Michał Radziszewski*

Teksturowanie proceduralne za pomocą języka funkcyjnego

1. Wprowadzenie

Teksturowanie jest techniką wykorzystywaną przy renderingu, stosowaną w celu poprawy jakości obrazów bez zwiększania złożoności reprezentacji geometrycznej obiektów. Przykładowe zastosowanie tekstury na rysunku 1 pokazuje, jak dużo we współczesnej grafice komputerowej modelowane jest za pomocą tekstur.

Rys. 1. Po lewej stronie: postać wykorzystuje wyłącznie model geometryczny. Po prawej stronie: model geometryczny z nałożoną teksturą

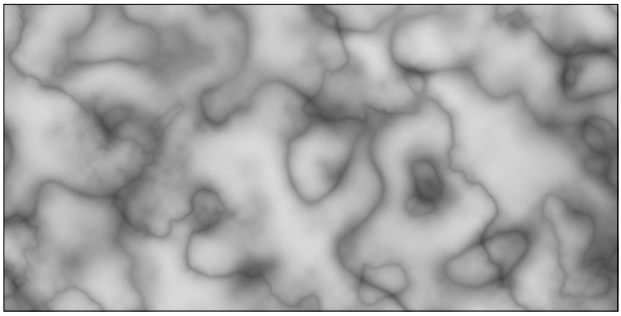
1.1. Teksturowanie zwykłe i proceduralne

Teksturowanie zwykłe (przedstawione na rys. 1) polega na nakładaniu dwuwymiarowych map na trójwymiarowe modele. Taka technika, pomimo że pozwala na niemalże dowolne opisanie wyglądu trójwymiarowych obiektów, ma kilka wad. Po pierwsze, mapy te muszą być wcześniej utworzone, co jest często pracochłonne. Po drugie, przechowywanie map w pamięci znacznie obciąża maszynę. Z tego powodu praktycznie niemożliwe jest przechowywanie dużej liczby szczegółowych map trójwymiarowych, które często są przy-

* Katedra Automatyki, Akademia Górniczo-Hutnicza w Krakowie

datne. Konsekwencją tych wad jest uproszczenie złożonych scen, polegające na wielokrotnym użyciu tych samych danych, co nie wpływa dobrze na jakość utworzonych obrazów.

Często rozwiązaniem tego problemu jest zastosowanie teksturowania proceduralnego. Polega ono na wykonaniu, zwykle krótkiego, programu określającego kolor mapy w danym punkcie. Przykład proceduralnej tekstury znajduje się na rysunku 2. Proceduralne teksturowanie, mimo że pozwala uzyskać minimalnym kosztem ogromną różnorodność obiektów na scenie, także nie jest pozbawione wad. Po pierwsze, nie zawsze daje się zastosować (np. przykład z rys. 1). Po drugie, znacznie zwiększa czas renderingu. Odczytanie żądanej wartości z mapy jest znacznie szybsze niż wykonanie nawet prostego programu. Z tych względów obecnie niemal zawsze stosuje się podejście hybrydowe – języka do teksturowania proceduralnego z dostępem do funkcji operujących na mapach.



Rys. 2. Przykład tekstury proceduralnej

1.2. Teksturowanie a cieniowanie

Aby uniknąć nieporozumień, należy wyraźnie rozróżnić pojęcia teksturowania i cieniowania. Często używane są one zamiennie, co nie jest całkowicie poprawne. Poprzez cieniowanie rozumie się określenie ostatecznego koloru renderowanego fragmentu obrazu, gotowego do umieszczenia w buforze ramki. Zatem cieniowanie łączy obliczanie własności materiału w danym punkcie z przybliżonym obliczaniem oświetlenia tego punktu. Teksturowanie natomiast polega jedynie na obliczaniu własności materiału.

