Politechnika Rzeszowska
Wydział Elektrotechniki i Informatyki
Katedra Informatyki i Automatyki

GRAFIKA KOMPUTEROWA

PROJEKT

Temat: Niemiecki czołg „Panther”

Wykonali:
Jerzy Pasławski
Robert Olech
II EF-DI, L03

Rzeszów 2019

1. Cel projektu

Realizowany projekt z przedmiotu Grafika Komputerowa polegał na stworzeniu modelu graficznego wybranego obiektu przy wykorzystaniu biblioteki graficznej OpenGL. Założono, że wybrany temat musi umożliwić zespołowi projektowemu wprowadzić podstawowe obroty w co najmniej dwóch punktach obiektu. Ponadto w celu opracowania modelu graficznego domyślnie należało korzystać jedynie z podstawowych obiektów graficznych nazywanych również prymitywami. Każdą część składającą się na model należało skonstruować osobno.

2. Wybór tematu i uzasadnienie

Zespół projektowy zdecydował się utworzyć model niemieckiego czołgu „Panther". Tematyka wozów bojowych jest bliska obydwóm członkom zespołu, ponadto w sieci można znaleźć dostateczną ilość informacji, aby taki model odtworzyć z dobrą dokładnością (mowa oczywiście o proporcjach maszyny). Program został napisany w języku C, oczywiście wykorzystując wspomnianą wcześniej bibliotekę.

3. Szczegółowy opis pracy nad projektem

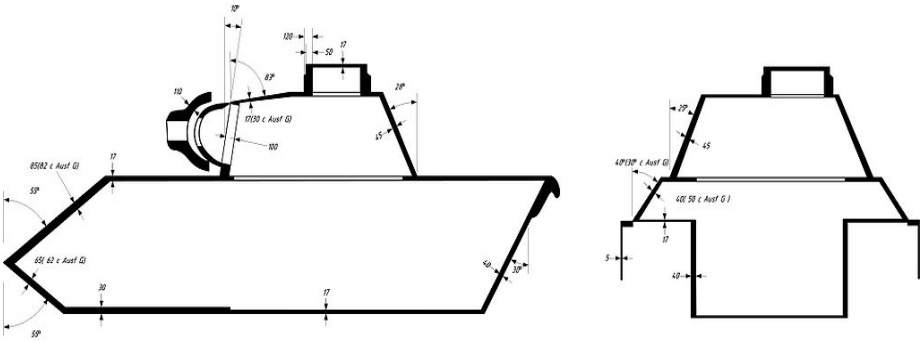
Zajęcia I – 04.03.2019

Aby zaznajomić się z tematyką poruszaną na przedmiocie, zrealizowano prosty model ruchomego ramienia. Należało połączyć podstawę ramienia z układem odniesienia oraz podobny układ umieścić w punkcie rotacji ramienia. Model znajdował się na płaszczyźnie (szachownicy). Takie ćwiczenie pozwoliło na zrozumienie w jaki sposób konstruować podstawowe obiekty oraz wykonywać rotację wokół wybranego punktu odniesienia.

Zajęcia II – 18.03.2019

Na kolejnych zajęciach, po konsultacji z prowadzącym wybrano temat projektu. Aby dokładniej określić wzór, na jakim będziemy bazować, wyszukano przykładowy wzorzec w sieci. Zespół zdecydował się na wykonanie graficznego modelu niemieckiego czołgu „Panther". Rozpoczęto pracę nad modelem, posiłkując się materiałami znalezionymi na Wikipedii dotyczących rozmiarów czołgu.

Rys.1: Zdjęcie oryginalnego modelu Panther'y z 1943r.



Rys.2: Projekt graficzny Panther'y, na podstawie którego zbudowano model.

Zajęcia III – 01.04.2019

Na podstawie wyżej umieszczonego rysunku określono wymiary czołgu oraz utworzono kadłub, wieżę oraz lufę wraz z kilkoma szczegółami. Dodano również podstawowe ruchy wieży oraz lufy. Niestety dokumentacja na temat niektórych szczegółów była niewystarczająca, należało więc dopasować wizualnie np. długość lufy. Na zajęcia następne zajęcia zaplanowano utworzyć układ gąsienicowy.

Zajęcia IV – 15.04.2019

Zgodnie z założeniami utworzono układ gąsienicowy. Dodano również ograniczenia ruchu lufy, ponieważ opracowywany czołg ma mniejszą zdolność depresji z tyłu (-6°) niż z przodu (-8°). Dodano również kilka szczegółów tj. blacha boczna oraz rury wydechowe. Na kolejne zajęcia zaplanowano wprowadzenie podstawowych ruchów całego modelu.

