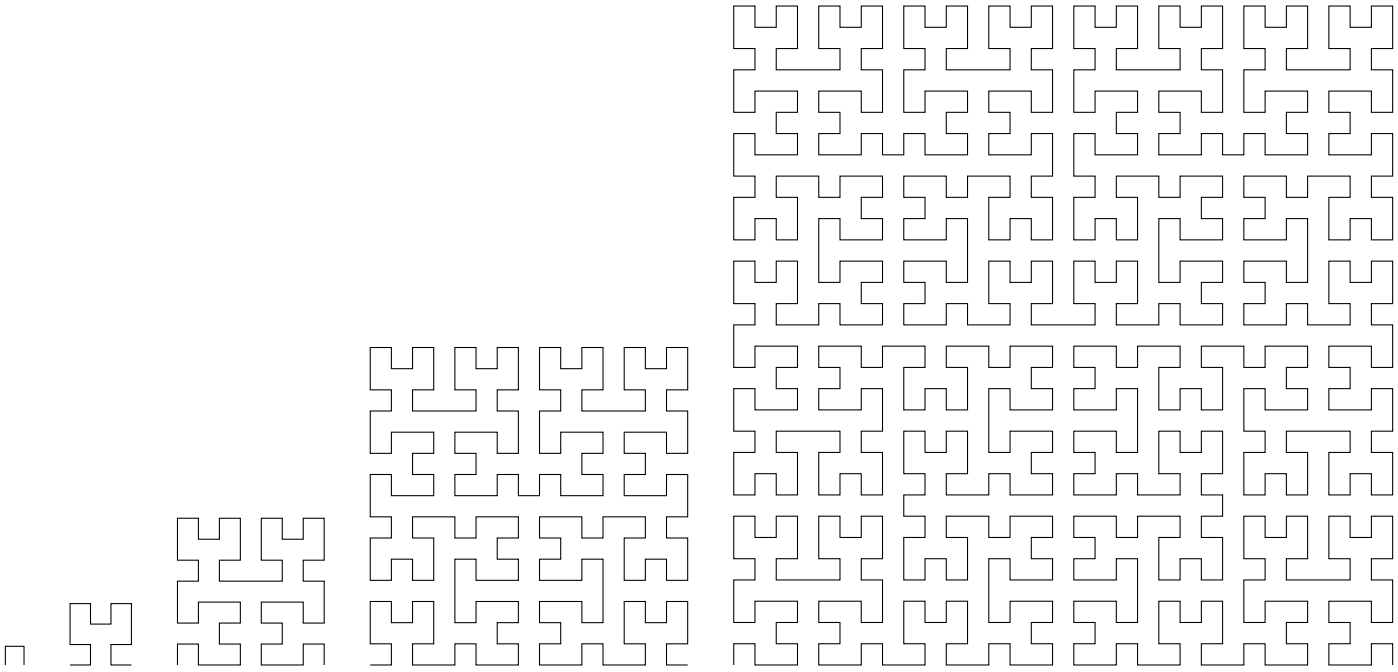
Metody programowania 2016

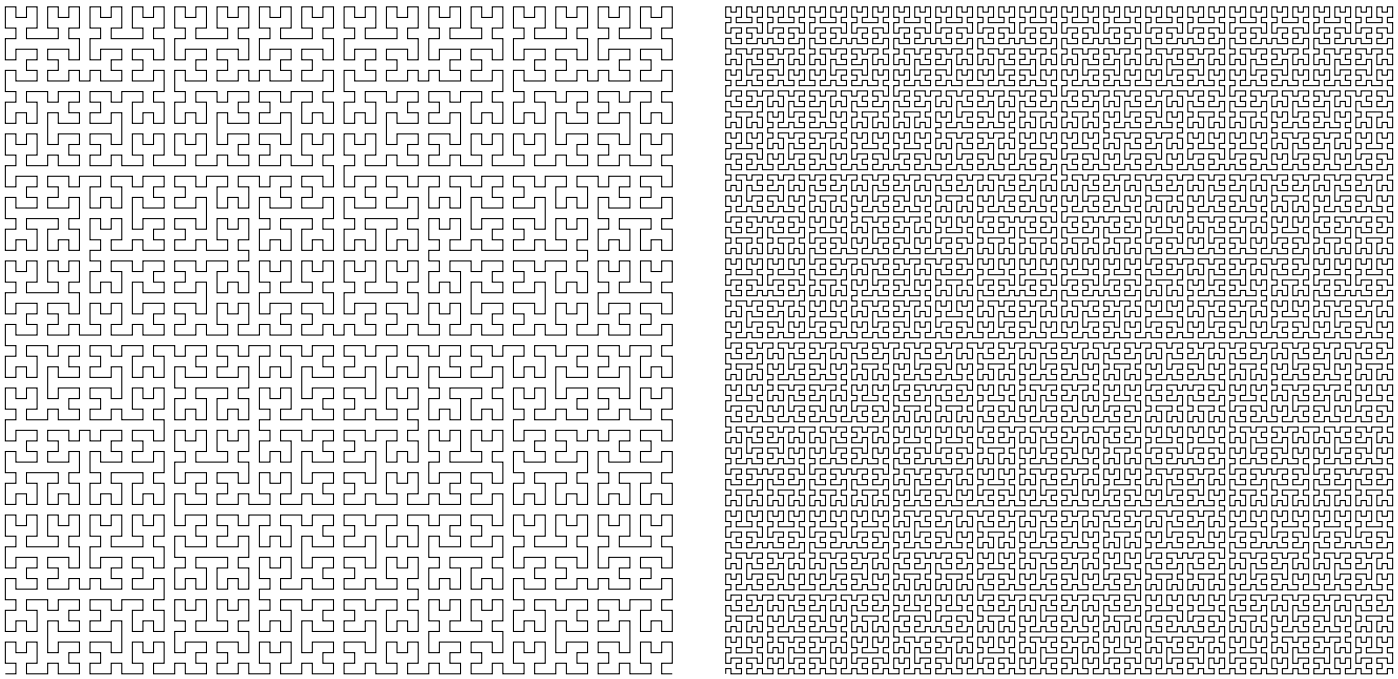
Lista zadań na pracownię nr 1

Termin zgłaszania w KNO: 14 marca 2016, godzina 6:00 AM CET

Krzywe Hilberta to ciąg łamanych zadany prostą zależnością rekurencyjną. O n-tym elemencie tego ciągu mówimy, że jest *rzędu* n ($n \geq 1$). Krzywe rzędów 1–5 złożone z odcinków długości 8pt[1] wyglądają następująco:

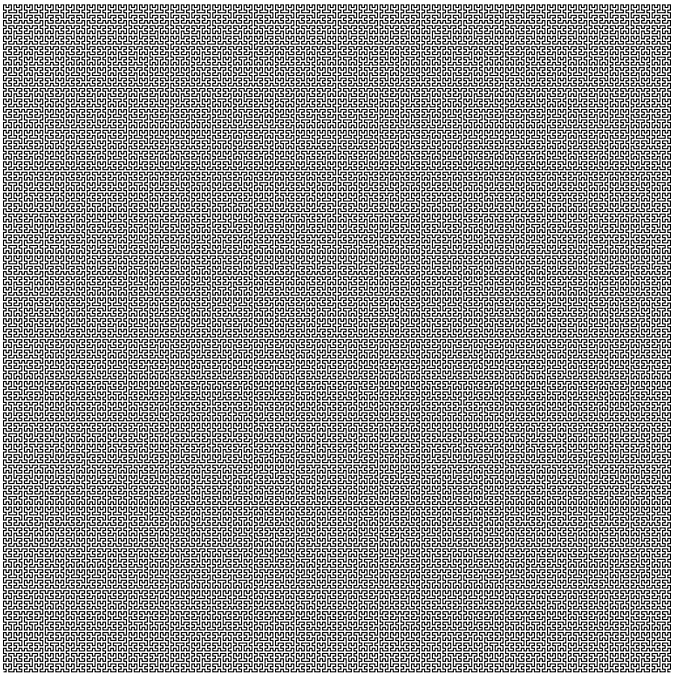


Jeśli przyjmiemy, że długość odcinka krzywej wynosi 1, to krzywa rzędu n mieści się w kwadracie o boku $2^n - 1$. Zwykle przeskalowuje się je tak, by mieściły się w kwadracie $[0,1] \times [0,1]$. Wówczas każda krzywa zaczyna się w punkcie $(0,0)$ i kończy w punkcie $(1,0)$, a długość odcinka krzywej rzędu n wynosi $1/(2^n - 1)$. Krzywe rzędu 6 i 7 narysowane w układzie współrzędnych, w którym jednostka długości jest równa 250pt wyglądają wówczas następująco:



[1] W typografii długość wyraża się przeważnie w tzw. *punktach typograficznych* równych 1/72 części cala.

