
		Instytut Informatyki Politechniki Śląskiej Zespół Mikroinformatyki i Teorii Automatów Cyfrowych			
Rok akademicki	Rodzaj studiów*: SSI/NSI/NSM	Przedmiot: (Języki Asemblerowe/SMIW)		Grupa	Sekcja
2018/2019	SSI	Języki Asemblerowe		2	3
Prowadzący przedmiot:	mgr inż. Oleg Antemijczuk			Termin: (dzień tygodnia godzina)	
Imię:	Grzegorz			Wtorek	
Nazwisko:	Kazana			17:15-20:15	
Email:	grzekaz068@student.polsl.pl				
<i>Sprawozdanie z projektu</i>					
Temat projektu:					
<p>Projekcja wielościanów 3 i 4-wymiarowych na płaszczyznę 2 wymiarową, na przykładzie animacji obracającego się tesseractu.</p>					
Główne założenia projektu:					
<p>Implementacja algorytmu rotacji oraz rzutowania (ortogonalnego i perspektywistycznego) punktów w przestrzeniach 3 i 4-wymiarowych do przestrzeni 2 wymiarowej. Przedstawienie wyniku rzutowania za pomocą biblioteki pozwalającej na naniesienie punktów na płaszczyznę 2d.</p>					

1. Temat projektu

Projekcja wielościanów 3 i 4-wymiarowych na płaszczyznę 2 wymiarową, na przykładzie animacji obracającego się tesseractu.

2. Założenia

Implementacja algorytmu rotacji oraz rzutowania (ortogonalnego i perspektywicznego) punktów w przestrzeniach 3 i 4-wymiarowych do przestrzeni 2 wymiarowej. Przedstawienie wyniku rzutowania za pomocą biblioteki pozwalającej na naniesienie punktów na płaszczyznę 2d.

3. Wstęp teoretyczny

a) Rzutowanie

Punkty w przestrzeni 3 wymiarowej reprezentowane za pomocą wektora mogą zostać rzutowane na płaszczyznę na wiele sposobów. Sposobami wybranymi przeze mnie jest rzut prostokątny i perspektywiczny.

Rzut prostokątny zaniedbuje zjawisko perspektywy, lecz zachowuje równoległość linii obiektu. Operacja rzutowania może zostać wykonana poprzez przemnożenie wektora przez macierz rzutowania mającą postać:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Rzut perspektywiczny oddaje zjawisko perspektywy, poprzez zmniejszenie rozmiarów obiektu proporcjonalne do współrzędnej Z. Macierz rzutowania jest parametryzowana współrzędną Z oraz odległością kamery od obiektu, i ma postać:

$$\begin{bmatrix} \frac{1}{z-d} & 0 & 0 \\ 0 & \frac{1}{z-d} & 0 \end{bmatrix}$$

Obie macierze rzutowania mogą zostać uogólnione na wyższe wymiary i przyjmą postacie:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{w-d} & 0 & 0 & 0 \\ 0 & \frac{1}{w-d} & 0 & 0 \\ 0 & 0 & \frac{1}{w-d} & 0 \end{bmatrix}$$

b) Rotacja

Punkty w przestrzeni 3 wymiarowej mogą zostać obracane względem poszczególnych osi poprzez przemnożenie przez macierze:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

https://en.wikipedia.org/wiki/Rotation_matrix

Uogólnienie powyższych macierzy na 4 wymiar pozwoliłoby nam uzyskać między innymi macierz rotacji względem płaszczyzny XY:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos \theta & -\sin \theta \\ 0 & 0 & \sin \theta & \cos \theta \end{bmatrix}$$

Możliwe jest jednak wykonanie rotacji względem środka układu współrzędnych. Rotacja ta nosi miano rotacji podwójnej i opisuje ją macierz:

$$\begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & \cos \theta & -\sin \theta \\ 0 & 0 & \sin \theta & \cos \theta \end{bmatrix}$$

4. Algorytm

Zastosowanie powyższych przekształceń pozwoli na:

- 1) obrót tesseractu w przestrzeni 4 wymiarowej
- 2) rzutowanie tesseractu na przestrzeń 3 wymiarową
- 3) obrót w przestrzeni 3 wymiarowej
- 4) rzutowanie na przestrzeń 2 wymiarową
- 5) naniesienie punktów na ekran

5. Implementacja

W celu realizacji opisanego problemu, zaimplementowałem funkcje wykonujące:

- Mnożenie macierzy – odpowiedzialna za wykonanie rotacji i projekcji
- Skalowanie macierzy – odpowiedzialna głównie za przekształcenie macierzy rzutu prostokątnego w macierz rzutu perspektywicznego
- Zapełnienie macierzy zerami – przygotowanie pustych macierzy
- Zapełnienie macierzy jednostkowej – przygotowanie macierzy rotacji
- Zapełnienie macierzy rzutu prostokątnego
- Zapełnienie macierzy rotacji
- Zapełnienie macierzy podwójnej rotacji

6. Analiza obciążenia

Za pomocą narzędzia Performance Profiler środowiska Visual Studio porównałem obciążenie procesora poprzez poszczególne funkcje. Interesującym faktem jest to, że duża część obciążenia wynika z pracy GUI, a kolejnym istotnym obciążeniem są operacje korzystające z operatora new i delete - tworzenie i kopiowanie macierzy. Wynika to z faktu, że każde rzutowanie i rotacja tworzy nowy punkt, zamiast modyfikować istniejący. Obciążenie CPU wynikające z realizacji wyżej wymienionego algorytmu zaznaczyłem kolorem żółtym.

Function Name	Total CPU [ms, %]	Self CPU [ms, %]	Module
Projector4d.exe (PID: 11948)	39590 (100.00%)	39590 (100.00%)	Projector4d.exe
__scrt_common_main_seh	28117 (71.02%)	0 (0.00%)	Projector4d.exe
WinMain	11193 (28.27%)	29 (0.07%)	Projector4d.exe
operator new	3412 (8.62%)	96 (0.24%)	Projector4d.exe
Matrix::copy	3214 (8.12%)	527 (1.33%)	Projector4d.exe
Canvas::draw2dMesh	2468 (6.23%)	64 (0.16%)	Projector4d.exe
operator delete	2249 (5.68%)	138 (0.35%)	Projector4d.exe
std::_Uninitialized_copy<Vector2d *, Vector2d *, std::allocator<Vector2d> >	2090 (5.28%)	136 (0.34%)	Projector4d.exe
std::vector<Vector4d, std::allocator<Vector4d> >::vector<Vector4d, std::allocator<...	1259 (3.18%)	12 (0.03%)	Projector4d.exe
std::vector<Vector2d, std::allocator<Vector2d> >::_Tidy	1009 (2.55%)	72 (0.18%)	Projector4d.exe
IGraphicsEngine::rotateX	895 (2.26%)	19 (0.05%)	Projector4d.exe
IGraphicsEngine::rotate	893 (2.26%)	15 (0.04%)	Projector4d.exe
IGraphicsEngine::rotateY	873 (2.21%)	11 (0.03%)	Projector4d.exe
IGraphicsEngine::rotateW	841 (2.12%)	17 (0.04%)	Projector4d.exe
IGraphicsEngine::rotateZ	833 (2.10%)	21 (0.05%)	Projector4d.exe
Canvas::drawLine	816 (2.06%)	122 (0.31%)	Projector4d.exe
std::vector<Vector4d, std::allocator<Vector4d> >::_Emplace_reallocate<Vector4d>	789 (1.99%)	15 (0.04%)	Projector4d.exe

Wyróżnione powyżej operacje z zakresu interfejsu IGraphicsEngine przeprowadzają rozpakowywanie obiektów geometrycznych na poszczególne wektory punktów, wywołują funkcje z biblioteki .dll, i ponownie pakują otrzymane wektory w obiekty.

