Grafika

METAPOST i wzorki

Piotr Bolek

Streszczenie

W artykule są opisane możliwości definiowania wypełnień wzorkowych w programie METAPOST. Opisany jest opracowany przez autora zestaw makr służących do wygodnego definiowania i wykorzystywania takich wypełnień. W makrach tych jest wykorzystywana Wzorkowa Przestrzeń Kolorów (ang. *Pattern Color Space*) dostępna w języku PostScript Level 2. Na kilku przykładach są pokazane problemy, które można napotkać przy definiowaniu wzorków oraz propozycje ich rozwiązywania.

Abstract

In this paper the possibilities of defining and using patterns in METAPOST program are presented. The idea of macro package allowing comfortable way of defining and using pattern fills is outlined. These macros enable user to access indirectly, but effectively Pattern Color Space of PostScript Level 2. The examples of patterns and solving problems which can arise during their definition are also given.

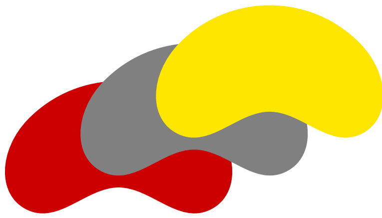
METAPOST i wypełnianie obszarów

METAPOST jest opartym na programie METAFONT programem graficznym. W przeciwieństwie do swojego przodka (matki – METAFONT jest jak wiemy płci pięknej) generuje on pliki wynikowe w PostScript, które mogą być w standardowy sposób umieszczane w dokumencie TeX-owym. Oprócz wszystkich najważniejszych cech METAFONT-a, METAPOST ma kilka rozszerzeń związanych bezpośrednio z PostScript-em, np. możliwość używania kolorów, definiowanie kształtów zakończeń odcinków i rogów w liniach łamanych.

METAPOST doskonale współpracuje z TeX-em. Przy tworzeniu napisów korzysta z plików TFM, może także wstawiać teksty przetworzone przez TeX-a. Daje to możliwość składania np. opisów na wykresach dokładnie tymi samymi czcionkami co w tekście.

Jak na porządny program graficzny przystało METAPOST pozwala wypełniać zamknięte obszary kolorami – mogą to być odcienie szarości, albo „prawdziwe” kolory. METAPOST używa modelu kolorów RGB – kolor jest podawany jako trójwymiarowy wektor liczb (typ `color`) z zakresu od 0 do 1. Kolor biały to `(1, 1, 1)`, czarny – `(0, 0, 0)`, czerwony – `(1, 0, 0)`, żółty – `(1, 1, 0)`, itd.

Możemy więc rysować dowolne wypełnione jednolitym kolorem figury (rys. 1).



Rys. 1.

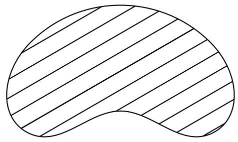
W programie METAPOST brakuje jednak możliwości łatwego definiowania wzorków do wypełniania obszarów. Nie ma mechanizmu analogicznego do mechanizmu wyboru grubości i kształtu pióra czy koloru wypełniania.

Można oczywiście spróbować zdefiniować sobie na przykład procedurę do kreskowania obszarów korzystającą ze standardowych możliwości jakie daje METAPOST. Prosta procedura rysowania figur zakreskowanych może wyglądać np. tak:

```
vardef kreskuj(expr p,n)=
   save x,y;
   z1=llcorner p; z2=lrcorner p;
   z3=urcorner p; z4=ulcorner p;
   for i=0 upto n/2:
      draw (i/n)[z1,z2]--(i/n)[z3,z2];
      draw (i/n)[z1,z4]--(i/n)[z3,z4];
   endfor;
   clip currentpicture to p;
enddef;
```

Procedura ta jest raczej nieskomplikowana. Jej argumentami są: ścieżka (oczywiście zamknięta) i liczba kresek, które mają się zmieścić w prostokącie ograniczającym figury. Korzystamy tutaj także z możliwości kadrowania rysunku do danej ścieżki. Zakładając, że w zmiennej `bean` zapamiętaliśmy naszą ścieżkę możemy ją zakreskować następująco:

```
   kreskuj(bean, 14);
   draw bean;
```

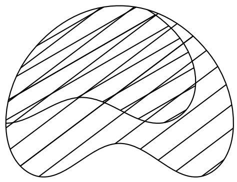


Rys. 2.

