

<b>Języki Asemblerowe</b>						
Rok akademicki	Termin	Rodzaj studiów	Kierunek	Prowadzący	Grupa	Sekcja
<b>2008/2009</b>	poniedziałek	<b>Dzienne</b>	<b>Inf</b>	<b>AO</b>	<b>1</b>	<b>1</b>
	9:30-11:00					

## **Projekt - Sprawozdanie**

Temat ćwiczenia:

**Raytracer 4-wymiarowych fraktali Julia**

**Tomasz bla Fortuna**

# 1 Temat

Tematem projektu było napisanie programu generującego statyczne obrazy rzutu czterowymiarowych fraktali Julia na przestrzeń trójwymiarową.

## 1.1 Założenia

- Wykorzystanie jednostki SIMD do obliczeń zmiennoprzecinkowych.
- Napisanie programu na architekturę x86.
- Podział rozwiązania problemu na dowolną liczbę wątków.
- Napisanie programu w sposób umożliwiający jego następne przeniesienie na system Windows oraz platformę x86\_64.

## 2 Analiza zadania

Zbiór Julii tworzą punkty  $P(x, y, z, w) \in \mathbb{Z}$  takie, że ciąg:

$$\begin{cases} z_0 &= x + yi + zj + wk \\ z_{n+1} &= z_n^2 + c \end{cases} \quad (1)$$

nie dąży do nieskończoności.

Parametr fraktala  $c$ , zdefiniowany jest następująco:

$$c = c_1 + c_2i + c_3j + c_4k$$

### 2.1 Metody renderowania

Fraktale tego typu można renderować na dwa sposoby:

1. Poprzez obliczanie pola skalarnego fraktala (obliczając z zadaną ziarnistością granicę ciągu ?? dla fragmentu przestrzeni, w której powinien znajdować się fraktal). Następnie skonwertować na siatkę trójkątów za pomocą algorytmu maszerujących sześcianów (*marching cubes*) oraz wyświetlić za pomocą np. *OpenGL*.
2. Za pomocą algorytmu śledzenia promieni.

Wybrałem drugi z algorytmów ze względu na jego liczne zalety. Jest szybszy, generuje obrazy bez widocznej ziarnistości związanej z doбором rozdzielczości próbkowania pola skalarnego oraz można go w całości zaimplementować samemu bez uciekania się do bibliotek takich jak *OpenGL*. Daje się go również zrównoleglać o wiele wygodniej niż algorytm maszerujących sześcianów.

## 2.2 Algorytm

1. Wygenerowanie kwaternionów opisujących kamerę.
2. Dla każdego piksela ekranu  $(x, y)$  szukamy przecięcia z fraktalem:
  - a) Stwórz promień (para  $k_p, k_d$  — kwaternion początku + kierunku) dla zadanego punktu.
  - b) Dopóki kwaternion początku znajduje się w pewnym otoczeniu fraktala oraz jego odległość od fraktala jest większa od  $\epsilon$  powtarzaj:
    - i. Oszacuj granicę ciągu ??.
    - ii. Określ odległość kwaternionu początku od fraktala i przesuń ten kwaternion w kierunku promienia:  $k_P = k_p + k_d * d$
  - c) Jeśli znaleziono kolizję oblicz wektor normalny i zapisz kolor wynikający z modelu *Phong*.
  - d) Jeśli nie ma kolizji przypisz pikselowi kolor czarny.
3. Wyświetl wygenerowany obraz

## 3 Specyfikacja zewnętrzna

Program jest uruchamiany z linii poleceń, w której można zdefiniować rozdzielczość generowanego ekranu oraz liczbę wątków na jaką ma być podzielona praca. Następnie program uruchamia wątki, dzieli zadany obraz na fragmenty i uruchamia rendering.

