekawsze fragmenty kodu:

metoda **obliczająca płaszczyznę**

```
   public Plane ( Point3D one, Point3D two, Point3D three ){
        Vector3D twoOne = new Vector3D( two, one );
        Vector3D twoThree = new Vector3D( two, three );
        Vector3D normal = Vector3D.getNormal( twoOne , twoThree );
        Plane P = new Plane ( normal , two );
        this.A = P.A;
        this.B = P.B;
        this.C = P.C;
        this.D = P.D;
    }
```

metoda **wektor normalny**:

```
public static Vector3D getNormal( Vector3D a, Vector3D b ){
        return new Vector3D( a.y*b.z - a.z* b.y , a.z* b.x - a.x* b.z , a.x*b.y - a.y*b.x  );
    }
```

metoda **obliczającą po której stronie leży punkt względem płaszczyzy**

```
  public int checkSideIsAtRightSide( Point3D p2 ){
        /* P2 o
              |  l= x=x0+ta / y=y0+tb / z=zo+tc  po - punkt prostej  wektor wzd. v=(a,b,c)
        ------|------
           P1 o        P1 nalezy do prostej,spełnia P1x = P2x +tA /
                                                    P1y = P2y +tB / P1z = P2z +tC
                       P1 nalezy do płaszczyzny spełnia Ax+By+Cz+D=0
                       zatem A(x2+tA) +B(y2+tB) +C(z2+tC) +D =0  nie znamy tylko t więc...
                       t*(A²+B²+C²)=D-Ax2-By2-Cy2
                       t= D-Ax2-By2-Cy2 / (A²+B²+C²);
     */

        Double ABC=A*p2.getX() +B*p2.getY() +C*p2.getZ();
        Double D_ABC = (D-ABC)/(A*A+B*B+C*C);
        if ( D_ABC  >  0.001 ) return +1;  // ponad plaszczyzna
        if ( D_ABC  < -0.001 ) return -1;  // pod plaszczyzna
        return 0;  // na tej samej plaszczyznie, uwzgledniajac bledy notacji zmiennoprzecinkowej
       }
```

metoda **obliczająca punkt wspólny odcinka i płaszczyzny**

```
   public Point3D getPointOfPlane( Point3D p0, Point3D p1  ) {
        Double dx=p1.getX()-p0.getX();
        Double dy=p1.getY()-p0.getY();
        Double dz=p1.getZ()-p0.getZ();

        Double t= ( -A*p0.getX() -B*p0.getY() -C* p0.getZ() -D )/( A*dx+B*dy+C*dz );
        return new Point3D( p0.getX()+t*dx, p0.getY()+t*dy,p0.getZ()+t*dz );
    }
```

