*student, Wydział Matematyki, Informatyki i Mechaniki, Uniwersytet Warszawski

Prosty sposób na trójwymiarową scenę

*Marian Marek KĘDZIERSKI**

W ciągu kilkunastu ostatnich lat możemy obserwować bardzo szybki rozwój grafiki trójwymiarowej. Staje się ona wszechobecna w filmach, grach, a także w wielu innych dziedzinach.

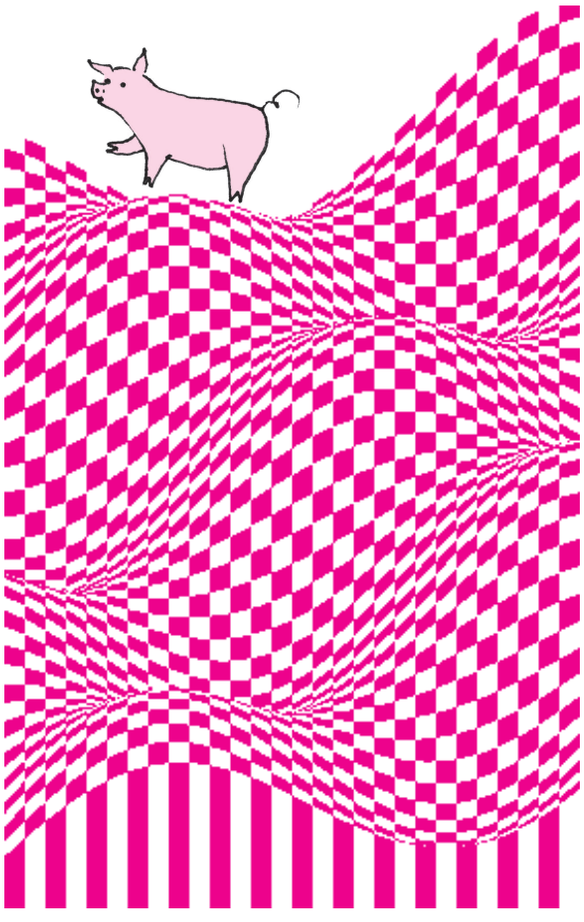
Patrząc na piękne efekty trójwymiarowych scen, chciałoby się umieć samemu stworzyć podobne dzieła. Niestety, wykorzystanie pełni możliwości współczesnych systemów *renderingu* (czyli generowania swoistych „zdjęć" trójwymiarowej sceny zapisanej w pamięci komputera w postaci wektorowej — za pomocą wielościanów, powierzchni, krzywych i tym podobnych obiektów) wymaga dobrej znajomości odpowiednich narzędzi (jak biblioteki OpenGL czy DirectX).

Chcielibyśmy jednak nie tylko umieć przekazać takiej bibliotece nasze żądanie i czekać na gotowy produkt. Znacznie bardziej interesujące jest to, w jaki dokładnie sposób taki rendering się dokonuje. W miarę możliwości chcielibyśmy być w stanie sami go przeprowadzić bez użycia skomplikowanych narzędzi.

Otóż jest łatwy sposób, aby bez większego trudu napisać program generujący trójwymiarową powierzchnię, np. pofałdowany teren gór i dolin. Technika ta nazywa się *wokselowaniem* ...

Podstawowy algorytm wokselowania

Zazwyczaj, kiedy chcemy wyświetlić jakiś obrazek (np. mapę pewnego obszaru albo inny obraz ilustrujący fakturę terenu: trawę, ziemię itp.), rysujemy go na odpowiedniej bitmapie piksel po pikselu. Natomiast idea wokselowania polega na tym, żeby piksele zamienić na wydłużone pionowe woksele, które oprócz koloru obrazka w danym punkcie mają również symbolizować wysokość tego punktu powierzchni.



Rys. 1

Na początku musimy dla każdego piksela obrazka ustalić, na jakiej wysokości $wys(x, y)$ całkowitej, dodatniej ma znajdować się punkt terenu, odpowiadający temu pikselowi. Następnie rysujemy kolejne piksele obrazka. Oto pseudokod prostego wokselowania:

```
for x := 1 to szerokość
for y := 1 to wysokość
  for i := 1 to wys(x,y)
    rysuj(x, y-i, obrazek[x][y])
```

Rysujemy w typowym „ekranowym" układzie stosowanym w grafice komputerowej (punkt $(0, 0)$ w lewym górnym rogu, oś Y skierowana w dół), a nie w standardowej pierwszej ćwiartce układu współrzędnych (punkt $(0, 0)$ w lewym dolnym rogu, oś Y skierowana do góry) .

Zakładamy tutaj, że piksele obrazka są przechowywane w tablicy $obrazek[1..szerokość][1..wysokość]$, zaś funkcja $rysuj(x, y, kolor)$ rysuje na ekranie punkt o współrzędnych (x, y) i kolorze $kolor$.