Dla tych, którzy jeszcze nie zauważyli prostej zależności rekurencyjnej oto kolejny — ósmy element ciągu (wygenerowaliśmy też krzywą Hilberta dziewiątego rzędu, ale zawiera już zbyt małe detale, by było cokolwiek widać na wydruku):



Krzywe Hilberta można opisać w postaci parametrycznej, tj. za pomocą ciągłych funkcji odwzorowujących odcinek $[0,1]$ w kwadrat $[0,1]\times[0,1]$. Okazuje się, że ciąg tych funkcji $f_n : [0,1] \to [0,1]\times[0,1]$ jest zbieżny jednostajnie do pewnej funkcji $f : [0,1] \to [0,1] \times [0,1]$. Funkcja f jest zatem ciągła. Okazuje się ponadto, że f jest „na"! Z kursu *Logiki dla informatyków* wiemy, że istnieje funkcja z odcinka $[0,1]$ na kwadrat $[0,1] \times [0,1]$. Okazuje się, że istnieje taka *ciągła* funkcja! Istnieje więc krzywa ciągła zaczynająca się w punkcie $(0,0)$ i kończąca w punkcie $(1,0)$ która przechodzi przez *wszystkie* punkty kwadratu $[0,1] \times [0,1]$! Ale zadanie nie jest o tym.

Elegancka zależność rekurencyjna i atrakcyjna forma wizualna krzywych Hilberta powoduje, że napisanie programu rysującego te krzywe (albo inne podobne, jak np. Moore'a, Peano, Sierpińskiego czy Gospera[2]) jest wdzięcznym zadaniem dla początkujących programistów i z powodzeniem mogłoby się znaleźć (i nie raz bywało) na liście zadań z kursu *ANSI C* lub *Wstępu do programowania*. Pochwalimy się tu naszym rozwiązaniem w języku Haskell (nie zakładamy przy tym, że studenci umieją już w nim programować — nie powinno być to przeszkodą w zrozumieniu algorytmu).

Rysowanie krzywych na płaszczyźnie wygodnie opisuje się korzystając z modelu tzw. *grafiki żółwia*. Żółw znajduje się w pewnym punkcie płaszczyzny. Jego głowa wskazuje pewien kierunek. Żółw może się obracać o ustalony kąt oraz wykonywać kroki określonej długości w kierunku, który wskazuje jego głowa. Żółw posiada ponadto pióro, które może być uniesione (wtedy nie pozostawia śladu na płaszczyźnie) lub opuszczone (i wówczas rysuje na płaszczyźnie kreskę, gdy żółw się przemieszcza). Możemy wydawać żółwiowi polecenia obracania się, przemieszczania, unoszenia i opuszczania pióra.

W naszym przypadku żółw rysuje krzywą ciągłą, jego pióro będzie więc zawsze opuszczone. Krzywe Hilberta składają się jedynie z pionowych i poziomych odcinków długości 1[3], głowa żółwia będzie więc skierowana wyłącznie na cztery strony świata, żółw będzie się obracał zawsze o 90° zgodnie lub przeciwnie do ruchu wskazówek zegara (tj. w prawo bądź w lewo) i wykonywał wyłącznie kroki długości 1.