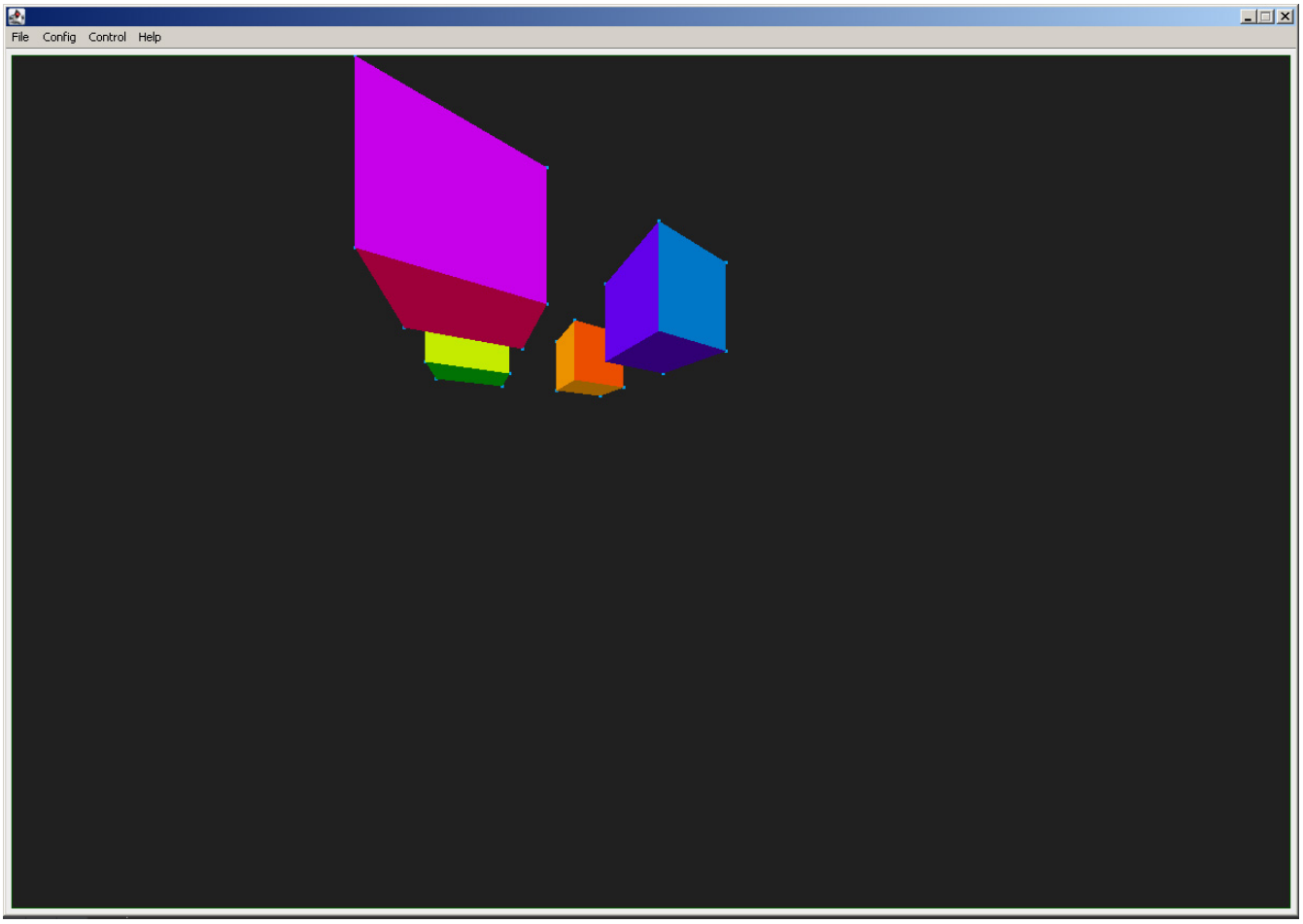
metoda

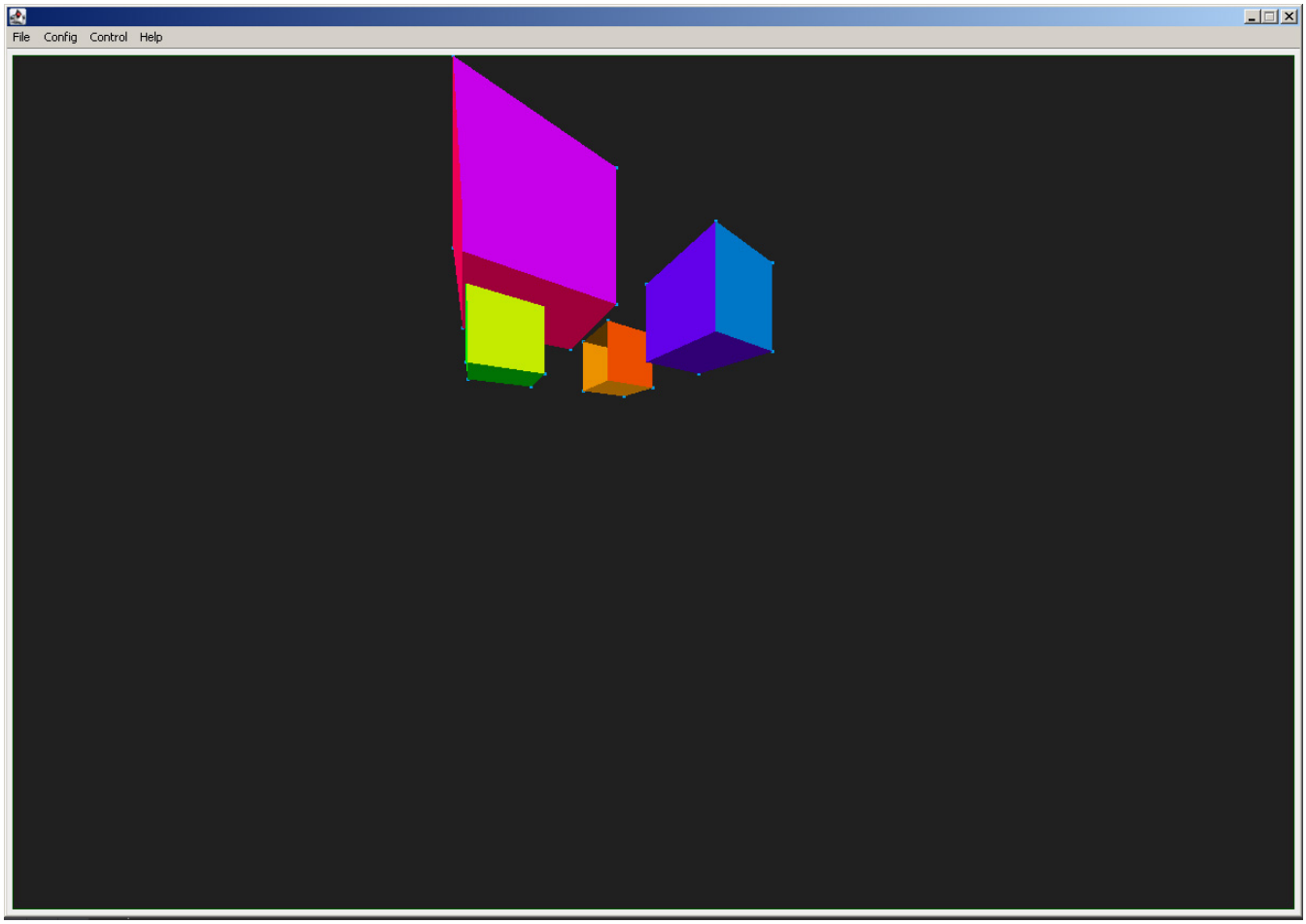
```
public class Plane {
    // Ax+By+Cz+D=0  Vnorm(a,b,c)

    private Double A;
    private Double B;
    private Double C;
    private Double D;
```