Poniżej zamieściłem fragment listy obciążenia z wyróżnionymi poszczególnymi funkcjami biblioteki .dll.

Function Name	Total CPU [ms, %]	Self CPU [ms, %]	Module
std::vector<Vector3d,std::allocator<Vector3d> >::_Assign_range<Vector3d *>	543 (1.37%)	7 (0.02%)	Projector4d.exe
multiplyMatrix	503 (1.27%)	405 (1.02%)	AsmImplementation.dll
Canvas::drawFps	448 (1.13%)	7 (0.02%)	Projector4d.exe
AsmGraphicsEngine::rotateW	422 (1.07%)	22 (0.06%)	Projector4d.exe
fillRotationMatrix	348 (0.88%)	137 (0.35%)	AsmImplementation.dll
Mesh3d::Mesh3d	326 (0.82%)	10 (0.03%)	Projector4d.exe
AsmImplementations::rotateW	318 (0.80%)	9 (0.02%)	AsmImplementation.dll
calculateMatrixIndex	286 (0.72%)	286 (0.72%)	AsmImplementation.dll
AsmGraphicsEngine::rotateY	281 (0.71%)	13 (0.03%)	Projector4d.exe
fillIdentityMatrix	260 (0.66%)	36 (0.09%)	AsmImplementation.dll
AsmGraphicsEngine::rotateZ	255 (0.64%)	7 (0.02%)	Projector4d.exe
AsmGraphicsEngine::rotateX	249 (0.63%)	11 (0.03%)	Projector4d.exe
std::vector<Vector4d,std::allocator<Vector4d> >::_Change_array	225 (0.57%)	12 (0.03%)	Projector4d.exe
AsmGraphicsEngine::project3dPerspective	225 (0.57%)	13 (0.03%)	Projector4d.exe
Mesh2d::Mesh2d	214 (0.54%)	0 (0.00%)	Projector4d.exe
fillZerosMatrix	209 (0.53%)	151 (0.38%)	AsmImplementation.dll
AsmImplementations::rotateY	202 (0.51%)	7 (0.02%)	AsmImplementation.dll
AsmGraphicsEngine::rotate	193 (0.49%)	15 (0.04%)	Projector4d.exe
_sprintf	186 (0.47%)	4 (0.01%)	Projector4d.exe
_vsprintf_l	184 (0.46%)	2 (0.01%)	Projector4d.exe
AsmImplementations::rotateZ	170 (0.43%)	5 (0.01%)	AsmImplementation.dll
AsmGraphicsEngine::project2dPerspective	167 (0.42%)	9 (0.02%)	Projector4d.exe
AsmImplementations::rotateX	162 (0.41%)	4 (0.01%)	AsmImplementation.dll
fillDoubleRotationMatrix	151 (0.38%)	67 (0.17%)	AsmImplementation.dll
AsmImplementations::project3dPerspective	144 (0.36%)	7 (0.02%)	AsmImplementation.dll
std::vector<std::pair<unsigned int,unsigned int>,std::allocator<std::pair<unsigned...	119 (0.30%)	41 (0.10%)	Projector4d.exe
AsmImplementations::rotate	112 (0.28%)	5 (0.01%)	AsmImplementation.dll
std::vector<std::pair<unsigned int,unsigned int>,std::allocator<std::pair<unsigned...	96 (0.24%)	12 (0.03%)	Projector4d.exe
scaleMatrix	82 (0.21%)	67 (0.17%)	AsmImplementation.dll
std::vector<Vector4d,std::allocator<Vector4d> >::_Buy	82 (0.21%)	4 (0.01%)	Projector4d.exe
_free	79 (0.20%)	79 (0.20%)	Projector4d.exe
AsmImplementations::project2dPerspective	75 (0.19%)	6 (0.02%)	AsmImplementation.dll
_vsprintf_s_l	73 (0.18%)	1 (0.00%)	Projector4d.exe
sprintf_s	73 (0.18%)	0 (0.00%)	Projector4d.exe

Jak widać, najwięcej czasu wykorzystuje funkcja mnożenia macierzy, gdyż jest najczęściej wywoływana - za równo przy rotacji, jak i projekcji.