Zajęcia V – 13.05.2019

Zaplanowane ruchy czołgu udało się zrealizować jedynie częściowo, ze względu na zbyt małą wiedzę na temat aktualizacji osi obrotu obiektu. Dopracowano natomiast pozycjonowanie poszczególnych części modelu, tak aby cały model nie „rozjeżdżał" się po wykonaniu translacji bądź rotacji. Dopracowano również układ gąsienicowy, który stanowił największe wyzwanie, jeśli chodzi o odpowiednie ułożenie gąsienic na kołach. Dodano również śruby na kołach oraz umożliwiono rotację kół w trakcie ruchu modelu. Takie rozwiązanie umożliwi lepszą widoczność ich obrotu. Na kolejne zajęcia prowadzący zasugerował wprowadzenie teksturowania.

Zajęcia VI – 27.05.2019

Nałożono tekstury na niemal każdy obiekt składający się na czołg. Pominięto gąsienice, koła oraz lufę. Po konsultacji z prowadzącym okazało się, że wybrane tekstury nie są odpowiednie (użyto tekstury imitujące kamuflaż), ponieważ zlewają się i sprawiają, że cały model jest niewyraźny. Zasugerowano więc zmianę tekstur na bardziej przypominające naturalną powierzchnię czołgu, oraz wprowadzenia oświetlenia.

Zajęcia VII – 10.06.2019

Zmieniono tekstury zgodnie z zaleceniami prowadzącego i wprowadzono podstawowe oświetlenie, jednak wymaga ono dopracowania. Model rusza się prawidłowo, dodano również możliwość swobodnego poruszania kamerą.

4. Budowa obiektu głównego

Przedstawione w tej części fragmenty kodu przedstawiają proces tworzenia wybranych części czołgu. Nie ma potrzeby przedstawiania kodu wszystkich obiektów składających się na model ze względu na analogiczni sposób ich budowania. W następujący sposób utworzono jedną ze ścian wieży:

```
1. glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
2. glColor3d(1, 1, 1);
3. glEnable(GL_TEXTURE_2D);
4. glBindTexture(GL_TEXTURE_2D, texture[1]);
5. glBegin(GL_QUADS);
6. glTexCoord2d(1.0, 1.0); glVertex3d(-34, 56, 6);
7. glTexCoord2d(0.0, 1.0); glVertex3d(-28, -44, 44);
8. glTexCoord2d(0.0, 0.0); glVertex3d(-34, -56, 6);
9. glTexCoord2d(1.0, 0.0); glVertex3d(-28, 44, 44);
10. glEnd();
```

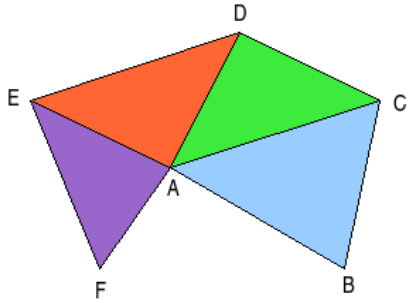
Na początek zadeklarowano wypełnianie obiektu zarówno od strony zewnętrznej jak i wewnętrznej. Następnie ustawiono kolor (w tym przypadku biały), aby określić odcień tekstury. W linii 3 włączono teksturowanie, a w kolejnej wybrano teksturę zadeklarowaną jako drugą w tablicy tekstur. Następnie określono prymityw, z którego powstanie obiekt. W kolejnych liniach za pomocą funkcji *glTexCoord2d* ustawiono pozycję tekstur na obiekcie, a korzystając z *glVertex3d* umieszczono wierzchołki obiektu w przestrzeni trójwymiarowej. Ostatecznie zakończono budowę obiektu funkcją *glEnd*.

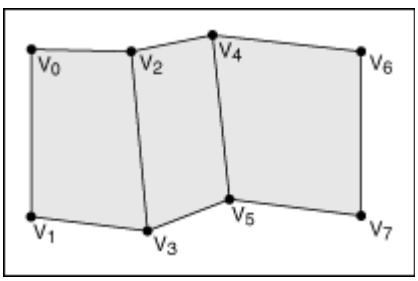
Każdy obiekt składający się z 4 wierzchołków zbudowano w sposób analogiczny, zmieniano jedynie używaną teksturę oraz pozycje wierzchołków w przestrzeni trójwymiarowej. Do budowy obiektów składających się z 3 wierzchołków użyto prymitywu o nazwie *GL_TRIANGLES*. Proces tworzenia takiej figury różnił się ilością ustawianych wierzchołków oraz pozycjonowanie tekstury.

Obiekty typu walec lub stożek tworzono w sposób następujący:

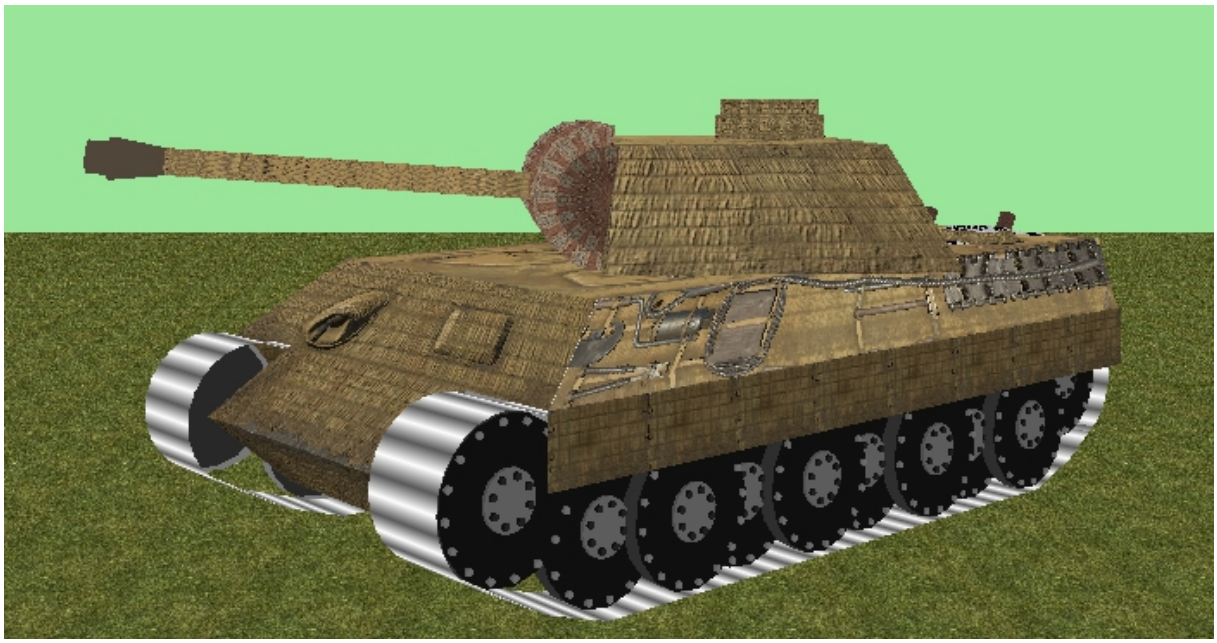
```
1.  glBegin(GL_TRIANGLE_FAN);
2.  glColor3d(0.1, 0.1, 0.1);
3.  glVertex3d(0, 0, 0);
4.  for (alpha = 0; alpha <= 8 * PI; alpha += PI / 8.0)
5.  {
6.          x1 = 21 * sin(alpha);
7.          z1 = 21 * cos(alpha);
8.          glVertex3d(x1, 0, z1);
9.  }
10. glEnd();
11.
12. glBegin(GL_QUAD_STRIP);
13. glColor3d(0.3, 0.3, 0.3);
14. for (alpha = 0.0; alpha <= 2 * PI; alpha += PI / 8.0)
15. {
16.         x1 = 21 * sin(alpha);
17.         z1 = 21 * cos(alpha);
18.         glVertex3d(x1, h, z1);
19.         glVertex3d(x1, 0, z1);
20. }
21. glEnd();
```

Struktura kodu różni się od poprzedniego przypadku zastosowanym prymitywem oraz pętlą. Prymitywy, o których mowa to odpowiednio: *GL_TRIANGLE_FAN* oraz *GL_QUAD_STRIP* (przykład ich konstrukcji przedstawiono na następnej stronie). Pozwalają one na szybkie tworzenie figur złożonych, korzystając równocześnie z typów prymitywnych. *TRIANGLE_FAN* użyto do zbudowania podstawy walca lub stożka, imitując dzięki niemu okrąg. Taki efekt osiągnięto łącząc dużą ilość trójkątów w pętli, tak jak widać w liniach 6-8.
Aby utworzyć powierzchnię boczną opisanych figur skorzystano z *QUAD_STRIP*, czyli kolejno połączonych ze sobą czworokątów. Również w tym przypadku połączono dużą ilość prostokątów, aby stworzyć iluzję powierzchni okrągłej.





Korzystając z wymienionych prymitywów udało się zbudować obiekt o następującym wyglądzie:



Tekstury oraz światło znacznie ulepszają ostateczny owoc pracy zespołu. Każdy element czołgu jest dobrze skontrastowany z pozostałymi. Fakt, że nie teksturowano gąsienic oraz kół jest widoczny, jednak kolory jakie przypisano tym obiektom są podobne do rzeczywistych.

5. Teksturowanie

Aby nałożyć teksturę tak jak opisano to w poprzedniej części, należy najpierw zaimportować teksturę do programu, Poniżej przedstawiono ten proces:

- utworzenie tekstury w formacie *.bpm*

W celu utworzenia tekstury korzystano zarówno z programu GIMP jak i Paint. Ten ostatni, pomimo złej opinii w środowisku informatycznym, okazał się skutecznym narzędziem do przekształcania plików znalezionych w sieci z formatu *.jpg* lub *.png* do wymaganego przez OpenGL.