metoda **dzieląca ścianę**

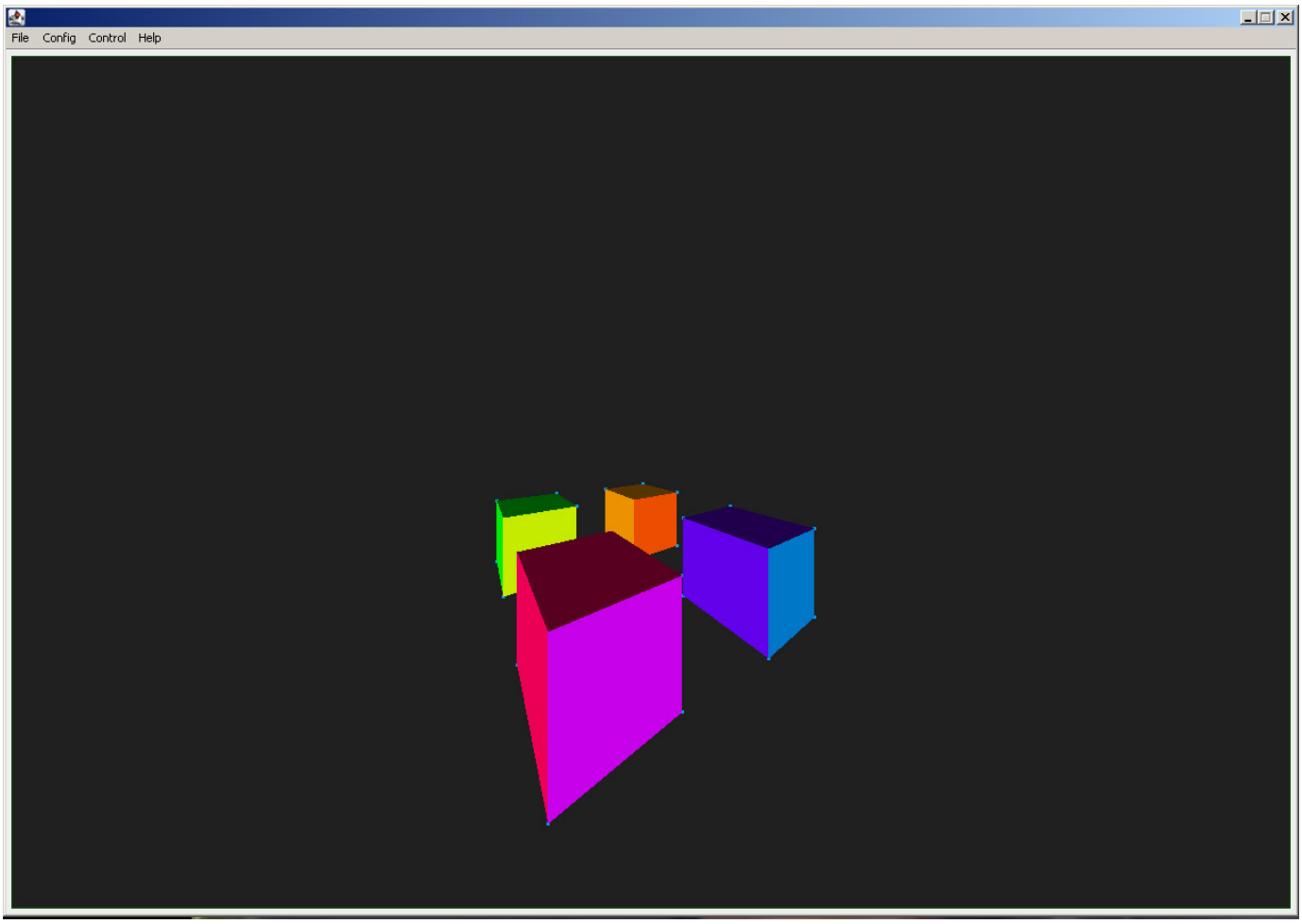
```
public Wall3D[] splitByPlane(Plane plane ){
        int[] signs=new int[4];
        signs[0]= plane.checkSideIsAtRightSide(one);
        signs[1]= plane.checkSideIsAtRightSide(two);
        signs[2]= plane.checkSideIsAtRightSide(three);
        signs[3]= plane.checkSideIsAtRightSide(four);

        Wall3D over = new Wall3D( this.one,this.two,this.three,this.four,this.color );
        Wall3D under = new Wall3D( this.one,this.two,this.three,this.four,this.color );

        Point3D one_two=null;
        Point3D two_three=null;
        Point3D three_four=null;
        Point3D four_one=null;
        if ( signs[0]!=signs[1] ){  one_two =  plane.getPointOfPlane( one   , two   ); }
        if ( signs[1]!=signs[2] ){  two_three =  plane.getPointOfPlane( two   , three ); }
        if ( signs[2]!=signs[3] ){  three_four =  plane.getPointOfPlane( three, four  ); }
        if ( signs[3]!=signs[1] ){  four_one =  plane.getPointOfPlane( four , one   ); }

        Wall3D up=null; Wall3D down=null;
        if ( one_two!=null && three_four!=null ){
                if ( signs[0]==1 ) { up=over; down=under;  } else { up=under; down=over; }
                up.two=one_two; down.one=one_two;
                up.three=three_four; down.four=three_four;
                return new Wall3D[]{ up , down };
        }
        if ( two_three!=null && four_one!=null ){
            if ( signs[0]==1 ) { up=over; down=under;  } else { up=under; down=over; }
            up.three=two_three; down.two=two_three;
            up.four=four_one;   down.one=four_one;
            return new Wall3D[]{ up , down };
        }
        return null;
    }
```