7. Opis wybranej funkcji

Postanowiłem opisać działanie funkcji podwójnej rotacji, gdyż jest ona kluczowym elementem tego projektu.

Wykonanie rotacji składa się z dwóch etapów – wypełnienia macierzy rotacji i przemnożenia jej przez wektor punktu.

```
void AsmImplementations::rotateW(unsigned int cols, unsigned int rows, double* arr, double* outarr, double angle) {  
    fillDoubleRotationMatrix(rows, rows, rotateWMatrix, angle);  
    multiplyMatrix(rows, rows, rotateWMatrix, cols, rows, arr, outarr);  
}
```

Macierz wypełnia procedura **fillDoubleRotationMatrix**.

Jej wykonanie rozpoczyna się od wywołania procedury wypełniającej macierz jednostkową. Procedura ta zeruje macierz, a następnie iteruje po przekątnej macierzy i wpisuje 64 bitową zmiennoprzecinkową wartość 1.0.

Następnie obliczane są wartości cosinus i sinus kąta rotacji, i wpisywane pod odpowiednie adresy tablicy reprezentującej macierz.

```
fillDoubleRotationMatrix proc cols: DWORD, rows: DWORD, arr: DWORD, angle: REAL8  
    push arr  
    push rows  
    push cols  
    call fillIdentityMatrix ; prepare matrix  
  
    push cols  
    push 0  
    push 0  
    call calculateMatrixIndex  
    mul [DOUBLE_SIZE]  
    add eax, arr  
    fld angle  
    fcos  
    fstp REAL8 PTR [eax] ; write cos(angle)  
    movsd xmm0, REAL8 PTR [eax]  
  
    push cols  
    push 1  
    push 1  
    call calculateMatrixIndex  
    mul [DOUBLE_SIZE]  
    add eax, arr  
    movsd REAL8 PTR [eax], xmm0  
  
    push cols  
    push 2  
    push 2  
    call calculateMatrixIndex  
    mul [DOUBLE_SIZE]
```

Początkowy fragment procedury **fillDoubleRotationMatrix**.

```

fillIdentityMatrix proc cols: DWORD, rows: DWORD, arr: DWORD
    push ebx
    push arr
    push rows
    push cols
    call fillZerosMatrix           ; fill w/ zeros
    movsd xmm0, [ONE]
    mov ebx, 0
dataloop:
    cmp ebx, cols                 ; iterate over diagonal, assuming cols==rows
    je finished
    push cols
    push ebx
    push ebx
    call calculateMatrixIndex
    mul [DOUBLE_SIZE]
    add eax, arr                  ; get address
    movsd REAL8 PTR [eax], xmm0  ; write one
    add ebx, 1
    jmp dataloop
finished:
    pop ebx
    ret 12
fillIdentityMatrix endp

```

Procedura **fillIdentityMatrix**.

Przemnożenie macierzy wykonuje procedura **multiplyMatrix**.