Dla prostego obrazka przedstawiającego czarno-białą szachownicę o wymiarach 300×450 oraz wysokości danej przez funkcję

$$wys(x, y) = \\ = 80 + 30 \cdot \sin(x - y) + 15 \cdot \sin(x + 2y) + 10x - 5y$$

otrzymujemy wynik przedstawiony na rysunku 1.

Jest to swego rodzaju wykres funkcji wysokości $wys(x, y)$. Wygląda on całkiem ładnie jak na produkt tak krótkiego kodu jak powyżej, nieprawdaż? Pozostaje po nim jednak na samym dnie obrazka swoista „smuga” — ślad po dolnych wokselach. Jest jeszcze inny, nieco poważniejszy problem: mianowicie, rysowanie długich wokseli zamiast pojedynczych pikseli bardzo spowalnia pracę algorytmu. Jak się okazuje, możemy upiec obie te pieczenie na jednym ogniu...

Modyfikacje

Prosta przeróbka podstawowego algorytmu rozwiązuje oba wyżej wspomniane problemy. Wystarczy bowiem, jeżeli będziemy każdy woksel rysować od góry w dół tylko do momentu, aż natrafimy na szczyt woksela będącego jego dolnym sąsiadem:

```
for x := 1 to szerokość
for y := 1 to wysokość
  for i := wys(x, y + 1) to wys(x, y)
    rysuj(x, y - i, obrazek[x][y]);
```

Jeżeli chcemy dodatkowo uatrakcyjnić wygląd rysowanej powierzchni, możemy ją wyświetlać tak, jakby była oglądana z boku, „pod kątem”. Aby to uzyskać, wystarczy przesuwać nieznacznie każdą kolejną poziomą warstwę wokseli.

Przyda się nam do tego pomocnicza funkcja $przesunięcie(y)$ zwracająca liczbę pikseli, o jaką należy przesunąć rząd na wysokości y. Możemy, na przykład, przyjąć $przesunięcie(y) = y \cdot 0{,}4$. Wtedy nasz program będzie miał następującą postać:

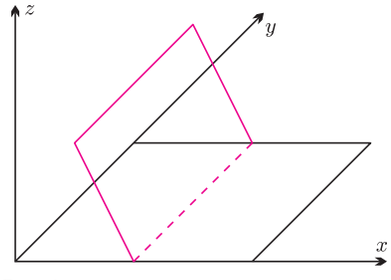
```
for x := 1 to szerokość
for y := 1 to wysokość
  x_sąsiada := x + przesunięcie(y + 1) - przesunięcie(y);
  y_sąsiada := y + 1;
  for i := wys(x_sąsiada, y_sąsiada) to wys(x, y)
    rysuj(x + przesunięcie(y), y - i, obrazek[x][y]);
```

Efekt jego wykonania przedstawia obrazek na okładce.

Ciekawe dodatki

Możemy wzbogacić nasz algorytm wokselowania wieloma ciekawymi efektami.

Jednym z nich może być dodanie cieniowania. Możemy, na przykład, przyjąć, że źródło światła (Słońce) znajduje się po lewej stronie sceny, przy czym jest na tyle daleko, że padające z niego promienie są równoległe.



Rys. 2

Przyjmijmy, że promienie te padają wzdłuż płaszczyzn o równaniach

$$a \cdot x + z = \text{const},$$

gdzie a jest pewnym współczynnikiem zależnym od kąta padania promieni (rys. 2).

Wartość $ax + wys(x, y)$ można utożsamiać z wysokością płaszczyzny, która przechodzi przez punkt naszej powierzchni leżący nad punktem (x, y). Oznaczmy tę wysokość $wys\_sty(x, y)$.

Żeby uzyskać efekt cieniowania, potrzebujemy dla każdego punktu (x, y) naszej powierzchni znać maksymalną spośród wartości $wys\_sty(x', y)$ dla wszystkich punktów (x', y) na lewo od punktu (x, y) (tzn. takich, że $x' < x$). Różnica tej maksymalnej wartości i $wys\_sty(x, y)$ będzie stopniem ocieniowania punktu (x, y). Wystarczy ją jeszcze przeskalować, np. przy użyciu funkcji *arcus tangens*:

$$cień(różnica\_wys\_sty) = 1 + \arctan(różnica\_wys\_sty) \cdot k,$$

gdzie k jest pewnym współczynnikiem określającym ostrość cienia i oświetlenia. Dobrze jest przyjąć k z zakresu $[-\frac{2}{\pi}; \frac{2}{\pi}]$, ponieważ sama funkcja *cień* powinna mieć wartości nieujemne.