Wynik widać na rys. 2.

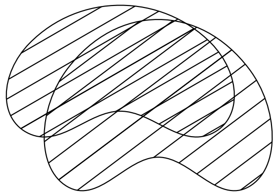
Na początek nieźle, ale rozwiązanie to ma niestety poważne wady:

- Nie można podać kąta kreskowania. Jest on taki jak kąt przekątnej prostokąta ograniczającego.
- Nie można też bezpośrednio podać odstępów między kreskami. Wynikają one z rozmiarów prostokąta ograniczającego ścieżkę i liczby kresek, które mają być narysowane.
- I wreszcie najgorsza wada: nie da się narysować na jednym rysunku dwóch zakreskowanych figur. Użycie procedury kreskowania dla kolejnej figury wykadruje nam poprzednio narysowane elementy (rys. 3).



Rys. 3.

Wadę ostatnią można usunąć zapamiętując przed każdym kreskowaniem aktualną zawartość rysunku w zmiennej i odtwarzając ją po zakreskowaniu i wykadrowaniu nowej ścieżki. Efekt widać na rys. 4.



Rys. 4.

Definiowanie wypełnień wzorkowych (nawet tak prostych jak kreskowanie, a gdzie np. kolorowe gwiazdki albo fale?) jest więc jak widać nienaturalne i dosyć kłopotliwe. Ja w każdym razie nie podjąłbym się uniwersalnego rozwiązania problemu wypełnień wzorkowych korzystając tylko z możliwości jakie daje sam METAPOST nawet dla przypadku uproszczonego, kiedy zadowolilibyśmy się tylko kreskowaniem, ale za to pod dowolnie podanym kątem. Wydaje się więc, że sytuacja jest beznadziejna. Na szczęście nie do końca.

Koligacje rodzinne i co z nich wynika

Jak już wielokrotnie w gustownym gronie mówiono – TEX, METAFONT i cała rodzinka w tym także METAPOST to programy otwarte, które można łatwo zachęcić do współpracy z innymi narzędziami i między sobą. Sam METAPOST jest przecież dzieckiem udanych rodziców: METAFONT (to mama) i POSTSCRIPT (tata). Znakomicie także korzysta z pomocy dziadka TEX-a przy tworzeniu napisów na rysunkach. W czasie dojrzewania taty-POSTSCRIPT-u ujawniła się u niego cecha bardzo dla nas ciekawa: jest to dodana w POSTSCRIPT Level 2 Wzorkowa Przestrzeń Kolorów (ang. *Pattern Color Space*). Dlaczego nie miałby skorzystać z tego i syn?

Wzorkowa Przestrzeń Kolorów a METAPOST

Wykorzystując wzorkową przestrzeń kolorów można łatwo definiować wzory wypełnień. Wszystko co trzeba zrobić sprowadza się w zasadzie do zdefiniowania procedury rysującej podstawową komórkę wzorku. Komórka ta będzie później automatycznie powielana przez interpreter POSTSCRIPT-u. Interpreter zajmie się także „obcięciem” wzorku na granicy wypełnianego obszaru.

Problem tylko w tym, jak zmusić METAPOST-a do skorzystania z odziedziczonych po rodzicu możliwości. Pomysł jest następujący: wzorek a właściwie jego procedura rysująca to przecież zwyczajna procedura w POSTSCRIPT, a METAPOST służy właśnie do tworzenia rysunków czyli procedur w POSTSCRIPT. Co więcej te rysunki są typu „encapsulated” czyli ograniczone i granice tych rysunków są znane METAPOST-owi. Wystarczy więc stworzyć za pomocą METAPOST-a rysunek i wykorzystać wygenerowany dla niego kod POSTSCRIPT-y jako procedurę rysującą wzorku. Definicję takiego wzorku umieścimy potem na początku pliku wynikowego zawierającego kod POSTSCRIPT-y dla rysunku, w którym użyjemy wzorku jako wypełnienia. Do rozwiązania pozostają już tylko drobne kwestie techniczne i interfejs użytkownika.