Rozróżnienie to wiąże się z klasami algorytmów renderingu, wykorzystujących proceduralne cieniowanie bądź teksturowanie. Cieniowanie może być wykorzystane tylko i wyłącznie w algorytmach używających tzw. lokalnego modelu oświetlenia. Model ten oblicza oświetlenie pochodzące tylko bezpośrednio od źródeł światła, całkowicie pomijając odbicia wielokrotne. Uwzględnienie tych odbić nie jest możliwe przy cieniowaniu, gdyż program cieniujący ma dostęp tylko do informacji o cieniowanym punkcie (np. jego współrzędnych czy normalnej do powierzchni) oraz niewielkiej liczbie stałych (np. położenia i moce źródeł światła). Z drugiej strony, algorytmy symulujące globalne oświetlenie obliczają je za pomocą niemodyfikowalnych, zapisanych na stałe algorytmów opartych na prawach fizyki. Jakakolwiek ingerencja w obliczane przez nie oświetlenie zaburza ich pracę, zatem algorytmy tej klasy mogą korzystać wyłącznie z teksturowania.

1.3. Języki funkcyjne

W uproszczeniu język funkcyjny oznacza język bez jawnych deklaracji zmiennych, a więc też bez możliwości używania obiektów przechowujących określony stan. Przykładem języka funkcyjnego jest Haskell [5]. Programy napisane w języku funkcyjnym mogą być zbudowane jedynie z deklaracji i wywołań funkcji. Zatem nie jest możliwe użycie konstrukcji iteracyjnych (wymagających logicznego warunku stopu, określonego na co najmniej jednej zmiennej). Konstrukcje takie są zastępowane rekurencyjnym wywoływaniem funkcji. Funkcje mogą używać jedynie stałych i parametrów formalnych oraz zwracać jedną wartość, więc tzw. efekty uboczne wywołania funkcji nie są dozwolone. Zatem wywołanie funkcji z takimi samymi argumentami zawsze zwraca taki sam wynik, co pozwala na stosowanie bardzo sprawnych metod optymalizacji.

Przedstawiony tutaj język funkcyjny stosowany jest do teksturowania, zatem nie musi posiadać pełnej funkcjonalności typowych języków funkcyjnych. W szczególności, funkcje nie mogą zwracać innych funkcji, nie można tworzyć własnych (złożonych) typów danych oraz nie ma zdefiniowanych operacji na łańcuchach znaków.

Ideą języków imperatywnych jest opisanie kolejnych zadań do wykonania. Sprawdzają się one dobrze jako języki ogólnego przeznaczenia. Można w nich opisać zarówno typowe funkcje matematyczne, jak i dowolne inne zadania. Jednakże w przypadku, gdy programy zawsze sprowadzają się wyłącznie do obliczenia pewnej wartości będącej funkcją z góry zdefiniowanych zmiennych i dodatkowych stałych, języki funkcyjne mogą być znacznie wygodniejsze. Znika w nich trudność związana z kolejnością obliczeń i błędami wynikającymi z niewłaściwego stanu zmiennych pomocniczych. Sytuacja taka ma miejsce w przypadku teksturowania, więc język funkcyjny do tego celu jest rozsądnym wyborem.

Poniższy przykład, przedstawiający funkcję silnia pozwala szybko zauważyć różnice pomiędzy programem napisanym w języku imperatywnym a funkcyjnym:

```
int silnia(int x) {
  int s=1;
  for (int i=2; i<=x; ++i) s*=i;
  return s;
}
```

```
int silnia(int x)
return cond(x>1,x*silnia(x-1),1)
```

Bardziej zaawansowane przykłady zostaną przedstawione w dalszej części niniejszego artykułu.

2. Inne języki teksturowania i cieniowania

Rendering, a w szczególności rendering w czasie rzeczywistym, przez długi czas wykorzystywał zaprogramowane na stałe algorytmy cieniowania, pozwalające na sterowanie wyglądem powierzchni za pomocą kilku parametrów. Popularnym, choć niezbyt realistycznym modelem była suma odbicia matowego, o współczynniku określonym przez teksturę, i białego połysku z regulowanym stopniem połyskliwości. Nieco nowsze podejścia pozwalały na stosowanie podstawowych operacji arytmetycznych na kilku teksturach. Metody te jednak nigdy nie były w stanie oddać olbrzymiej różnorodności rzeczywistych materiałów.