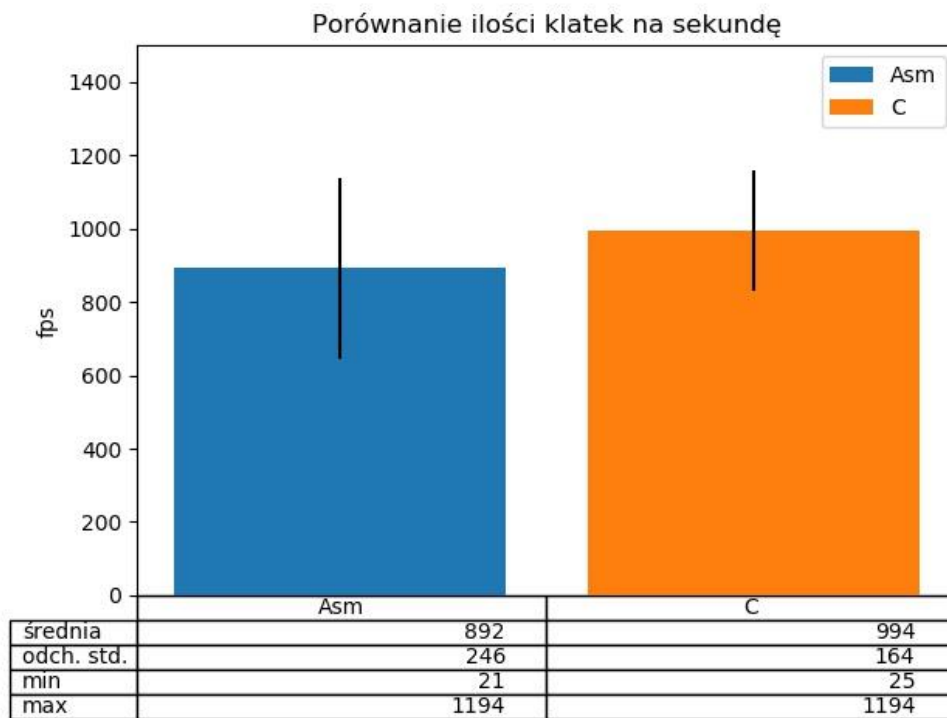
```
multiplyMatrix proc cols1: DWORD, rows1: DWORD, arr1: DWORD, cols2: DWORD, rows2: DWORD, arr2: DWORD, outarr: DWORD
    push ebx
    mov ebx, 0 ; initialize row index (iterates over rows of output mat)
rowloop:
    cmp ebx, rows1 ; new matrix has same rows as first mat
    je finished ; loop is ended
    mov ecx, 0 ; initialize column index (iterates over cols of output mat)
colloop:
    cmp ecx, cols2 ; new matrix has same cols as second mat
    je rowloopend ; move to next row
    mov edx, 0 ; init depth index
    xorpd xmm1, xmm1 ; zero mm1 register
depthloop:
    cmp edx, cols1 ; iterate over cols of first mat and rows of second
    je colloopend
    push edx ; hide edx on stack (mul in called func may overwrite)
    push cols1 ; read element from first matrix
    push ebx
    push edx
    call calculateMatrixIndex
    pop edx ; get edx back
    mul [DOUBLE_SIZE]
    add eax, arr1
    movsd xmm0, REAL8 PTR [eax] ; write it to mm0
    push edx
    push cols2 ; read element from second matrix
    push ecx
    call calculateMatrixIndex
    pop edx
    mul [DOUBLE_SIZE]
    add eax, arr2
    mulsd xmm0, REAL8 PTR [eax] ; multiply it to previous value
    addsd xmm1, xmm0 ; aggregate in mm1
    add edx, 1
    jmp depthloop
colloopend:
    push cols2
    push ebx
    push ecx
    call calculateMatrixIndex
    mul [DOUBLE_SIZE]
    add eax, outarr
    movsd REAL8 PTR [eax], xmm1 ; write aggregated sum to output matrix
    add ecx, 1 ; increment idx
    jmp colloop
rowloopend:
    add ebx, 1 ; increment idx
    jmp rowloop
finished:
    pop ebx
    ret
multiplyMatrix endp
```

Operacja mnożenia macierzy wymaga trzykrotnie zagnieżdżonej iteracji, mnożenia oraz sumowania wyniku. Operacje algebraiczne odbywają się przy pomocy rejestrów liczb zmiennoprzecinkowych xmm.