Implementacja

Wzorki są w POSTSCRIPT realizowane jako szczególnego rodzaju kolory i tak jak one służą do wypełniania obszarów. U nas na poziomie METAPOST-a wzorki także będą podawane jako kolory, ale takie, które nie mają raczej szans być podane przez użytkownika – to kolory o wartościach od `epsilon*white` do `n*epsilon*white` gdzie n jest liczbą wzorków zdefiniowanych w danym pliku źródłowym (oraz w plikach włączanych do niego poleceniem `input`). Na wydruku kolory te byłyby zupełnie czarne. Zastąpienie „wywołania" takiego koloru przez „wywołanie" odpowiedniego wzorku jest wykonywane przez skrypt w perlu po wygenerowaniu przez METAPOST plików POSTSCRIPT-owych.

Jak już wspomniałem każdy wzorek jest tworzony tak jak rysunek i w rzeczywistości jest rysunkiem. Aby uniknąć konfliktu z numerami rysunków wykorzystywanymi przez użytkownika, do definicji wzorku wybierany jest duży wolny numer (zwykle 999). POSTSCRIPT-owy kod procedury rysującej jest wczytywany z pliku wynikowego dla tego rysunku. Po usunięciu niepotrzebnych nam komentarzy i dodaniu elementów wymaganych przez składnię definicji wzorku, definicja ta jest umieszczana poleceniem `special` na początku każdego rysunku, w którym dany wzorek został użyty do wypełnienia jakiegoś obszaru.

Interfejs użytkownika. Definiowanie wzorków zrealizujemy w sposób możliwie najbardziej naturalny dostosowując się do konwencji programów METAFONT i METAPOST. Aby zdefiniować wzorek należy napisać:

```
beginpattern(⟨NazwaWzorku⟩);
...
endpattern;
```

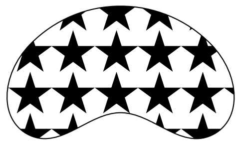
⟨*NazwaWzorku*⟩ jest nazwą, jakiej będziemy używać do identyfikacji wzorku. Jest to jednocześnie zmienna używana do przechowywania definicji wzorku[1]. Między poleceniami `beginpattern` i `endpattern` umieszczamy kod rysujący wszystkie elementy podstawowej komórki wzorku – przy czym zwykle trzeba zadbać o to aby sąsiednie komórki wzorku dobrze się „sklejały" (wkrótce zobaczymy o co chodzi). Dodatkowo w dowolnym miejscu między tymi poleceniami można użyć wywołania `patternbbox` dla podania prostokąta ograniczającego (a raczej obcinającego) naszą podstawową komórkę wzorku. Argumentami tego polecenia mogą być cztery liczby, albo dwie pary określające lewy-dolny i prawy-górny róg prostokąta. Podanie tego polecenia nie jest obowiązkowe (chociaż zwykle konieczne) – jeśli go nie będzie, to użyty zostanie prostokąt ograniczający całego rysunku wykorzystanego jako definicja wzorku.

Używanie zdefiniowanego wcześniej wzorku jest jeszcze prostsze. Aby wypełnić naszą ścieżkę zdefiniowanym wcześniej wzorkiem piszemy:

```
fill ⟨p⟩ withpattern ⟨NazwaWzorku⟩;
```

gdzie ⟨*p*⟩ jest dowolnym wyrażeniem określającym zamkniętą ścieżkę, a ⟨*NazwaWzorku*⟩ jest nazwą podaną w definicji wzorku.

Teraz możemy już wypełniać obszary dowolnie definiowanymi wzorami np. gwiazdkami jak na rys. 5. Definiowanie wzorów wypełnień nie jest jednak takie całkiem proste. Na razie rozwiązaliśmy ograniczenia techniczne. Możemy więc teraz trochę poeksperymentować.

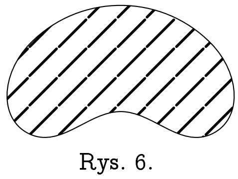


Rys. 5.

Definiowanie wzorów wypełnień

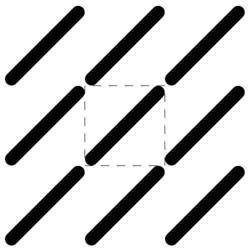
Kreskowanie. Mając już wygodne narzędzie do tworzenia i używania wzorków możemy zrobić drugie podejście do problemu kreskowania. Kreskowanie pod kątem 45° można zrealizować teraz bardzo łatwo:

```
beginpattern(xlv);
   u:=10;
   drawoptions(withpen
      pencircle scaled 1);
   draw (0,0)--(u,u)
endpattern;
beginfig(6);
   fill bean withpattern xlv;
   draw bean;
endfig;
```



Rys. 6.