Pierwszym dobrze znanym podejściem do stworzenia elastycznego modelu cieniowania są drzewa cieni [1]. Pozwalają one na wyrażenie algorytmu za pomocą drzewa operacji arytmetycznych, w którego liściach mogą być umieszczane m.in. stałe, mapy albo parametry cieniowanego punktu. Jednakże, drzewo cienia może być widziane jako bardzo prosty program. Od czasu publikacji drzew cienia powstało wiele dużo bardziej złożonych języków przeznaczonych do opisu wyglądu renderowanych powierzchni. Popularny język cieniowania jest zawarty w systemie RenderMan [8]. Został on zaprojektowany tak, aby był wystarczający do tworzenia fotorealistycznych obrazów. Typowymi językami używanymi do programowania renderingu sprzętowego są Cg [4] oraz GLSL [9]. Języki te mają składnię podobną do C, ale są kompilowane na specjalizowane procesory graficzne. Kompilatory Cg i GLSL zawierają zbyt dużo poważnych ograniczeń, aby były one postrzegane jako uniwersalne języki cieniowania, ale te ograniczenia nie są wbudowane w same definicje języków. Są one raczej wymuszone poprzez niedoskonałości współczesnych procesorów graficznych, a więc siła języków Cg i GLSL prawdopodobnie zwiększy się w niedalekiej przyszłości.

Języki funkcyjne były już wcześniej wykorzystywane do teksturowania [6]. Jednakże podejście to wykorzystuje gotowy język funkcyjny Clean, za pomocą którego została utworzona biblioteka z narzędziami do syntezy i przetwarzania obrazów. Język ten nie jest zintegrowany z żadnym programem do renderingu, a utworzony system potrafi wygenerowane obrazy zapisywać do bitmap. Inny język – Vertigo [3] jest funkcyjnym językiem do programowania procesorów graficznych. Programy napisane w nim kompilowane są do asemblera w standardzie DirectX.

Skrypty teksturowania w prezentowanym języku, napisane bez użycia definicji i wywołań nowych funkcji, są równoważne drzewom cienia. Zdefiniowanie dodatkowych funkcji umożliwia użycie rekurencji. Rekurencja, połączona z warunkowym wyborem gałęzi drzew do obliczenia, pozwala na zapisanie dowolnie złożonych programów w stylu języka funkcyjnego. Biblioteka standardowa prezentowanego języka jest wzorowana na bibliotekach Cg oraz GLSL. Zawiera zatem podstawowe funkcje ułatwiające pisanie programów przeznaczonych do teksturowania. Dodatkowo, ze względu na fakt, że prezentowany język jest interpretowany programowo, biblioteka ta może posiadać znacznie rozszerzoną funkcjonalność względem bibliotek przeznaczonych do języków programowania procesorów graficznych.

3. Projekt języka

Przy projektowaniu języka teksturowania bardzo ważne są założenia dotyczące środowiska, w jakim ma on być wykonywany. Prezentowany język ma współpracować z algorytmami globalnego oświetlenia opartego na śledzeniu promieni. Podstawową funkcją programu zatem jest określanie odbicia promieni od opisywanej powierzchni oraz emisji światła z niej pochodzącej. Ma to wpływ na wszystkie elementy języka, opisane w kolejnych podrozdziałach.

3.1. Składnia i semantyka

Składnia języka funkcyjnego jest bardzo prosta. Program w nim napisany składa się wyłącznie z definicji i wywołań funkcji. Każda funkcja przyjmuje dowolną liczbę parametrów i musi zwracać dokładnie jedno wyrażenie. Funkcje mogą korzystać ze swoich argumentów, stałych oraz danych ze środowiska zewnętrznego (parametrów teksturowanego punktu oraz dodatkowych stałych, modyfikujących zachowanie skryptu). Wyrażenia tworzone są za pomocą standardowych operatorów arytmetycznych, relacyjnych i logicznych oraz wywołań innych funkcji, zarówno z biblioteki standardowej, jak i zdefiniowanych w skrypcie. Nie istnieje operator przypisania, a instrukcja warunkowa jest funkcją zdefiniowaną w bibliotece standardowej. Wszystkie operatory i większość funkcji są przeciążane tak, aby działały dla liczb rzeczywistych, tablic o zmiennej długości oraz kolorów. Bibliotekę standardową można rozszerzać o dodatkowe funkcje, napisane w innym, kompilowanym języku programowania. W tym celu w skrypcie należy umieścić jedynie nagłówek funkcji, a zamiast zwracanego wyrażenia podać nazwę pliku z biblioteką dynamiczną zawierającą rozszerzenie.