Tak uproszczoną grafikę żółwia możemy zaprogramować w Haskellu następująco:

```
data Direction = East | South | West | North deriving (Enum, Show)
data Orientation = CounterClockwise | Clockwise deriving (Enum, Show)
changeDirection :: Orientation -> Direction -> Direction
changeDirection rotation direction = toEnum $ (fromEnum direction + 2 * fromEnum rotation - 1) `mod` 4
swapOrientation :: Orientation -> Orientation
swapOrientation Clockwise = CounterClockwise
swapOrientation CounterClockwise = Clockwise

type Point = (Int,Int)
type Vector = (Int,Int)
movePoint :: Point -> Vector -> Point
movePoint (x1,y1) (x2,y2) = (x1+x2,y1+y2)
dirToVector :: Direction -> Vector
dirToVector = ([(1,0), (0,-1), (-1,0), (0,1)] !!) . fromEnum
```

[2] Ta ostatnia jest wyjątkowo piękna!

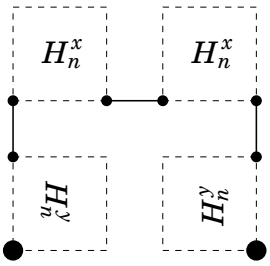
[3] Wygodnie jest tak przyjąć, a następnie przeskalować powstały obrazek.

```
stepPoint :: Point -> Direction -> Point
stepPoint v = movePoint v . dirToVector
```

Typ `Direction` ma cztery wartości przedstawiające strony świata, a typ `Orientation` określa kierunek skrętu. Dla podanego kierunku skrętu i strony świata funkcja `changeDirection` wyznacza stronę świata w którą będzie zwrócony żółw po wykonaniu skrętu. Funkcja `swapOriention` wyznacza przeciwny do podanego kierunek skrętu.

Punkty (typ `Point`) i wektory (typ `Vector`) reprezentujemy w postaci par liczb całkowitych. Jedyna potrzebna żółwiowi funkcja `stepPoint` wyznacza punkt do którego przejdzie żółw jeśli znajduje się w podanym punkcie i jest zwrócony w podaną stronę świata. Ta funkcja korzysta z dwóch pomocniczych: `movePoint` wyznacza obraz podanego punktu w przesunięciu o podany wektor, a `dirToVector` ujawnia wektor o jaki trzeba przesunąć punkt jeśli żółw chce wykonać krok w podanym kierunku.

Zauważmy, że aby narysować krzywą rzędu $n+1$ (oznaczmy ją H^x_{n+1}, gdzie x jest orientacją), musimy narysować kolejno cztery kopie krzywej rzędu n wykonując pomiędzy nimi pojedyncze kroki w odpowiednich kierunkach:



Zauważmy że pierwszą i czwartą krzywą (oznaczone na rysunku symbolem H^y_n) rysujemy od końca do początku, tj. wykonując wszystkie skręty w przeciwną stronę, niż w przypadku drugiej i trzeciej kopii (oznaczonych na rysunku symbolem H^x_n). Aby poradzić sobie z tym problemem najprościej przyjąć, że sam żółw ma ustaloną orientację i zależnie od niej wykonuje wszystkie czynności normalnie (gdy jest prawoskrętny) lub na opak (gdy jest lewoskrętny).

Możemy już opisać pracę żółwia. Aby narysować krzywą H^x_0, żółw nic nie musi robić. Żółw rysuje krzywą H^x_{n+1} (gdzie x określa „skrętność" żółwia) następująco:

- skręca w stronę y, gdzie y jest kierunkiem przeciwnym do x,
- rysuje rekurencyjnie krzywą H^y_n,
- wykonuje krok do przodu,
- skręca w stronę x,
- rysuje rekurencyjnie krzywą H^x_n,
- wykonuje krok do przodu,
- rysuje rekurencyjnie krzywą H^x_n,
- skręca w stronę x,
- wykonuje krok do przodu,
- rysuje rekurencyjnie krzywą H^y_n,
- skręca w stronę y.