Przykład uruchomienia:

```
% ./Main.x86_asm_sse 640 480 8
```

```
Rendering 640x480 picture with 8 threads
Using hand-crafted SSE2 code
Rendering using: CPU only
Initializing display with 640x480 size
thread 0 = from (0 0) to (80 480) [size: 80x480]
thread 1 = from (80 0) to (160 480) [size: 80x480]
thread 2 = from (160 0) to (240 480) [size: 80x480]
thread 3 = from (240 0) to (320 480) [size: 80x480]
thread 4 = from (320 0) to (400 480) [size: 80x480]
thread 5 = from (400 0) to (480 480) [size: 80x480]
thread 6 = from (480 0) to (560 480) [size: 80x480]
thread 7 = from (560 0) to (640 480) [size: 80x480]
Realtime = 0.26905
```

Parametr  $c$  fraktala, parametry kamery (położenie, kierunek, FOV) oraz, w przypadku generowania ciągu klatek celem wyrenderowania animacji, ilość klatek, iterowany parametr, są na razie na sztywno zapisane w funkcji main. Planuję wczytywać je wszystkie z wiersza poleceń.

## 4 Specyfikacja wewnętrzna

Specyfikacja wewnętrzna (opis funkcji oraz struktur) została wygenerowana za pomocą *Doxygen* i jest dostępna na płycie z projektem.

Program jest napisany w języku C oraz Assembler. W trakcie kompilacji są generowane dwa pliki `.o` — jeden z części HLL, drugi z kodu assembler, które są ze sobą linkowane tworząc gotowy wynikowy program, w którym język C może bezpośrednio wołać funkcje assemblera.

Ponadto w trakcie kompilacji można zdefiniować za pomocą stałych parę parametrów programu.

- `USE_ASM`, włącza implementacje algorytmu w języku asm.
- `USE_CUDA`, włącza implementacje realizowaną na procesorze graficznym
- `USE_HALFCUDA`, ponadto dzieli obraz pomiędzy GPU i CPU w celu przyspieszenia obliczeń.

Na razie ustawienie flagi `USE_ASM` wymaga (cross)kompilacji na platformę x86, natomiast ze względu na ograniczenia użytego SDK CUDA, włączenie `USE_CUDA` wymaga użycia platformy na którą był kompilowany SDK. W moim przypadku jest to x86\_64.

Do wyświetlania grafiki program używa biblioteki *SDL*.