widok z renderowanymi ścianami od spodu

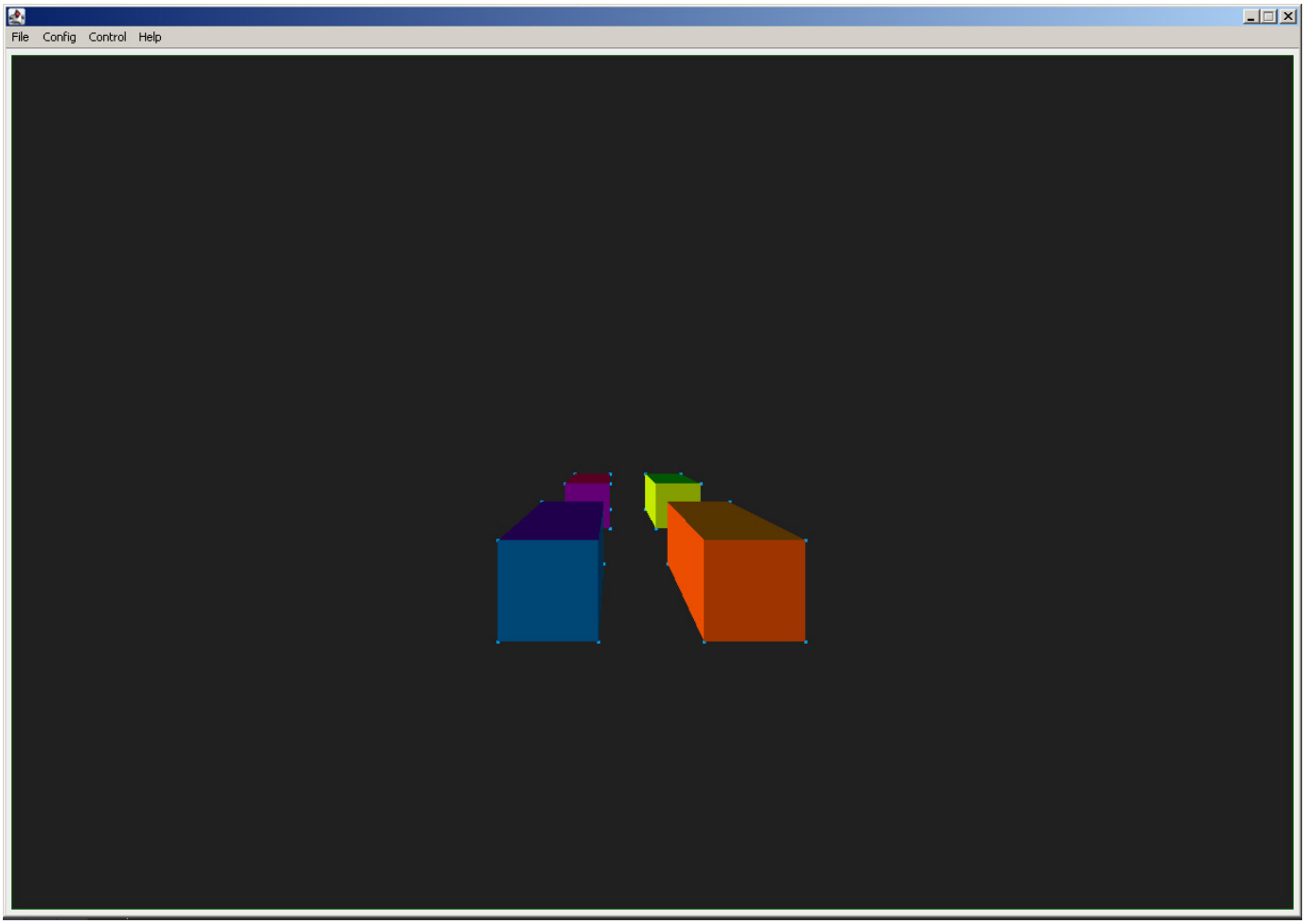




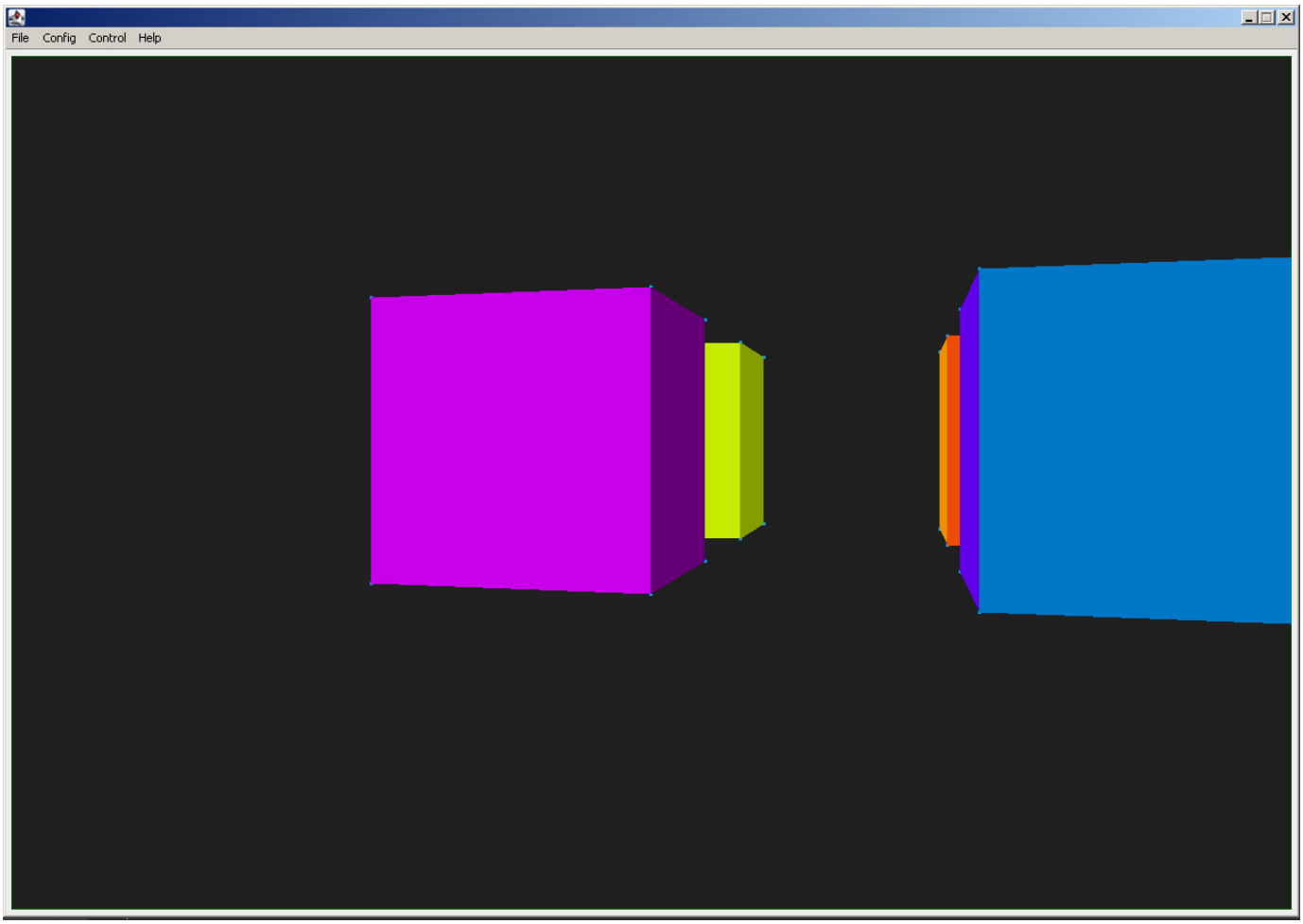
widok po przesunięciu o krok - czasem zdarzają się jakieś błędy w obliczeniach, może to kwestia mojego błędu w implementacji lub wina sprzętu lub języka.



widok z drugiej strony z góry



widok z drugiej strony



widok po podejściu bliżej (zmiana parametru d - czyli odległości rzutni)