- deklaracja tablic przechowującej tekstury w programie

Na początku programu należało min. zadeklarować nagłówek obrazu oraz utworzyć obiekt tekstury. W tym przypadku obiekt struktury jest tablicą, ze względu na większą ilość stosowanych tekstur.

```
// Opis tekstury
BITMAPINFOHEADER    bitmapInfoHeader;   // nagłówek obrazu
unsigned char*              bitmapData;               // dane tekstury
unsigned int        texture[20];              // obiekt tekstury
```

- funkcja importująca teksturę

Aby zaimportować teksturę skorzystano z wcześniej wcześniej przygotowanej funkcji o protokole następującym:

```
unsigned char *LoadBitmapFile(char *filename, BITMAPINFOHEADER *bitmapInfoHeader);
```

Jako argumenty przyjmuje ona nazwę pliku oraz nagłówek obrazu. W ciele funkcji odbywa się kolejno:

- otwarcie pliku w trybie „read binary”
- wczytanie nagłówka pliku
- sprawdzenie formatu pliku (musi to być plik w formacie *.bmp*)
- wczytanie nagłówka obrazu
- ustawienie wskaźnika pozycji pliku na początek danych obrazu
- przydzielenie pamięci buforowi obrazu
- sprawdzenie, czy udało się przydzielić pamięć
- wczytanie danych obrazu
- sprawdzenie, czy wczytanie się powiodło
- zamiana miejscami składowych R i B
- zamknięcie pliku oraz zwrócenie wskaźnika bufora zawierającego wczytany obraz

- załadowanie tekstury

W funkcji `LRESULT CALLBACK WndProc` należy załadować plik tekstury oraz przypisać go do kolejnego rejestru tablicy, którą utworzono w drugim kroku. Odbywa się to następująco:

```
// ładuje drugi obraz tekstury:
bitmapData = LoadBitmapFile("objects/wieza_bok_prawo.bmp",&bitmapInfoHeader);
glBindTexture(GL_TEXTURE_2D, texture[1]);       // aktywuje obiekt tekstury

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);

// tworzy obraz tekstury
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, bitmapInfoHeader.biWidth,
bitmapInfoHeader.biHeight, 0, GL_RGB, GL_UNSIGNED_BYTE, bitmapData);
if (bitmapData)
      free(bitmapData);
```

Po takim procesie możliwe będzie teksturowanie obiektów za pomocą funkcji *glBindTexture.*

6. Rotacje i translacje

Rotacje i translacje umożliwiły wprowadzenie sterowania w modelu i nie tylko. Rotacja, jak sama nazwa wskazuje, jest to obrót danego obiektu wokół wybranej osi, natomiast translacja jest to przesunięcie danego obiektu z punktu A do punktu B. Aby takie działania były możliwe w programie, stosowano się do następujących kroków:

- deklaracja zmiennych odpowiadających za ruch na początku programu

```
1. static GLfloat xRot = 270.0f;
2. static GLfloat xMove = -100.0f;
3. static GLfloat yMove = 0.0f;
4. static GLfloat yRot = 270.0f;
```

Zmienne przedstawione na powyższym wycinku kodu odpowiadają za sterowanie kamerą. *xRot* jest to zmienna odpowiadająca za kąt, o który będzie się obracała kamera względem osi X; podobnie jest dla zmiennej *yRot*. Pozostałe zmienne określają położenie kamery w przestrzeni w tym przypadku dwuwymiarowej. Zrezygnowano z trzeciego wymiaru, ponieważ stwierdzono, że jest to zbędne.

- określenie osi obrotu oraz wymiarów translacji

W funkcji tworzącej scenę należy określić względem jakich osi program ma obracać/przesuwać obiekt. Przykładowy kod:

```
1. glPushMatrix();
2. glRotatef(xRot, 1.0f, 0.0f, 0.0f);
3. glRotatef(yRot, 0.0f, 0.0f, 1.0f);
4. glPushMatrix();
5. glTranslatef(xMove, yMove, -25.0f);
```

Funkcja *glRotatef* jako pierwszy argument przyjmuje zmienną, która odpowiada za kąt obrotu, natomiast kolejne 3 argumenty to odpowiednio osie X, Y oraz Z. Programista ustawia na 1 oś wokół której ma się obiekt obracać.
Funkcja *glTranslatef* przyjmuje argumenty dotyczące tego, ile ma się przesunąć później rysowany obiekt względem punktu odniesienia.

- ustawienie przycisku pod którym będzie zmieniała się wartość zmiennej

W funkcji `LRESULT CALLBACK WndProc` programista ustawia przycisk na klawiaturze bądź innym urządzeniu, na który będzie reagowała wcześniej zadeklarowana zmienna. Oto przykład:

```
1. if (wParam == VK_UP)
2. {
3.         if (yRot > 0.0f)
4.         {
5.                 xMove -= 10.0f * sinus[(int)yRot / 10];
6.                 yMove -= 10.0f * cosinus[(int)yRot / 10];
7.         }
8.         else if (yRot < 0.0f)
9.         {
10.                xMove += 10.0f * (-sinus[(int)yRot / 10 + 36]);
11.                yMove += 10.0f * (-cosinus[(int)yRot / 10 + 36]);
12.        }
13. }
```