Za celowe uważam umieszczenie głównego fragmentu kodu obliczającego wartość ciągu. W wielu miejscach gdzie to było możliwe przedzielałem instrukcje, których argumenty były zależne od wyników poprzedniej instrukcji innymi, w celu ułatwienia procesorowi równoległego wykonywania zadań. Ze względu na to, że utrudnia to odczytywanie kodu starałem się obficie te zmiany komentować

```
1      ;;-----
2      ;; Functions steps the Julia equation
3      ;;  $d_{\{n+1\}} = z_n * d_n * 2.0$  (1)
4      ;;  $z_{\{n+1\}} = z_n^2 + c$  (2)
5      ;; And then estimates distance from Julia
6      ;; and returns result in st0
7 julia_limit:
8     push ebp
9     mov  ebp, esp
10
11     prefetch [julia_d0]
12     ;; Data fetch
13     ;; xmm6 - z bound across loop execution
14     ;; xmm7 - d
15     ;; 00 01 10 11
16     ;; z.x, z.y, z.z, z.w
17     ;; d.x, d.y, d.z, d.w
18
19 .prologue:
20     ;; Read z0 from stack
```

```

21     mov ecx, [8+ebp]
22     movupd xmm6, oword [ecx]
23
24     ;; Initialize d with d0
25     movapd xmm7, oword [julia_d0]
26
27     movapd xmm3, oword [julia_mul1]
28     movapd xmm4, oword [julia_mul2]
29     movapd xmm5, oword [julia_mul3]
30
31     ;; Limit loop
32     mov ecx, [julia_iter]
33     prefetch [julia_inf]
34 .loop:
35     dec ecx
36     push ecx
37
38     ;; Calculate d_{n+1}
39 .equat1:
40 ;
41 ;           (1)           (2)           (3)           (4)
42 ;   x = a.x * b.x - a.y * b.y - a.z * b.z - a.w * b.w; 00
43 ;   y = a.x * b.y + a.y * b.x + a.z * b.w - a.w * b.z; 01
44 ;   z = a.x * b.z - a.y * b.w + a.z * b.x + a.w * b.y; 10
45 ;   w = a.x * b.w + a.y * b.z - a.z * b.y + a.w * b.x; 11
46 ;   order in register: 11 10 01 00
47
48     ;; Prepare set of A.x and multiply them by b.x, b.y, b.z, b.w
49     pshufd xmm0, xmm6, 00000000b ; (1) set of a.x
50     pshufd xmm1, xmm6, 01010101b ; (2) set of a.y
51     mulps xmm0, xmm7 ; (1)
52     xorps xmm1, xmm3 ; (2) change sign
53
54     pshufd xmm2, xmm7, 10110001b ; (2) "swap in pairs"
55     mulps xmm2, xmm1 ; (2)
56     pshufd xmm1, xmm6, 10101010b ; (3) set of a.z
57     addps xmm0, xmm2 ; (1+2)
58     xorps xmm1, xmm4 ; (3) change sign
59
60
61     pshufd xmm2, xmm7, 01001110b ; (3) "swap in qwords"
62     mulps xmm2, xmm1 ; (3)
63     pshufd xmm1, xmm6, 11111111b ; (4) set of a.w
64     addps xmm0, xmm2 ; (1+2+3)
65     xorps xmm1, xmm5 ; (4) change sign
66
67
68     pshufd xmm7, xmm7, 00011011b ; (4) "swap in qwords and in pairs"
69

```

```

70     mulps xmm7, xmm1                ; (4)
71     addps xmm7, xmm0                ; (1+2+3+4)
72
73     addps xmm7, xmm7                ; * 2
74
75     prefetch [julia_zmul]
76     prefetch [julia_c]
77 .equat2:
78     ; x = 1 * x * x - y*y - z*z - w*w + c    (1)
79     ; y = 2 * x * y + c                      (2)
80     ; z = 2 * x * z + c                      (3)
81     ; w = 2 * x * w + c                      (4)
82
83     pshufd xmm0, xmm6, 00000000b        ;; Create a set of z.x
84     mulps xmm0, [julia_zmul]           ;; Multiply by 1, 2, 2, 2
85     mulps xmm0, xmm6                   ;; Multiply by z.x, z.y, z.z, z.w
86
87     mulps xmm6, xmm6                   ;; Square x, y, z and w (x dismissed)
88     addps xmm0, [julia_c]              ;; Add c constant
89
90     psrldq xmm6, 4                      ;; extract z.y^2
91     subss xmm0, xmm6                   ;; sub z.y^2
92
93     psrldq xmm6, 4                      ;; extract z.z^2
94     subss xmm0, xmm6                   ;; sub z.z^2
95
96
97     ;; extract z.z^2
98     psrldq xmm6, 4
99     ;; sub z.w^2
100    subss xmm0, xmm6
101
102    ;; place it back in xmm6
103    movaps xmm6, xmm0
104
105    ;; Calculate zlen and check if we can break free
106    ;; Square
107    mulps xmm0, xmm0
108    movhps xmm1, xmm0
109    ;; Sum two pairs
110    addps xmm0, xmm1
111    pshufd xmm1, xmm0, 01010101b
112    ;; Sum rest
113    addss xmm0, xmm1
114
115    ;; if smaller continue
116    comiss xmm0, [julia_inf]
117    jb .loop_epilogue
118