W Haskellu pracę żółwia możemy opisać następująco:

```
hilbertRec :: Int -> Orientation -> StateComp State ()
hilbertRec 0 orientation = return ()
hilbertRec order orientation = do
   turn contra
   hilbertRec (order-1) contra
   step
   turn orientation
   hilbertRec (order-1) orientation
   step
   hilbertRec (order-1) orientation
   turn orientation
   step
   hilbertRec (order-1) contra
   turn contra
      where contra = swapOrientation orientation
```

Funkcja `hilbertRec` dla podanego rzędu krzywej i podanej skrętności żółwia buduje obliczenie, którego wynikiem będzie lista wierzchołków łamanej, którą narysuje żółw. To obliczenie zostaje wykonane w funkcji `hilbertCurve2D`, która dla podanego rzędu krzywej, punktu początkowego, skrętności żółwia i jego początkowego ustawienia względem stron świat tworzy listę punktów łamanej:

```
hilbertCurve2D :: Int -> Point -> Orientation -> Direction -> [Point]
hilbertCurve2D order start orientation direction =
   reverse . snd . snd $ runState (hilbertRec order orientation) (direction, [start])
```

Należałoby jeszcze opowiedzieć, jak symulujemy w Haskellu obliczenia ze stanem za pomocą monad:

```
newtype StateComp state result = StateComp { runState :: state -> (result, state) }
instance Monad (StateComp st) where
   return = StateComp . (,)
   (StateComp comp) >>= f = StateComp $ uncurry (runState . f) . comp

type State = (Direction, [Point])
step :: StateComp State ()
step = StateComp $ \ (dir, moves@(pos:_)) -> ((), (dir, stepPoint pos dir : moves))
turn :: Orientation -> StateComp State ()
turn rot = StateComp $ \ (dir, moves) -> ((), (changeDirection rot dir, moves))
```

ale odłożymy to na potem (będzie o tym sporo na późniejszych zajęciach), bo zadanie jest nie o tym.

Liczba odcinków d_n łamanej H_n^x podlega następującej zależności rekurencyjnej:

$$\begin{cases} d_0 &= 0 \\ d_{n+1} &= 4 \cdot d_n + 3 \end{cases}$$

Rozwiązaniem tej zależności rekurencyjnej jest $d_n = 2^{2n} - 1$ (mała powtórka z *Logiki dla informatyków*: udowodnij słuszność tego wzoru przez indukcję względem n). Liczba ta rośnie bardzo szybko (np. $d_8 = 65535$). Przedstawianie krzywej Hilberta w postaci listy współrzędnych jest więc niezbyt praktyczne. Wolelibyśmy wygenerować odpowiedni rysunek.

Najprościej użyć w tym celu języka Postscript. Jest to język programowania przeznaczony do opisu grafiki wektorowej w sposób podobny do grafiki żółwia. Strona w Postscripcie to prostokąt opisany za pomocą współrzędnych dwóch przeciwległych narożników. Np. wiersz

```
%%BoundingBox: 0 0 500 500
```

oznacza, że będziemy rysować w kwadracie o wymiarach 500×500 pt, a początek układu współrzędnych będzie się znajdował w lewym dolnym narożniku strony. Początkowo znajduje się tam też żółw. Możemy żółwiowi wydawać polecenia:

- `newpath` – rozpocznij kreślenie nowej krzywej.
- x y `moveto` – przejdź do punktu o współrzędnych (x, y) z uniesionym piórem (współrzędne wyrażamy w punktach typograficznych w postaci liczb zmiennopozycyjnych).
- x y `lineto` – przejdź do punktu o współrzędnych (x, y) z opuszczonym piórem (kreśląc odcinek).
- w `setlinewidth` – ustal grubość linii (domyślnie jest 1pt, my wolimy nieco cieńsze linie).
- `stroke` – zrealizuj kreślenie linii. Postscript bazuje na języku FORTH, w którym cała praca programu obraca się wokół stosu. Tu jest podobnie: polecenia `moveto` i `lineto` nie są wykonywane, tylko odkładane na stosie. Dopiero polecenie `stroke` powoduje zdjęcie ze stosu i wykonanie wszystkich poleceń aż do polecenia `newpath`. Ze stosem związana jest też odwrotna notacja polska, w której zapisuje się postscriptowe programy (dlatego argumenty poleceń piszemy *przed* nimi).
- `showpage` – powoduje zakończenie tworzenia strony.