W tym przypadku, po naciśnięciu przycisku „strzałka w górę” kamera przemieści się w stronę w jaką jest skierowana. Z racji tego, że kąt pod jakim może się znaleźć kamera względem osi może być ujemny, ustawiono odpowiedni warunek przewidujący alternatywne zachowanie kamery w takim przypadku. Jak można zauważyć na powyższym fragmencie kodu, pojawiają się dwie tablice: *sinus* oraz *cosinus*. W trakcie opracowywania programu okazało się, że funkcje systemowe źle obliczają podane wartości, zdecydowano więc utworzyć tablicę z wartościami przyjmowanymi przez każdą z funkcji z krokiem co 10.
Zmienne odpowiadające za rotacje powinny domyślnie nie przekraczać wartości 360. Z tego powodu na koniec funkcji umieszczono następujące polecenie:

```
1. xRot = (const int)xRot % 360;
2. yRot = (const int)yRot % 360;
```

Stosując dzielenie zresztą, zmienne będą miały zawsze wartości z przedziału obustronnie otwartego (-360, 360).
Proces wprowadzania translacji oraz rotacji w programie był wyzwaniem, ponieważ zespół projektowy po raz pierwszy pracuje w takiej technologii. Często pojawiały się problemy przy zapisaniu kolejności ruchów oraz umieszczeniem ich w odpowiedni miejscu w funkcji rysującej.
Poniżej przedstawiono listę wszystkich obiektów, które poruszają się w programie:

- translacja oraz rotacja całego czołgu (sterowanie pod klawiszami W, A, X, D)
- rotacja kół
- rotacja wieży wraz z lufą (klawisze F1 oraz F2)
- depresja oraz podnoszenie lufy (klawisze F3 oraz F4)
- translacja oraz rotacja kamery (sterowanie strzałkami)

7. Kamera - podejście teoretyczne

OpenGL oczekuje, że wszystkie widoczne wierzchołki, będą w znormalizowanych współrzędnych urządzenia (normalized device coordinates - NDC) po każdym wywołaniu vertex shader'a. Oznacza to, że współrzędne *x*, *y* i *z* każdego wierzchołka powinny mieścić się w przedziale od -1,0 do 1,0. Współrzędne poza tym zakresem nie będą widoczne. Zazwyczaj określamy współrzędne w zakresie, który sami skonfigurujemy, a w shaderze przekształcamy te współrzędne do NDC. Następnie są one przekazywane do rasteryzatora w celu przekształcenia ich na współrzędne 2D lub piksele na ekranie.

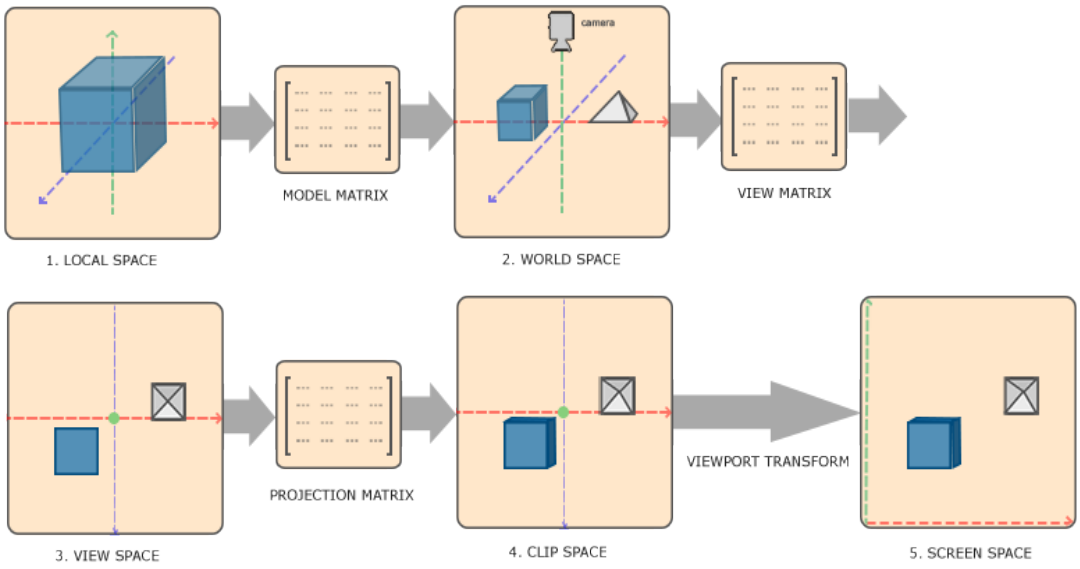
Przekształcanie współrzędnych do NDC, a następnie do współrzędnych ekranu jest zwykle wykonywane krok po kroku, gdzie przekształca się wierzchołki obiektu do kilku układów współrzędnych, przed ostatecznym przekształceniem ich do współrzędnych ekranu. Zaletą przekształcenia ich do kilku pośrednich układów współrzędnych jest to, że niektóre operacje / obliczenia są łatwiejsze w niektórych z nich. Wyróżniamy 5 różnych układów:

- przestrzeń lokalna
- przestrzeń świata
- przestrzeń widoku / kamery
- przestrzeń obcinania (ang. Clip Space)
- przestrzeń ekranu

Ogólny obraz

Aby przekształcić współrzędne z jednej przestrzeni do następnej, używa się kilku macierzy transformacji, z których najważniejsze są macierze modelu, widoku i projekcji. Współrzędne wierzchołków znajdują się najpierw w przestrzeni lokalnej jako współrzędne lokalne i następnie są przetwarzane do współrzędnych globalnych, współrzędnych widoku, współrzędnych obcinania i ostatecznie kończą jako współrzędne ekranu. Poniższy obraz przedstawia cały proces i pokazuje, co każda transformacja robi:

1. Lokalne współrzędne są współrzędnymi obiektu względem jego lokalnego punktu początkowego.



2. Następnym krokiem jest przekształcenie lokalnych współrzędnych do współrzędnych przestrzeni świata, które są współrzędnymi względem większego świata. Te współrzędne odnoszą się do globalnego punktu początkowego świata, łącznie z wieloma innymi obiektami, które są również umieszczone w stosunku do tego punktu.
3. Następnie przekształca się współrzędne świata do przestrzeni widoku, w taki sposób, aby każda współrzędna była widoczna z punktu widzenia kamery/obserwatora.
4. Po tym jak współrzędne znajdą się w przestrzeni widoku, chcemy rzutować je do przestrzeni obcinania. Współrzędne obcinania są przetwarzane do zakresu -1.0 i 1.0 i określają, które wierzchołki będą pojawią się na ekranie.
5. Na koniec przekształcamy współrzędne obcinania na współrzędne ekranu w procesie, który nazywamy transformacją obszaru renderowania (ang. *viewport transform*), która zmienia współrzędne z zakresu -1.0 i 1.0 do współrzędnych z zakresu zdefiniowanego przez *glViewport*. Otrzymane współrzędne są następnie wysyłane do rasteryzera, aby przekształcić je we fragmenty.

Powodem, dla którego przekształcamy wierzchołki do różnych przestrzenie, jest to, że niektóre operacje mają sens lub są łatwiejsze przy użyciu niektórych układów współrzędnych. Na przykład modyfikowanie obiektu najlepiej jest zrobić w przestrzeni lokalnej, podczas gdy obliczanie pewnych operacji na obiekcie w odniesieniu do pozycji innych obiektów ma największy sens we współrzędnych świata.

Przestrzeń Lokalna

Przestrzeń lokalna to układ współrzędnych, który jest lokalny dla danego obiektu, tzn. układ, w którym obiekt ma swój początek. Kiedy tworzymy obiekt, jego środek prawdopodobnie jest ustawiony na punkt (0,0,0). Zatem wszystkie wierzchołki tego obiektu modelu znajdują się w przestrzeni *lokalnej*: wszystkie są lokalne dla tego obiektu.

Przestrzeń Świata

Po przypuszczalnym zaimportowaniu kilku obiektów bezpośrednio do aplikacji, prawdopodobnie wszystkie znalazłyby się gdzieś ułożone jeden na drugim wokół środka całego wirtualnego świata (0,0,0). Chcemy z, aby umieścić je w większym świecie. Współrzędne przestrzeni świata definiują pozycję dla każdego obiektu z osobna: współrzędne wszystkich wierzchołków są umieszczane względem globalnego punktu początkowego świata. Obiekty przekształcane są tak, aby każdy został rozmieszczony w jak najbardziej realistyczny sposób. Współrzędne obiektu są przekształcane z przestrzeni lokalnej do przestrzeni świata za pomocą macierzy modelu. Jest to macierz transformacji, która przesuwa, skaluje i/lub obraca obiekt, aby umieścić go w świecie w określonej lokalizacji/orientacji.

Przestrzeń Widoku

Przestrzeń widoku jest tak zwaną kamerą OpenGL (czasami też znaną jako przestrzeń kamery lub przestrzeń oka). Przestrzeń widoku jest wynikiem przekształcania współrzędnych przestrzeni świata na współrzędne, które znajdują się przed widokiem użytkownika. Przestrzeń widoku jest zatem przestrzenią, dzięki której obserwujemy scenę z punktu widzenia kamery. Zwykle odbywa się to za pomocą kombinacji translacji i rotacji, aby przesunąć/obrócić scenę tak, aby niektóre elementy znajdowały się na przeciw kamery. Te połączone transformacje są zazwyczaj przechowywane wewnątrz macierzy widoku, która przekształca współrzędne świata do przestrzeni widoku.

Przestrzeń Obcinania

Na końcu każdego wywołania vertex shader’a, OpenGL oczekuje, że współrzędne znajdą się w określonym przedziale i każda współrzędna poza tym zakresem zostanie

obcięta. Współrzędne, które są obcięte, są odrzucane, a pozostałe współrzędne kończą jako fragmenty widoczne na ekranie. Od tego zabiegu pochodzi nazwa przestrzeń obcinania.