Tak otrzymaną wartość cienia możemy pomnożyć przez wszystkie składowe koloru w danym punkcie, aby otrzymać kolor, który ma zostać wyświetlony. Potrzebujemy jeszcze tylko znaleźć metodę wyznaczania maksymalnej wartości $wys\_sty(x', y)$ dla $x' < x$. Jeżeli będziemy rysowali powierzchnię rząd po rzędzie od góry do dołu, każdy zaś rząd od lewej do prawej, to wartości te możemy zliczać dynamicznie, nie pogarszając złożoności algorytmu wyświetlającego, co Czytelnik bez problemu zaprogramuje. Wewnętrzna pętla rysująca woksel będzie teraz mieć postać:

```
wsp_cienia := cień (różnica_wys_sty)
for i := wys(x_sąsiada, y_sąsiada) to wys(x, y)
  kolor := skaluj(obrazek[x][y], wsp_cienia);
  rysuj(x + przesunięcie(y), y - i, kolor);
```

Efekty takiego cieniowania dla stałej $k = 0{,}6$ oraz różnych stałych a przedstawiają obrazki na okładce. Można tam także zobaczyć takie same sceny, ale już z nałożonymi konkretnymi teksturami bitmapowymi zamiast sztucznego obrazka w szachownicę, jednak tym razem dla stałej $k = 0{,}5$.

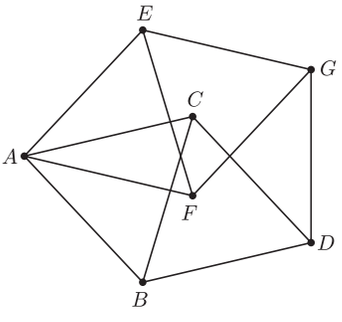
Technika wokselowania daje ciekawe efekty przy bardzo niewielkim nakładzie pracy. Niestety, ogranicza się ona do pofałdowanych powierzchni, a już zupełnie nie nadaje się do renderowania wielościanów, które są podstawą większości systemów grafiki trójwymiarowej. Jeżeli więc nie chcemy ograniczać się do efektów podobnych do przedstawionych w tym artykule, to potrzebujemy wniknąć znacznie głębiej w dziedzinę informatyki, jaką jest grafika trójwymiarowa.

Ale to już zupełnie inna opowieść...

Zestaw programów, którymi wygenerowane zostały obrazki z okładki, można pobrać ze strony WWW *Delty*.

Liczba chromatyczna płaszczyzny

Jeżeli każdy punkt płaszczyzny pomalujemy na jeden z trzech kolorów, to znajdą się dwa punkty tego samego koloru w odległości 1. Łatwo to wykazać. Rozważmy w tym celu konfigurację 7 punktów położonych tak, jak na rysunku, przy czym każdy z zaznaczonych odcinków ma długość 1. Łatwo stwierdzamy, że nie da się ich pokolorować trzema kolorami bez otrzymania monochromatycznej pary w odległości 1 (spróbujmy – malujemy A; wtedy B i C muszą otrzymać pozostałe dwa kolory; D musi więc mieć ten sam kolor co A; tak samo dla G; okazuje się więc, że D i G mają ten sam kolor).



Z kolei można tak pokolorować płaszczyznę 7 kolorami, żeby monochromatycznej pary w odległości 1 nie było. Zaczynamy od wyparkietowania płaszczyzny siatką sześciokątów foremnych, jak na planszy do gry Hex. Czytelnik bez problemu odpowiednio je pokoloruje, unikając jednokolorowej pary punktów odległych o 1 (każdy sześciokąt i jego sześciu sąsiadów powinno mieć w sumie 7 różnych kolorów; ostrożnie na krawędziach; jaką wziąć średnicę sześciokątów?).

Liczba chromatyczna płaszczyzny $\chi(\mathbb{R}^2)$ to najmniejsza liczba kolorów, potrzebna do pokolorowania płaszczyzny w taki sposób, aby uniknąć dwóch punktów odległych o 1 w tym samym kolorze. Wiemy już zatem, że:

$$4 \leqslant \chi(\mathbb{R}^2) \leqslant 7.$$

Co jeszcze wiadomo? W ogólnym przypadku – nic. Nie umiemy posunąć się ani o krok względem tych bardzo prostych oszacowań. Nierozwiązany dotąd problem znalezienia liczby chromatycznej płaszczyzny (albo chociaż zmniejszenia zakresu widełek ją ograniczających) nosi też nazwę problemu Hadwigera–Nelsona.

Liczba chromatyczna płaszczyzny jest szczególnym przypadkiem ogólniejszego pojęcia *liczby chromatycznej grafu*. Jest to najmniejsza liczba kolorów, potrzebna do pokolorowania wierzchołków danego grafu tak, aby żadna krawędź nie łączyła wierzchołków o tym samym kolorze. Dla płaszczyzny (i każdej innej przestrzeni, w której można mierzyć odległość) możemy skonstruować tzw. graf odległości jednostkowych, którego wierzchołkami są punkty płaszczyzny, a krawędzie łączą punkty odległe o 1. Powiedzielibyśmy więc, że ten graf jest 7-kolorowalny, nie jest 3-kolorowalny i nie wiemy, czy jest 4-, 5- lub 6-kolorowalny.

A zatem – kredki w dłonie i do pracy! Ale uwaga – znane są różne ograniczenia. Wiadomo na przykład, że jeśli chcielibyśmy użyć tylko 4 kolorów, to którymś z nich musielibyśmy namalować zbiór niemierzalny, czyli wyjątkowo paskudny (nie do namalowania kredką).

M.A.