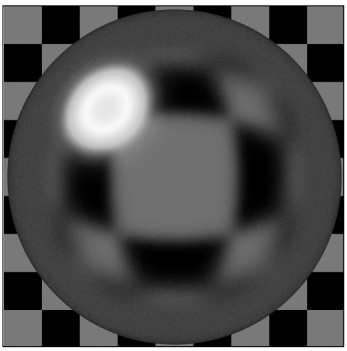
Za obliczanie współczynnika odbicia odpowiadają materiały. Dostępne są one jako funkcje w bibliotece standardowej. Materiały określają jedynie, czy powierzchnia jest np. matowa, półprzepuszczalna czy połyskliwa. Definicje materiałów nie są więc kompletne, a brakujące dane uzupełniane są przez tzw. źródła kolorów i wartości skalarnych. Zwiększa to elastyczność skryptów, gdyż np. ten sam materiał matowy może mieć stały kolor, używać mapy, czy też posiadać proceduralnie określony wzór. Emisja z powierzchni opisywana jest w podobny sposób za pomocą sztuczki z użyciem materiałów. Materiał normalnie oblicza współczynnik odbicia, biorąc pod uwagę kierunek promienia padającego i odbitego. Przekazując jako promień padający wektor równoległy do normalnej do powierzchni, można użyć materiałów do opisania emisji.

3.2. Biblioteka standardowa

Biblioteka ta zawiera bogaty zestaw funkcji przydatnych do pisania skryptów teksturujących. Funkcje te można podzielić na zestaw materiałów, źródeł skalarów, kolorów, tablic skalarów i kolorów oraz operatorów.

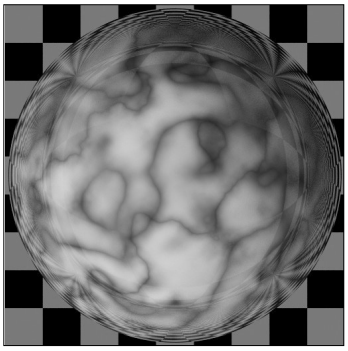
Podstawowe materiały, które są w bibliotece, to: materiał matowy odbijający, matowy przepuszczalny, połyskliwy o regulowanym stopniu połyskliwości, idealny lustrzany oraz idealny załamujący. We wszystkich tych materiałach należy określić dodatkowo źródło koloru, a w przypadku materiału połyskliwego jeszcze stopień połyskliwości. Materiały można łączyć, używając materiałów złożonych. Materiał złożony wymaga podania dwóch materiałów oraz współczynnika ich kombinacji. Na rysunkach 3 i 4 przedstawione są przykładowe materiały wraz z odpowiadającymi im skryptami.

Biblioteka standardowa oferuje rozbudowane wsparcie operacji na obrazach z plików zewnętrznych. Obrazy te można przechowywać w pamięci zarówno w wersji podstawowej, jak i spakowanej (oszczędność pamięci kosztem czasu dostępu). Obrazy można odczytywać w trybie bez żadnych modyfikacji oraz z filtrem rozmazującym.



```
material sphere {
return mtgloss_(
  cs=color(0.75)
  gloss=color([32, 128])
)
}
```