1: W definicji `beginpattern` ⟨*NazwaWzorku*⟩ jest parametrem typu `suffix` dlatego możemy ją podać bez cudzysłowów.

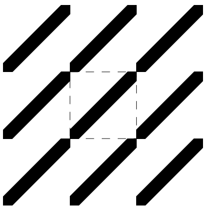


Rys. 7.

Uzyskany efekt nie jest jednak idealny (rys. 6). Kreski mają drobne nieciągłości. Popatrzmy na nasz wzorek w powiększeniu (rys. 7). Definiując go zostawiliśmy wyznaczenie prostokąta ograniczającego podstawowej komórki METAPOST-owi. Jest on bardzo dokładny w wyznaczaniu tego prostokąta i uwzględnia w nim także grubość kreski. Dlatego prostokątem ograniczającym nie jest, jak moglibyśmy się spodziewać [0 0 10 10], ale [−0.5 −0.5 10.5 10.5] (grubość pióra wynosiła 1bp). Podanie prostokąta ograniczającego jawnie niestety nie wystarczy. Jeśli dodamy do definicji wzorku wiersz:

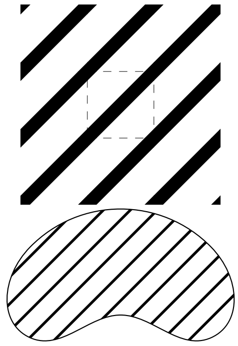
```
patternbbox(0,0,u,u);
```

to uzyskamy efekt taki jak na rys. 8. Prostokąt ograniczający jest teraz właściwy, ale ponieważ wzorek jest poza nim obcinany, to w miejscu stykania się rogów komórek kreski są „wygryzione”. Definiując wzorek trzeba więc narysować także kreski w lewym-górnym i prawym-dolnym rogu komórki.



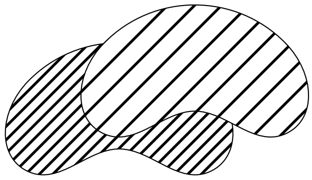
Rys. 8.

```
beginpattern(xlv_OK);
   u:=10;
   drawoptions(withpen
      pencircle scaled 1);
   draw (0,0)--(u,u);
   draw ((0,0)--(u,u))
      shifted (-u/2,u/2);
   draw ((0,0)--(u,u))
      shifted (u/2,-u/2);
   patternbbox(0,0,u,u);
endpattern;
```



Rys. 9.

Wreszcie udało się nam zdefiniować wzorek, dzięki któremu kreskowanie zrobione jest tak jak trzeba (rys. 9). Długość kreski w definicji wzorku określa gęstość kreskowania. Jeśli będziemy definiować wzorek rysując krótszą kreskę, kreskowanie będzie gęstsze, jeśli dłuższą – rzadsze (rys. 10).



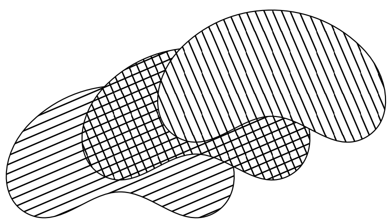
Rys. 10.

Umiemy już zdefiniować kreskowanie pod kątem 45°. A co z innymi kątami? Możemy spróbować uogólnić sposób tworzenia wzorów kreskowania dla dowolnego kąta. Kreska rysowana w komórce wzorku musi być przekątną prostokąta ograniczającego. Nie możemy też zapomnieć o lewym górnym i prawym dolnym rogu komórki. Uogólniona definicja wzorku może wyglądać tak:

```
def kreski(suffix name)(expr alpha, d)=
beginpattern(name);
   z1=(0,0);
   z2=(d,0) rotated alpha;
   draw z1--z2;
   draw (z1--z2) shifted
      (sind(alpha)*x2*dir(alpha+90));
   draw (z1--z2) shifted
      (sind(alpha)*x2*dir(alpha-90));
   patternbbox(z1,z2);
endpattern;
enddef;
```