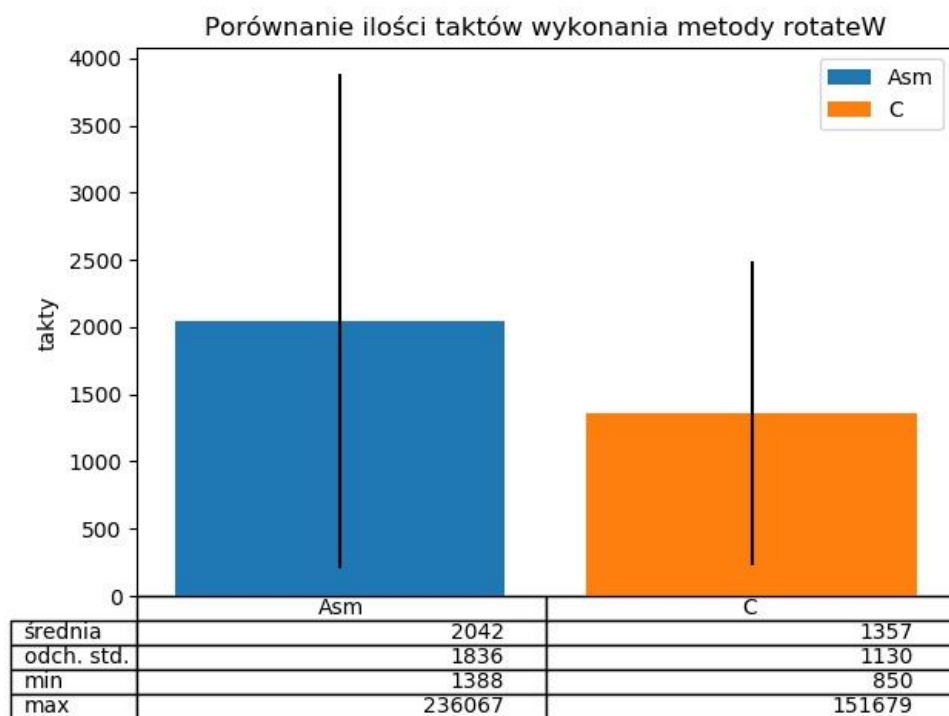
8. Porównanie wydajności implementacji w języku asemblera i C

Implementacja w języku asemblera została porównana z implementacją w języku C o identycznej strukturze i działaniu.

Testy zostały wykonane poprzez uruchomienie aplikacji na 60 sekund, pomiar ilości klatek na sekundę oraz ilość taktów potrzebnych na wykonanie funkcji **rotateW**. Wyniki zostały naniesione na wykresy.



Wykres 1. Ilość klatek na sekundę podczas generowania animacji obracającego się tesseractu



Wykres 2. Ilość taktów wykonania funkcji rotateW

Na podstawie powyższych danych jesteśmy w stanie stwierdzić, że implementacja w języku C generuje animację z ilością klatek większą o 11%, a ilość taktów potrzebnych na wykonanie funkcji rotateW jest mniejsza o 43%.

9. Wnioski

Realizacja projektu oraz porównanie wydajności implementacji w języku assemblera i C pozwala wysnuć wiele wniosków. Tworzenie oprogramowania w języku assemblera, pomimo walorów edukacyjnych, prowadzi do powstania kodu o większej objętości i często wymaga poświęcenia większej ilości czasu. W przypadku mojej aplikacji powyższe trudności nie przełożyły się na większą wydajność aplikacji, co mogło być spowodowane lepszą optymalizacją kodu w języku C przez kompilator. Inną interesującą konkluzją płynącą z analizy obciążenia CPU, jest fakt, że kod assemblera odpowiedzialny jest za około 5% wykorzystanego czasu procesora. Pokazuje to, że kolejnym krokiem w celu optymalizacji programu powinna być poprawa wydajności części kodu w języku wyższego poziomu. W przypadku mojego projektu, byłaby to poprawa zarządzania tworzenia i usuwania obiektów macierzy.

10. Bibliografia

- <http://www.mathaware.org/mam/00/master/essays/dimension/dimen11.html>
- https://en.wikipedia.org/wiki/Orthographic_projection
- <https://ef.gy/linear-algebra:perspective-projections>
- <http://mathworld.wolfram.com/RotationMatrix.html>
- <https://www.sfml-dev.org/index.php>