Rys. 3. Materiał połyskliwy



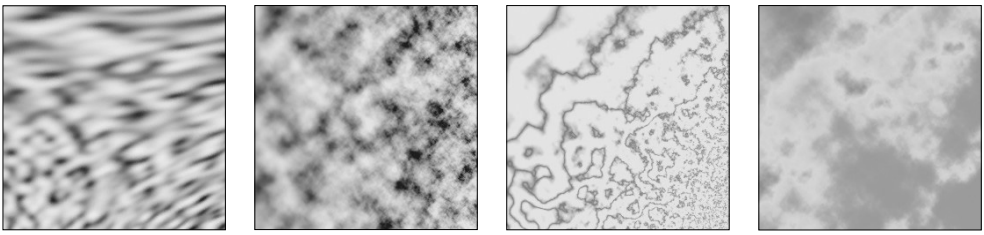
```
material sphere {
return mtcomplex(
  mt1=mtspecrefl(cs=color(1))
  mt2=mtdiffuse(cs=marble)
  ss=fresnel(n=1.4, k=0)
)}
```

Rys. 4. Marmur będący kombinacją liniową odbicia matowego z proceduralnie określonym wzorem i odbicia lustrzanego. Jako współczynnik kombinacji liniowej użyte zostało równanie Fresnela opisujące odbicie światła od gładkiej powierzchni

Biblioteka definiuje też funkcje liczące gradient na zadanych obrazach. Obrazy mogą być zarówno kolorowe, jak i w skali szarości.

Kolejnym zestawem funkcji jest szum Perlina [2, 7] oraz oparte na nim tekstury proceduralne symulujące naturalne materiały, takie jak drewno czy marmur. Biblioteka standardowa posiada obszerny zestaw funkcji tego typu. Wyniki przykładowych funkcji umieszczone są na rysunku 5. Należy zaznaczyć, że funkcje te są sparametryzowane i pewne cechy obrazów mogą być łatwo modyfikowane. Wybrane cechy prezentowanych obrazów proceduralnych są płynnie modyfikowane w poziomie i w pionie.

Dodatkowo biblioteka zawiera kilka użytecznych funkcji opartych na równaniach fizycznych. Funkcje te to promieniowanie ciała doskonale czarnego oraz odbicie Fresnela od gładkich powierzchni metalicznych i niemetalicznych. Funkcje te przydają się do poprawiania realizmu renderingu.



Rys. 5. Standardowe obrazy proceduralne. Od lewej: szum, szum Perlina, marmur, chmury. Każdy obraz ma płynnie modyfikowane wybrane parametry

3.3. API interpretera

Kolejnym zagadnieniem, koniecznym do uzyskania poprawnej współpracy pomiędzy interpreterem skryptów a algorytmem renderingu jest określenie API, poprzez które algorytm ten może sterować interpreterem.

Aby algorytm renderingu mógł skorzystać z funkcji interpretera, musi utworzyć obiekt 'Graf Struktury Materiałów'. Pusty graf jest tworzony bezparametrową funkcją. Następnie, graf ten jest zapełniany poprzez przetwarzanie skryptów. W jednym grafie można umieścić dowolnie wiele skryptów pod warunkiem, że nazwy ich funkcji nie pokrywają się. Rezultat ostatniego przetwarzania (na przykład raport o błędach) można odczytać z grafu za pomocą odpowiedniej funkcji.

Następnie, algorytm musi utworzyć obiekt 'Program'. Podobnie jak graf, pusty program tworzony jest odpowiednią bezparametrową funkcją. Do programu można dokładać dowolną liczbę funkcji zbudowanych na podstawie grafu. Funkcje określane są według nazw. Podobnie jak przetwarzanie skryptu, budowanie funkcji może się nie powieść. Raport z ostatniej budowy można odczytać z obiektu 'Program' za pomocą odpowiedniej funkcji. Bezpośrednio po zbudowaniu żądanych funkcji, obiekt 'Graf' może zostać usunięty. Program usunąć można jednak dopiero po zakończeniu korzystania z jego funkcji.

Przetwarzanie dwuetapowe zostało wybrane ze względu na jego większą elastyczność. Na przykład, skrypty A i B mogą korzystać z tej samej funkcji, która jest zakodowana jednokrotnie w skrypcie C. Dodatkowo, jeżeli przetwarzany skrypt zawiera funkcję, która nigdy nie zostanie zbudowana, może zostać poprawnie przetworzony, nawet jeżeli ta funkcja zawiera błędy (na przykład odwołanie do innej, niezdefiniowanej funkcji).