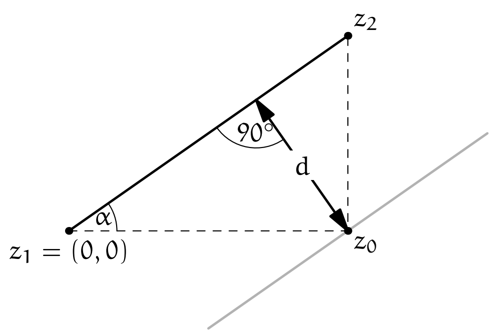
Parametr `name` jest nazwą definiowanego wzorku, a parametry `d` i `alpha` decydują o kącie i gęstości kreskowania. Gęstość kreskowania nie jest tutaj podana wprost jako odległość między kreskami ale za pośrednictwem długości kreski używanej przy rysowaniu podstawowej komórki kreskowania. Zmieniając parametry d i alpha możemy uzyskiwać kreskowanie pod różnymi kątami i z różną gęstością. Kreskując ten sam obszar dwa razy z taką samą gęstością i pod kątami różniącymi się od siebie o 90° otrzymamy obszar zakratkowany (rys. 11).

```
kreski(k_a, 23, 3mm);
kreski(k_b, 113, 3mm);
beginfig(11);
   ...
   unfill bean;
   fill bean withpattern k_a;
   fill bean withpattern k_b;
   draw bean;
   ...
endfig;
```



Rys. 11.

Mamy już niezłe narzędzie do kreskowania obszarów ale chcielibyśmy jeszcze mieć możliwość definiowania kreskowania przez podanie kąta i odstępu między kreskami. Musimy zdefiniować więc makro, które dla znanej odległości między kreskami oraz kąta nachylenia kresek wyznaczy przekątną komórki wzorku. Zadanie, które trzeba rozwiązać warto zilustrować rysunkiem, co znakomicie ułatwi jego rozwiązanie (rys. 12).



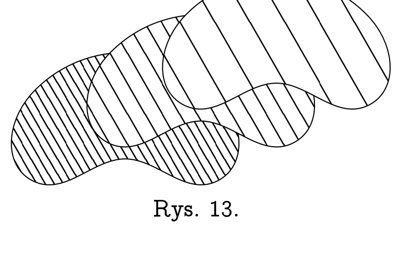
Rys. 12.

Punkt z_1 jest znany – to lewy-dolny róg komórki, czyli początek układu współrzędnych. Nieznane są współrzędne punktu z_2. Znamy za to odległość punktu z_0 od odcinka $\overline{z_1z_2}$ oraz kąt α między odcinkami $\overline{z_1z_2}$ i $\overline{z_1z_0}$. Definicja makra tworzącego komórkę naszego wzorku może być następująca:

```
1. def kreskowanie(suffix nazwa)(expr
2.    alpha, d)=
3. beginpattern(nazwa);
4.    z1=origin; z0=(x2,y1);
5.    z0+dir(90+alpha)*d
6.       =z1+whatever*dir(alpha);
7.    z2=whatever[z1,z1+dir(alpha)]
8.       =whatever[z0,z0+up];
9.    draw z1--z2;
10.   draw (z1--z2) shifted
11.      (d*dir(angle(z2-z1)+90));
12.   draw (z1--z2) shifted
13.      (d*dir(angle(z2-z1)-90));
14.   patternbbox(z1,z2);
15. endpattern;
16. enddef;
```

Równanie w wierszach 5 i 6 mówi, że odległość punktu z_0 od odcinka $\overline{z_1z_2}$ wynosi d. Dzięki temu równaniu wyznaczona zostaje współrzędna x_0, która jest równa x_2. Drugie równanie (wiersze 7 i 8) mówi, że punkt z_2 leży na przecięciu pionowej prostej przechodzącej przez punkt z_0 i prostej o nachyleniu α przechodzącej przez punkt z_1. Ponieważ współrzędne punktu z_0 są już znane, to wystarczy to do wyznaczenia nieznanej współrzędnej y_2. Tak jak poprzednio pamiętamy o rogach komórki (wiersze 10–13).

Możemy teraz definiować kreskowanie podając jego gęstość wprost, a nie tak jak poprzednio tylko pośrednio. Na rys. 13 widać trzy figury zakreskowane kreskami z odstępami 1mm, 2mm i 4mm.

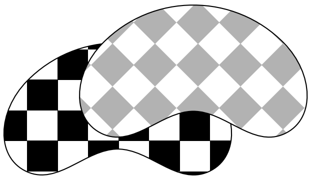


Rys. 13.