Ponieważ określenie wszystkich widocznych współrzędnych znajdujących się w zakresie -1.0 i 1.0 nie jest zbyt intuicyjne, określa się własny zestaw współrzędnych do pracy i przekształca się je do NDC, tak jak OpenGL tego oczekuje.

W celu przekształcenia współrzędnych wierzchołków z przestrzeni widoku do przestrzeni obcinania, definiujemy tzw. macierz projekcji, która określa zakres współrzędnych np. -10000 i 10000 w każdym wymiarze. Macierz projekcji przekształca współrzędne w tym określonym zakresie do NDC (- 1.0, 1.0). Wszystkie współrzędne poza tym zakresem nie będą mapowane do przedziału -1.0 i 1.0, a zatem zostaną obcięte. We wspomnianym zakresie macierzy projekcji, współrzędna (13500, 4000, 5500) nie byłaby widoczna, ponieważ współrzędna *x* jest poza zakresem, a zatem przekształca się w współrzędną większą niż 1.0 w NDC i zostaje obcięta.

Przestrzeń utworzona przez macierzy projekcji nazywa się frustum i każda współrzędna, która znajduje się wewnątrz, pokaże się na ekranie użytkownika. Całkowity proces przekształcania współrzędnych z określonego zakresu do NDC, który może być łatwo odwzorowywany na współrzędne 2D przestrzeni widoku, nazywa się projekcją, ponieważ macierze projekcji rzutują współrzędne 3D na współrzędne 2D NDC.

Kiedy wszystkie wierzchołki zostaną przekształcone do przestrzeni obcinania, wykonana zostaje ostatnia operacja zwana dzieleniem perspektywicznym, gdzie dzielone są komponenty *x*, *y* i *z* wektorów pozycji przez składnik *w* tego wektora. Dzielenie perspektywiczne jest tym, co przekształca współrzędne przestrzeni obcinania 4D do współrzędnych 3D NDC. Ten krok jest wykonywany automatycznie po zakończeniu każdego wywołania shadera.

Po tym etapie uzyskane współrzędne są odwzorowywane na współrzędne ekranu (używając ustawień funkcji glViewport) i zostają zamienione na fragmenty.

Macierz projekcji przekształcająca współrzędne widoku do współrzędnych obcinania może przybrać dwie różne formy, przy czym każda forma określa własne niepowtarzalne frustrum. Możemy utworzyć macierz projekcji prostokątnej lub macierz projekcji perspektywicznej. W tym projekcie wykorzystano macierz projekcji perspektywicznej.

Projekcja perspektywiczna

Projekcja perspektywiczna stara się naśladować efekt perspektywy i wykorzystuje do tego macierz projekcji perspektywicznej. Macierz projekcji mapuje dany zakres frustum do przestrzeni obcinania, ale również manipuluje wartością '*w*' każdej

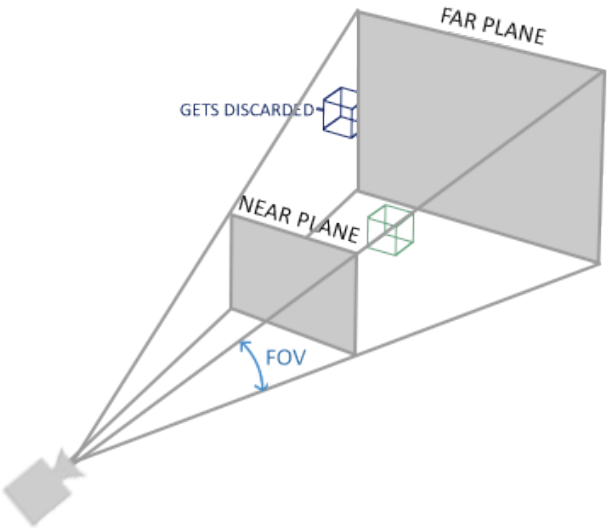
współrzędnej wierzchołka w taki sposób, że im dalej od obserwatora znajduje się wierzchołek, tym większy staje się ten składnik. Kiedy współrzędne zostaną przekształcone do przestrzeni obcinania to znajdują się one w zakresie -*w* i *w* (każdy wierzchołek poza tym zakresem zostanie obcięty). OpenGL wymaga, aby widoczne współrzędne znajdowały się w zakresie -1.0 i 1.0 jako wyjściowy wierzchołek vertex shader'a, więc gdy współrzędne znajdują się w przestrzeni obcinania, zostaje zastosowane dzielenie perspektywiczne:

$$out = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix}$$

Każdy składnik wierzchołka podzielony jest przez jego składnik *w*, dający tym mniejsze współrzędne wierzchołka, im dalej znajduje się wierzchołek od obserwatora. Otrzymane współrzędne znajdują się następnie w NDC. W GLUT'cie można utworzyć macierz projekcji perspektywicznej w następujący sposób:

```
gluPerspective(60.0f,fAspect,1.0,2000.0f);
```