Poniższy przykład prezentuje działanie przedstawionych powyżej funkcji:

```
MMsGraph* g = mmsMkGraph();
if (!mmsParseString(g, script)) throw Error(mmsGetGraphLog(g));
if (!mmsParseFile(g, "marble.txt")) throw Error(mmsGetGraphLog(g));
MMsProgram* program = mmsMkProgram();
Material* mat = mmsMkMaterial(g, "matname", program);
if (!mat) throw Error(mmsGetProgramLog(program));
mmsDeleteGraph(g);
//… tu można użyć zbudowanego materiału
mmsDeleteProgram(program);
```

Przykład ten tworzy materiał o nazwie „matname" używając skryptów z łańcucha „script" oraz z pliku „marble.txt".

4. Wykonywanie programów

Na podstawie skryptów budowana jest reprezentacja grafowa wywołań funkcji i operatorów. Jeżeli nie występują wywołania rekurencyjne, to graf ten jest drzewem. Konstrukcja funkcji za pomocą grafu odbywa się inaczej dla drzew, a inaczej dla grafów z cyklami. W pierwszym przypadku interpreter zestawia żądaną funkcję z wywołań wewnętrznych funkcji wirtualnych, odpowiadających operatorom, stałym, odwołaniom do zmiennych zewnętrznych oraz funkcjom bibliotecznym. Nierekurencyjne funkcje zdefiniowane w skrypcie są zawsze wstawiane w miejscu wywołania.

W przypadku rekurencji nie da się wykonać wstawiania, więc dla każdej funkcji użytkownika, która bezpośrednio albo pośrednio wywołuje siebie, jest tworzony obiekt ramki stosu. Obiekt ten posiada pola będące uchwytami do parametrów funkcji zdefiniowanej w skrypcie oraz uchwyt do wyrażenia obliczanego w tej funkcji. Obliczenie wyrażenia objętego ramką stosu odbywa się następująco. W pierwszym kroku obliczane są wartości parametrów, a wyniki umieszczane są na stosie. Potem obliczane jest wyrażenie użytkownika, z podstawionym odwołaniem do wierzchołka stosu, zamiast do nieobliczonych parametrów. Na koniec zwalniana jest pamięć na stosie i zwracany jest wynik wyrażenia użytkownika.

5. Dodatkowy przykład

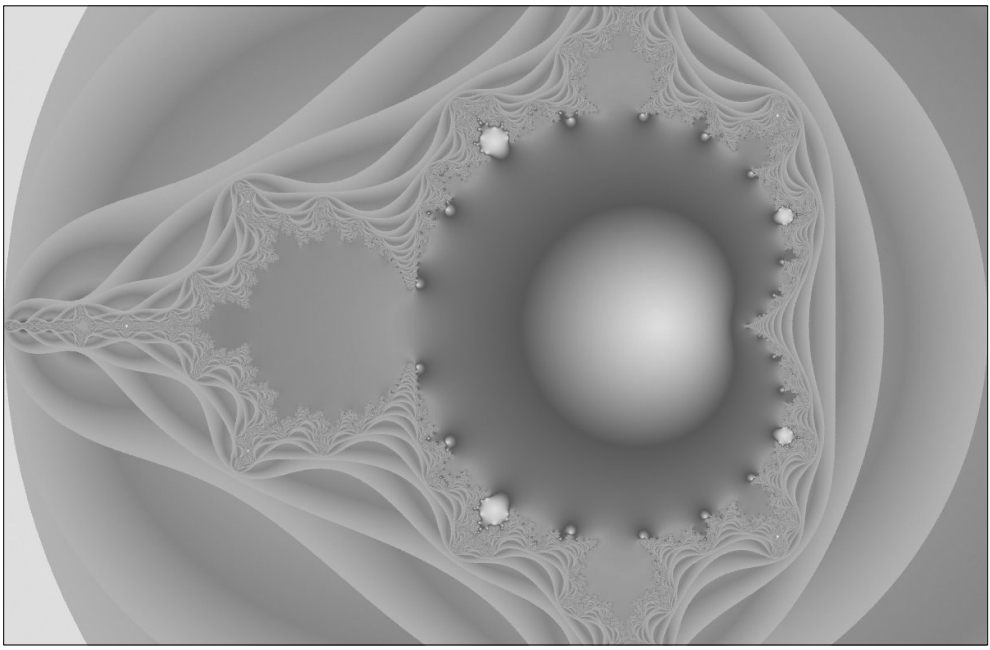
Do tej pory wszystkie przykłady programów były bardzo proste. Sprowadzały się one wyłącznie do prostych obliczeń albo wywołań funkcji bibliotecznych. Przykład prezentowany w tym rozdziale demonstruje fraktal Mandelbrota jako teksturę proceduralną opisaną w prezentowanym języku. Na rysunku 6 przedstawiony jest wynik działania programu.