```

```

119         pop ecx
120         jmp .epilogue
121
122
123 .loop_epilogue:
124         pop ecx
125         or ecx, ecx
126         jnz .loop
127
128 .epilogue:
129         ;; Calculate zlen
130         sqrtss xmm0, xmm0
131
132         ;; Calculates dlen given 'd' in xmm7 register
133
134 .julia_get_dlen:
135         mulps xmm7, xmm7
136         movhyps xmm5, xmm7
137         addps xmm7, xmm5
138         pshufd xmm5, xmm7, 01010101b
139         addss xmm7, xmm5
140         sqrtss xmm7, xmm7
141
142 .get_distance:
143         ;; xmm0 = zlen; xmm7 = dlen
144         ;; dist = log(zlen) * zlen / (2*dlen);
145
146         ;; Logarithm:
147         ;; a(x) = (x-1) / x
148         ;; log(x) = a(x) * ( 1 + 0.5*a(x) )
149         ;; Simplified:
150         ;; dist = (zlen-1)/zlen * zlen * (1+0.5 * a) / (2*dlen);
151         ;; = (zlen-1) * (1 + 0.5*(zlen-1)/zlen) / (2*dlen)
152
153         ;; (1) Constants
154         ;; (2) xmm1 = zlen - 1
155         ;; (3) xmm3 = (zlen-1) / zlen
156         ;; (4) xmm3 /= 2
157         ;; (5) xmm3 += 1
158         ;; (6) xmm1 = (zlen-1) * (...)
159         ;; (7) xmm1 /= (2*dlen)
160
161         movss xmm2, [one]           ; (1)
162
163         movss xmm1, xmm0           ; (2)
164         subss xmm1, [one]         ; (2)
165
166         movss xmm4, [two]         ; (1)
167

```

```

168     movss xmm3, xmm1           ; (3)
169     divss xmm1, xmm7           ; (7)
170     divss xmm3, xmm0           ; (3)
171     divss xmm1, xmm4           ; (7)
172
173     divss xmm3, xmm4           ; (4)
174
175     addss xmm3, [one]          ; (5)
176
177
178     mulss xmm1, xmm3           ; (6)
179     movss [esp-4], xmm1
180
181     ;;
182     ;; Leave return value as calling convention wants in st0
183     ;;
184     fld dword [esp-4]
185
186     pop ebp
187     ret

```

## 5 Problemy

Największym problemem, który trzeba było rozwiązać było wymyślenie dobrej metody wykonywania mnożenia kwaternionów za pomocą SSE. Istnieją dwa podstawowe podejścia.

$$\begin{cases} s_x &= z_x d_x - z_y d_y - z_z d_z - z_w d_w \\ s_y &= z_x d_y + z_y d_x + z_z d_w - z_w d_z \\ s_z &= z_x d_z - z_y d_w + z_z d_x + z_w d_y \\ s_w &= z_x d_w + z_y d_z - z_z d_y + z_w d_x \end{cases}$$