Przy definiowaniu kreskowania pod kątami bliskimi zeru albo wielokrotnościom 90° trzeba uważać, ponieważ jeśli odległości między kreskami będą duże, to prostokąt, który trzeba skonstruować i jego przekątna mogą być tak duże, że spowodują przekroczenie ograniczeń albo samego METAPOST-a albo interpretera POSTSCRIPT-u, który będzie użyty do wydrukowania bądź wyświetlenia rysunku. Nie można także podać kątów równych wielokrotności 90° ponieważ rozwiązania równań użytych do wyznaczenia współrzędnych punktu z_2 są wtedy niejednoznaczne. Ale to nie jest wielki problem – zdefiniowanie wzorku do kreskowania poziomego albo pionowego jest przecież trywialne.

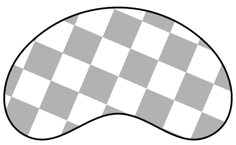
Inne wzorki. Mechanizm definiowania wzorków jest oczywiście uniwersalny i nie musimy ograniczać się tylko do banalnych (chociaż pożytecznych) wzorków kreskowych. Ciekawszym i dosyć prostym wzorem są szachownice. Proste szachownice możemy zdefiniować z łatwością:

```
beginpattern(szachy_i);
   fill unitsquare scaled 4mm;
   fill unitsquare scaled 4mm
      shifted (4mm,4mm);
endpattern;
beginpattern(szachy_ii);
   fill unitsquare scaled 4mm
      rotated 45
      withcolor .7white;
endpattern;
```



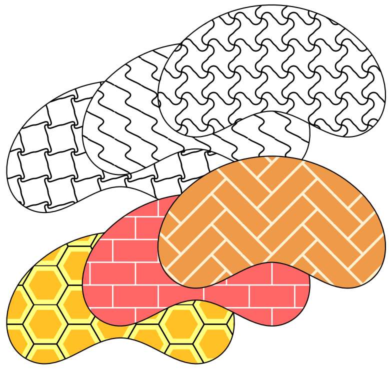
Rys. 14.

Wypełnienie obszaru szachownicą pod dowolnym kątem nie jest już takie proste... ale możliwe (rys. 15). Nie potrafię w każdym razie podać ogólnej procedury na definiowanie szachownicy pod dowolnym kątem tak jak dla kreskowania – chociaż potrafię narysować obszar wypełniony szachownicą pod dowolnym kątem! Czy ktoś zgadnie jaką sztuczkę zastosowałem dla uzyskania rys. 15? Dla ułatwienia powiem, że obszar na tym rysunku został wypełniony wzorkiem szachy_ii zdefiniowanym w ostatnim przykładzie.

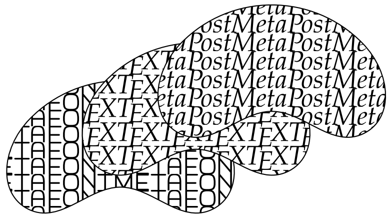


Rys. 15.

Oczywiście wzorki nie muszą być czarno-białe – definiując je możemy używać dowolnych kolorów.



Rys. 16.



Rys. 17.

Kilka przykładów wzorków kolorowych (w druku niestety nie całkiem, ale w wersji elektronicznej, np. w PDF-ie tak) widać na rys. 16.

We wzorkach możemy używać nie tylko elementów graficznych można także tworzyć wzorki zawierające napisy. Trzy wzorki widoczne na rys. 17 zostały zdefiniowane w następujący sposób:

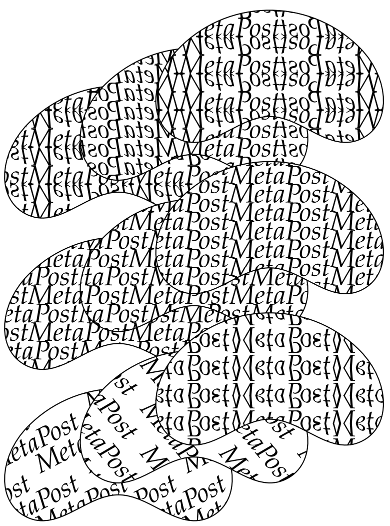
```
verbatimtex \font\q=qplri etex;
beginpattern(simpletext_i);
   label(btex \MF etex, origin);
endpattern;
beginpattern(simpletext_ii);
   label(btex \q \TeX etex, origin);
endpattern;
beginpattern(simpletext_iii);
   label(btex \q MetaPost etex, origin);
endpattern;
```

Możliwe są też bardziej wymyślne wzorki wykorzystujące tekst dla uzyskania ciekawych efektów graficznych. Na rys. 18 wzorki zawierają tekst przesuwany, obracany i „odbijany w lustrze”.