Obraz został wygenerowany za pomocą następujących funkcji:

```
color main {
 return blend(
 cs = [color([0.2 0.6]), color([0.2 0.6 0.1]),
 color([0.7 0.2 0.1]), color([0.1 0.2 0.7 0.4]),
 color([0.1 0.1 0.3 0.7]), color([0.9 0.65])]
 alpha = 2^(-mandelbrot(addr=$uv*[3,2]-[2.0,1.0])))
}

scalar mandelbrot(scalar[] addr) {
 return norm_l2(mdb_rec(x=[0, 0, 0], addr=addr){0, 1})
}

scalar[] mdb_rec(scalar[] x, addr) {
return cond(
 sqr(x[0])+sqr(x[1])>4.0 or x[2]>24, x,
 mdb_rec(
 x=[sqr(x[0])-sqr(x[1])+addr[0], 2*x[0]*x[1] + addr[1], x[2]+1],
 addr=addr)
)}
```



Rys. 6. Fraktal Mandelbrota

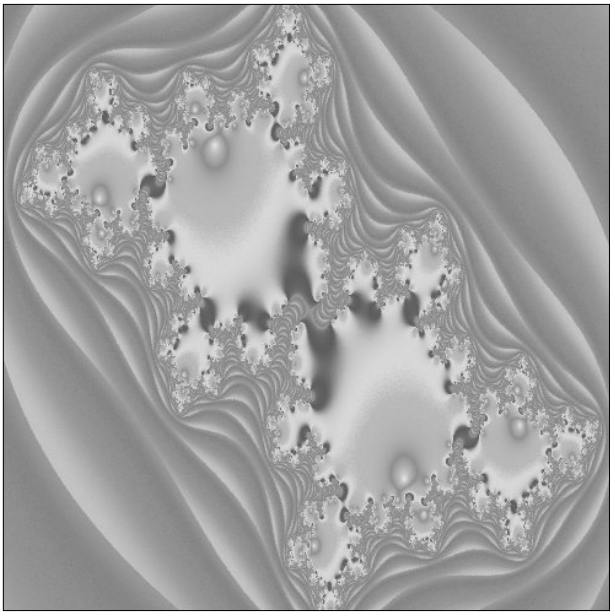
Funkcja `main` odpowiada za konwersję fraktala na kolor z zadanej palety. Biblioteczna funkcja `blend` oblicza płynne przejścia miedzy kolorami, sterowane wartością `alpha`. Funkcja wykładnicza zawęża przedział wartości fraktala do [0, 1]. Ze względu na to, że obraz wartości binarnych określających przynależność punktów do zbioru Mandelbrota nie wygląda szczególnie ciekawie, funkcja `mandelbrot` jako wartość fraktala określa moduł z wyliczonej w czasie testów przynależności liczby zespolonej. Instrukcja `{0, 1}` powoduje utworzenie nowego wektora z dwóch pierwszych elementów poprzedniego. Jako argument funkcji rekurencyjnej `mdb_rec` przekazywany jest wektor złożony z części rzeczywistej testowanej liczby, części urojonej, oraz licznika iteracji. Jest to dobry przykład na zastąpienie iteracji rekurencją.