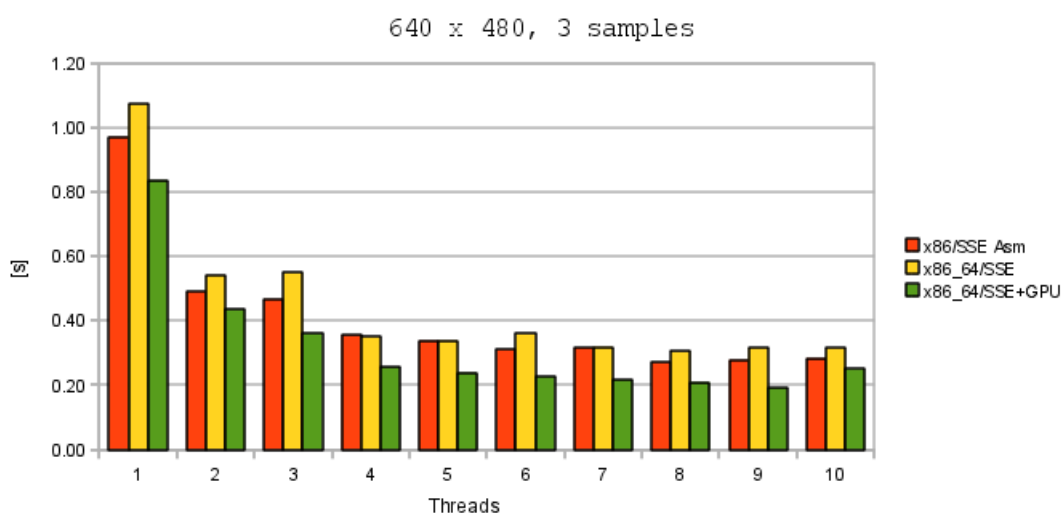
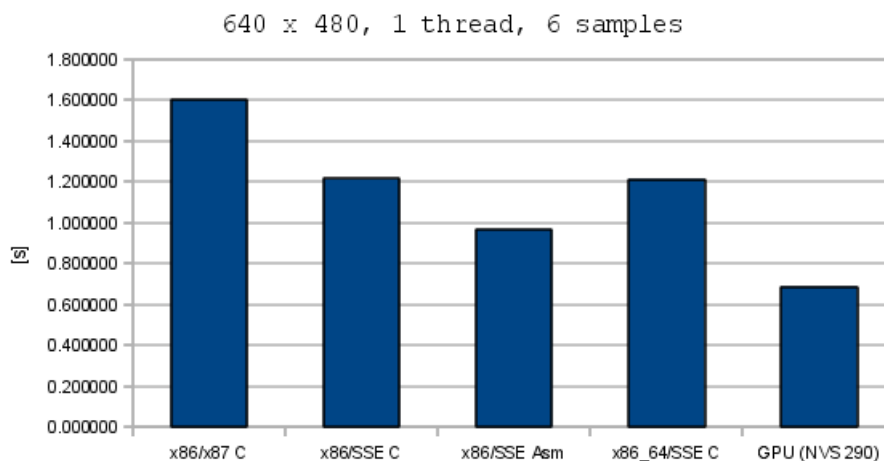
Pierwsze podejście zakłada obliczenie jednego mnożenia  $z * d$  na raz, i zrównoleglenie mnożeń występujących w równaniu. Jego wadą jest to, że wymaga częstych zmian kolejności wartości zmiennych przecinkowych znajdujących się w rejestrach xmm.

Drugie podejście zakłada zrównoleglenie jednoczesnego obliczania 4 równań.

Wybrałem pierwszą metodę, choć ze względu na optymalność obliczeń druga prawdopodobnie wyszła by znacznie lepiej. Druga metoda wymaga każdorazowego konwertowania danych przed wrzuceniem do SSE albo przechowywania ich przez cały czas nie jako kwaternionu, tylko jako współrzędnych x, 4 różnych kwaternionów. Uniemożliwia również optymalizacje polegające na przerwaniu obliczeń ciągu jeśli jego wartość dąży do nieskończoności - ponieważ inny obliczany w tym samym momencie ciąg mógłby mieć granicę właściwą.

## 6 Wnioski

Przetestowałem wszystkie skompilowane wersje tego programu pod kątem wydajności:



Obie zastosowane metody optymalizacji - podział na wątki, oraz przepisanie często wykonywanego kodu do assemblera i zoptymalizowanie za pomocą SSE - przyniosły widoczne rezultaty. Kod optymalizowany ręcznie wykonuje się znacznie szybciej od kodu generowanego przez kompilator - zarówno na x87 jak i na SSE2.

Nie można natomiast zauważyć znaczącej różnicy pomiędzy kodem generowanym przez kompilator na architekturę 32 bitową i 64 bitową — jeżeli w obu przypadkach korzysta on z SSE. Uważam jednak, że gdybym sam pisał kod na arch. x86\_64 mógłbym dodatkowo zwiększyć szybkość wykorzystując dostępne wtedy dodatkowe 7 rejestrów xmm.