Do kompletu trzeba jeszcze dodać nagłówek i stopkę pliku (które w naszym przypadku zawsze mogą być takie same). Oto przykład kompletnego pliku w Postscripcie:

```
%!PS-Adobe-2.0 EPSF-2.0
%%BoundingBox: 0 0 500 500
newpath
1.00 1.00 moveto
1.00 499.00 lineto
499.00 499.00 lineto
499.00 1.00 lineto
.4 setlinewidth
```

```
stroke
showpage
%%Trailer
%EOF
```

Powyższy obrazek zawiera krzywą Hilberta rzędu 1. Został wygenerowany za pomocą następującego programu:

```
genPostscript :: [Int] -> String
genPostscript [order, size, margin] =
   "%!PS-Adobe-2.0 EPSF-2.0\n%%BoundingBox: 0 0 " ++ show size ++ " " ++ show size ++ "\nnewpath\n"
   ++ genCommand "moveto" start ++ concatMap (genCommand "lineto") moves
   ++ ".4 setlinewidth\nstroke\nshowpage\n%%Trailer\n%EOF\n" where
      start:moves = hilbertCurve2D order (0,0) Clockwise East
      genCommand command (x,y) = showCoord x ++ " " ++ showCoord y ++ " " ++ command ++ "\n"
      showCoord c = show (sc `div` 100) ++ "." ++ show ((sc `mod` 100) `div` 10) ++ show (sc `mod` 10)
                    where sc = (100 * c * drawSize) `div` (2^order-1) + 100 * margin
      drawSize = size - 2 * margin

main :: IO ()
main = getFullArgs >>= putStr . genPostscript . map read
```

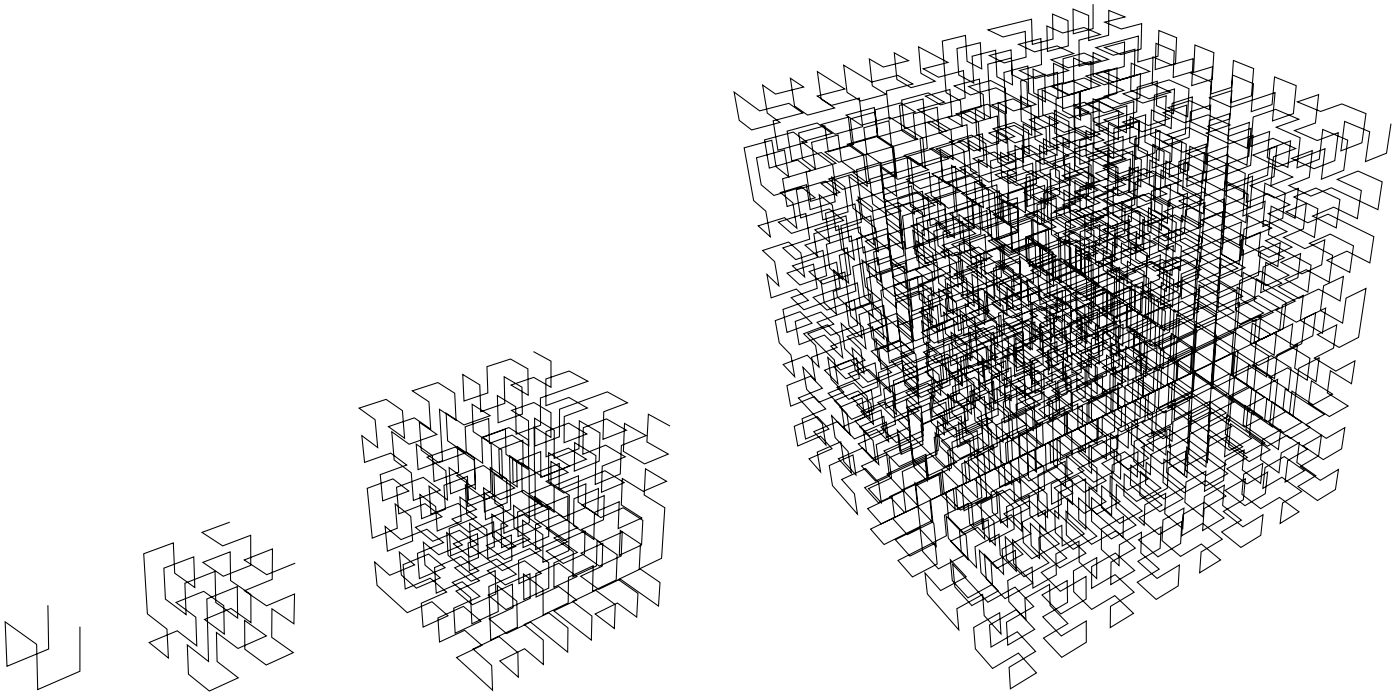
Nie będziemy teraz szczegółowo opisywać tego programu (w końcu zadanie jest nie o tym). Taki program napisany w Pythonie lub C wyglądałby podobnie. Kompletny kod źródłowy jest dostępny w serwisie KNO.