W podobny sposób można wygenerować fraktal Julii. W tym celu zostały użyte następujące funkcje:

```
scalar julia(scalar[] addr) {
 return norm_l2(jul_rec(x=[addr[0], addr[1], 0]){0, 1})
}

scalar[] jul_rec(scalar[] x) {
return cond(
 sqr(x[0]) + sqr(x[1]) > 4 or x[2]>24, x,
 julr(x=[sqr(x[0])-sqr(x[1]) - 0.4, 2.0*x[0]*x[1] + 0.6, x[2]+1]))
}
```

Na rysunku 7 przedstawiony jest wynik ich działania. Funkcja `main` jest analogiczna do wersji generującej fraktal Mandelbrota. Argument `x` i wartość zwracana z funkcji rekurencyjnej zawiera kolejno: część rzeczywistą ciągu liczb, którego zbieżność jest badana, jego część urojoną oraz licznik iteracji. Wartością zwracaną z funkcji nierekurencyjnej jest moduł z obliczonej liczby zespolonej.



Rys. 7. Fraktal Julii

6. Wnioski

Nowością w prezentowanym tutaj języku jest pełna integracja języka funkcyjnego z algorytmami globalnego oświetlenia opartego o śledzenie promieni. Skrypty w nim napisane zapewniają automatyczne sterowanie odbiciami promieni od definiowanych powierzchni oraz określenie ich absorpcji, a zatem postrzeganego koloru. Kolor ten opisywany jest za pomocą pełnego widma, co umożliwia m.in. poprawne modelowanie materiałów dyspersyjnych i powierzchni metalicznych.

Prezentowany język jest bardzo wygodnym narzędziem do teksturowania proceduralnego. Pewnym jego niedostatkiem jest fakt, że jest on interpretowany. W przyszłości może on jednak zostać zmodyfikowany na język kompilowany, np. do kodu procesorów x86, co znacznie zwiększy szybkość działania. Modyfikacja taka oczywiście nie będzie się wiązać z jakąkolwiek zmianą definicji języka, więc wszystkie skrypty zachowają taką samą postać.

Porównanie wydajności prezentowanego języka z innymi językami o podobnym przeznaczeniu prawdopodobnie nie jest możliwe. Autor, pomimo intensywnych poszukiwań, nie napotkał rozwiązania oferującego podobne możliwości, co było główną motywacją wykonania opisywanego oprogramowania. Testy wskazują jednak, ze wydajność ta jest nieistotna w porównaniu z czasem działania globalnego oświetlenia. W praktycznie nieistotnym przypadku, gdy scena zawiera kilka figur, złożone skrypty mogą podwoić czas renderingu względem jednolicie szarych powierzchni. Jednak, gdy scena jest złożona z setek tysięcy figur, wydłużenie skryptów nie spowalnia zauważalnie czasu tworzenia obrazu.

Literatura

[1] Cook R.L., *Shade Trees.* ACM Siggraph Computer Graphics. t. 18, 1984, 223–231.
[2] Ebert D.S., Musgrave F.K., Peachey D., Perlin K., Worley S., *Texturing and Modeling: A Procedural Approach.* 3rd ed., Morgan Kaufmann, San Francisco, 2003.
[3] Elliot C., *Programming Graphics Processors Functionally*. Proceedings of the 2004 Haskell Workshop, ACM Press 2004.
[4] Fernando R., Kilgard M.J., *Jezyk Cg. Programowanie grafiki w czasie rzeczywistym.* Helion, Gliwice 2003.
[5] Hudak P., Hughes J., Jones S.P., Walder P., *A History of Haskell: Being Lazy with Class.* History of Programming Language Conference (HOPL), 2007, 1–55.
[6] Kaczmarczyk J., *Functional Approach to Texture Generation*. Practical Aspects of Declarative Languages (PADL), Portland, Springer LNCS 2257, 2002, 225–242.
[7] Perlin K., *An image synthesizer.* Proceedings, t. 19, Siggraph 1985, 287–296.
[8] Pixar: *The RenderMan Interface*. Pixar 2005.
[9] Roost R.J., *OpenGL Shading Language.* 2nd ed., Addison Wesley, 2006.