Ostatnie zastosowania wzorków są już mało poważne i ich znaczenie jest w zasadzie jedynie estetyczne. Na zakończenie zobaczymy do czego naprawdę mogą nam się przydać wzorki.

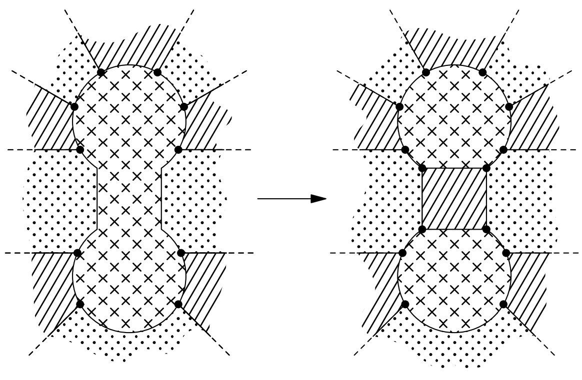
Zakończenie

Moje zainteresowanie wzorkami w programie METAPOST nie jest przypadkowe. W książce o kombina-



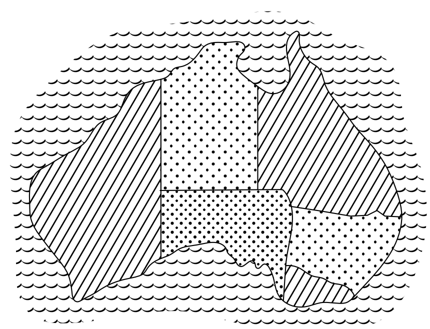
Rys. 18.

toryce, do której niedawno robiłem rysunki był rozdział o kolorowaniu map. Oczywiście przykładowe mapy, które były „kolorowane” w książce nie były wcale kolorowe. Zamiast kolorów używane były właśnie różnego rodzaju wzorki (rys. 19 i 20), takie jak kreskowanie, kropkowanie, krzyżyki, itp. Możliwość wypełniania obszarów wzorkami była więc czymś czego mi bardzo brakowało. Początkowo próbowałem poradzić sobie używając mechanizmów dostępnych w samym METAPOST – wypełniając obszary i kadrując je zamkniętymi ścieżkami (coś podobnego jak przykład na początku artykułu). Metoda ta okazała się dosyć kłopotliwa, a w niektórych przypadkach zupełnie zawodziła.



Rys. 19.

Z drugiej strony wiedziałem, że definiowanie wzorków w PostScript jest proste. Pierwszym pomysłem było definiowanie procedur wzorkowych bezpośrednio w tym języku, włączanie definicji wzorków jawnymi wywołaniami poleceń `special` i zamiana ustalonych kolorów na wzorki. Nie było to jednak ani wygodne, ani eleganckie. Co prawda



Rys. 20.

prosty skrypt w perlu i Makefile upraszczał sprawę, ale po dodaniu nowego wzorku trzeba było wybrać kolejny kolor, który będzie później zastąpiony tym wzorkiem. W pliku źródłowym nie było oczywiste, które kolory zostaną zastąpione wzorkami, a które pozostaną „prawdziwymi” kolorami.

Rozwiązanie ostateczne jest dużo bardziej eleganckie i dużo wygodniejsze dla użytkownika. Jak widzieliśmy na przykładzie kreskowania definiowanie wzorków, których komórki będą dobrze do siebie pasowały nie jest takie całkiem trywialne. Zaimplementowany mechanizm definiowania wzorków uwalnia nas jednak od technicznych problemów związanych z „układaniem” i „obcinaniem” wzorku a pozwala skupić się na samej definicji wzorku, którą realizujemy w sposób dla użytkowników programu METAPOST (albo METAFONT) naturalny.

Literatura

[1] Adobe Systems Inc., *PostScript Language Reference Manual*, Second Edition, Addison-Wesley, 1990.

[2] John D. Hobby, *A User's Manual for* METAPOST, AT&T Bell Laboratories, 1995.

[3] Donald E. Knuth, *The METAFONTbook*, Addison-Wesley, 1986.

◇ Piotr Bolek
P.Bolek@ia.pw.edu.pl