W wielu zastosowaniach Postscript ustąpił miejsca językowi opisu strony PDF. PDF jest językiem dużo niższego poziomu, ręczne edytowanie pliku w PDF-ie jest dosyć żmudne, a wygenerowanie poprawnego pliku PDF wymaga nieco większej pracy, niż w przypadku Postscriptu. Dlatego wybraliśmy Postscript, zwłaszcza że plik postscriptowy można z łatwością zamienić na PDF, np. za pomocą programu Ghostscript:

```
ps2pdf -dEPSCrop obrazek.eps obrazek.pdf
```

Okazuje się, że ideę krzywych wypełniających płaszczyznę można przenieść do wyższych wymiarów. W szczególności możemy zdefiniować krzywe Hilberta w 3D, rysowanie których jest świetnym zadaniem dla studentów drugiego semestru.[4]

Tutaj zależność rekurencyjna jest nieco bardziej skomplikowana: krzywa rzędu $n+1$ składa się z 8 krzywych rzędu n i 7 odcinków:



Komplikuje się też nieco sposób rysowania krzywych. Jeśli nie korzystamy z drukarek 3D, to pozostaje nam jedynie drukować rzuty perspektywiczne obiektów trójwymiarowych na kartkę papieru. Wyznaczenie takiego rzutu jest na szczęście prostym zadaniem geometrycznym.

Załóżmy, że środek układu współrzędnych znajduje się w środku kartki (a nie w lewym dolnym rogu, jak zakładaliśmy do tej pory). Niech oś OZ będzie prostopadła do płaszczyzny kartki. W punkcie o współrzędnych $V = (0, 0, -d)$ umieszczamy

[4] O ile rysowanie dwuwymiarowych krzywych Hilberta jest dobrym zadaniem dla studentów pierwszego semestru, a trójwymiarowych — drugiego, to nie możemy w tej chwili zagwarantować, że w trzecim semestrze będą Państwo rysować krzywe czterowymiarowe.

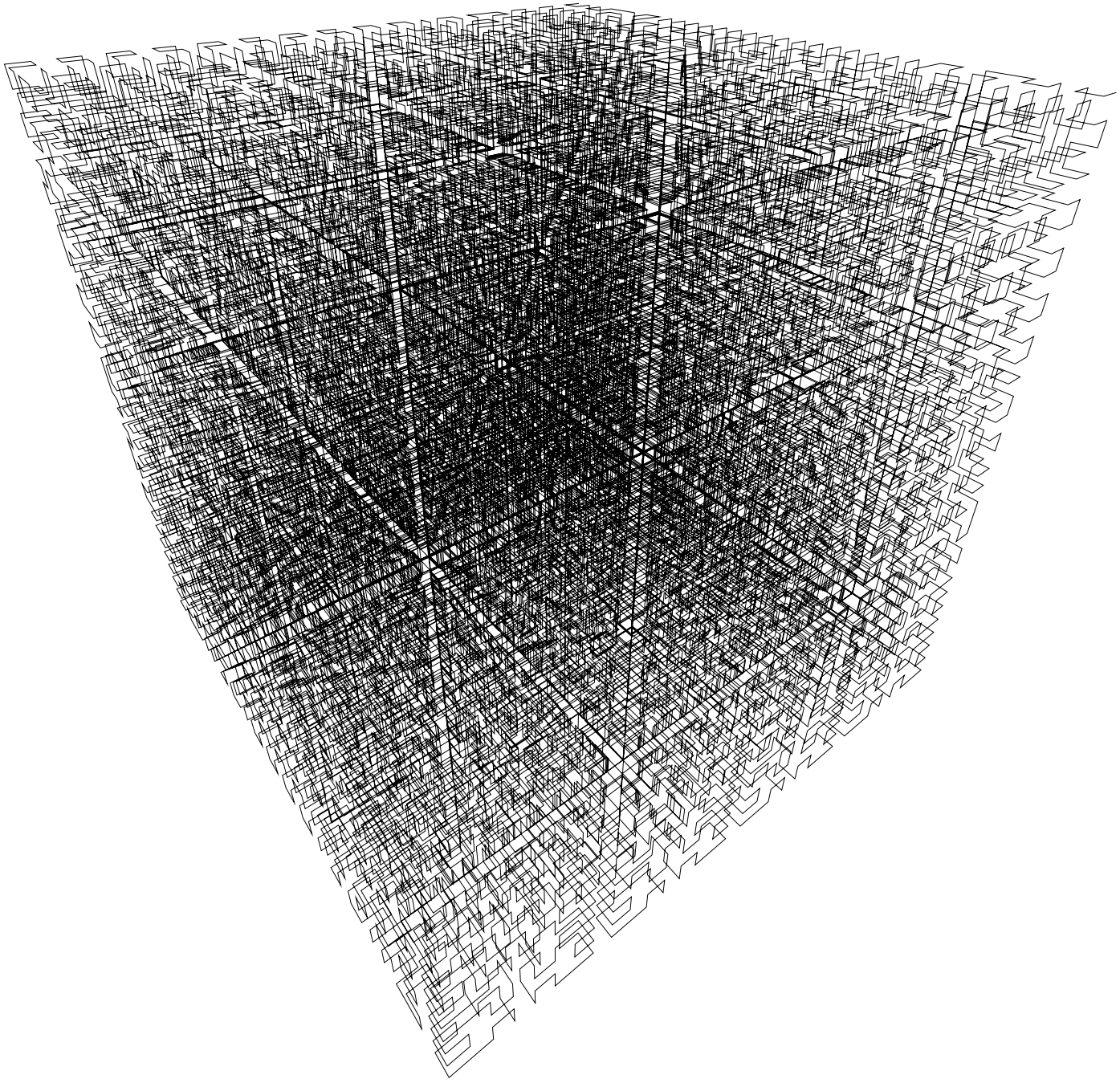
obserwatora. W punkcie $O = (x, y, z)$, gdzie $z > 0$, umieszczamy początek układu współrzędnych rysowanego obiektu i obracamy go o kąty ϕ i ψ względem osi, odpowiednio, OX i OY (zatem np. $\phi = 32.5°$ spowoduje, że będziemy patrzeć na obiekt nieco z góry, zaś $\psi = -38°$ — nieco z lewej strony; takie parametry zostały wybrane w zamieszczonych powyżej obrazkach). Współrzędne punktu $A' = (u', v', w')$ w układzie obiektu można łatwo przeliczyć na współrzędne $A = (u, v, w)$ w układzie kartki. Punkt w którym odcinek AV przebija płaszczyznę kartki jest rzutem perspektywicznym punktu A.

Zadanie. Napisz w języku ANSI C lub Pythonie program, który wypisze do standardowego strumienia wyjściowego plik Postscriptowy zawierający rzut perspektywiczny trójwymiarowej krzywej Hilberta rzędu n, jeśli zostanie uruchomiony z następującymi parametrami:

`hilbert3D` $n\ s\ u\ d\ x\ y\ z\ \phi\ \gamma$

gdzie n to rząd krzywej, s — rozmiar obrazka (w pt), u — długość krawędzi sześcianu w który jest wpisana krzywa (w pt), d — odległość obserwatora od płaszczyzny rzutu (w pt), (x, y, z) — współrzędne początku układu współrzędnych obiektu (w pt), a ϕ i ψ — kąty obrotu (w stopniach) układu współrzędnych obiektu względem obserwatora.

Na koniec przedstawiamy jeszcze trójwymiarową krzywą Hilberta piątego rzędu:



Zwróćmy uwagę na silny skrót perspektywiczny związany z tym, że umieściliśmy obserwatora blisko kartki (d jest małe) i obrazek wygląda tak, jakby był zrobiony obiektywem szerokokątnym.