Funkcja *gluPerspective* tworzy duże frustum, które definiuje widoczną przestrzeń, a cokolwiek znajdzie się poza tym frustum zostanie obcięte (nie będzie widoczne na ekranie użytkownika). Frustum perspektywiczne można zwizualizować jako niejednorodnie ukształtowane pudełko, gdzie każda współrzędna wewnątrz tego pola zostanie później dopasowana do przestrzeni obcinania. Wygląd frustum perspektywicznego przedstawiono poniżej:



Pierwszy parametr definiuje wartość *fov*, która oznacza pole widzenia (ang. *field of view*) i określa jak duży jest obszar widoku. Dla realistycznego widoku, ta wartość,

jest zazwyczaj ustawiana na 45.0f stopni. Drugi parametr określa współczynnik *aspect ratio*, który jest obliczany przez podzielenie szerokości przez wysokość ekranu. Trzeci i czwarty parametr ustawia bliską i daleką płaszczyznę frustum. Zwykle ustawia się odległość do bliskiej płaszczyzny w pobliżu wartości 0.1f i odległość od dalszej płaszczyzny w okolicy 100.0f. Wszystkie wierzchołki między płaszczyzną *bliską* a *daleką* są wewnątrz frustum i zostaną wyrenderowane.

Kiedy używamy projekcji prostokątnej, każda z współrzędnych wierzchołka jest bezpośrednio odwzorowywana w przestrzeni obcinania bez jakiegokolwiek eleganckiego dzielenia perspektywicznego (nadal następuje dzielenie perspektywiczne, ale wartość *w* nie jest zmieniana (pozostaje równa 1), a zatem nie ma żadnego efektu). Ponieważ projekcja prostokątna nie wykorzystuje rzutów perspektywicznych, obiekty znajdujące się dalej nie wydają się mniejsze, co powoduje dziwny odbiór wizualny. Z tego powodu projekcja prostokątna jest wykorzystywana głównie do renderowania 2D oraz dla niektórych zastosowań architektonicznych lub inżynierskich, w których nie chcemy zniekształcać wierzchołków.

Podsumowanie

Tworzona jest macierz transformacji dla każdego z wyżej wymienionych etapów: macierz modelu, widoku i projekcji. Następnie współrzędna wierzchołka zostaje przekształcona do współrzędnych obcinania w następujący sposób:

$$V_{clip} = M_{projection} \cdot M_{view} \cdot M_{model} \cdot M_{local}$$

Należy nadmienić, że kolejność mnożenia macierzy jest odwrotna (odczytuje się od prawej do lewej). Otrzymany wierzchołek powinien być następnie przypisany do zmiennej wbudowanej gl_Position w vertex shaderze, a następnie OpenGL automatycznie wykona dzielenie perspektywiczne i obcinanie.

Wyjście vertex shader'a wymaga, aby współrzędne znajdowały się w przestrzeni obcinania, co jest spełnione za pomocą macierzy transformacji. Następnie OpenGL wykonuje dzielenie perspektywiczne na współrzędnych w przestrzeni obcinania, aby przekształcić je w znormalizowane współrzędne urządzenia (NDC). OpenGL używa parametrów z funkcji glViewport, aby odwzorować znormalizowane współrzędne urządzenia do współrzędnych w przestrzeni ekranu, gdzie każda współrzędna odpowiada punktowi na ekranie. Proces ten nazywa się transformacją obszaru renderowania.

7. AntTweakBar

W kodzie zaimplementowane dosyć znane narzędzie w projektach graficznych o nazwie AntTweakBar. Prezentuje się ono następująco:

W programie wykorzystano to narzędzie jedynie do obserwacji zmian poszczególnych parametrów sterowania zarówno kamerą jak i czołgiem. Ułatwiło to w dużej mierze poprawę występujących błędów jak np. jazda w złym kierunku po przyciśnięciu danego klawisza lub obrót względem złej osi. Pozwoliło to również zauważyć wcześniej opisany błąd, czyli niepoprawne obliczanie wartości sinusa lub cosinusa. Aby utworzyć AntTweakBar' a należało dodać bibliotekę *AntTweakBar.h* oraz *MiniGLUT.h* . Poniżej przedstawiono sposób tworzenia oraz dodawania zmiennych do tego narzędzia:

```
1. TwInit(TW_OPENGL, NULL);
2. TwBar *bar;
3. bar = TwNewBar("Sterowanie");
4. TwAddVarRW(bar, "zoom", TW_TYPE_FLOAT, &zoom,"keyIncr=F9 keyDecr=F8
   help='Przyblizanie oraz oddalanie sceny'");
```

W pierwszej linii zainicjowano narzędzie (skrót *Tw* odnosi się właśnie do nazwy AntTweakBar), a następnie utworzono obiekt *bar*. W linii 3 nadano nazwę „paskowi”. W ostatniej linii przedstawiono przykładowy sposób wprowadzenia wiersza do „paska”. Jak widać należy na początku zadeklarować nazwę obiektu, do którego zostanie wstawiony wiersz (może być więcej niż jeden AntTweakBar w programie), następnie nazwę wyświetlaną na pasku, typ zmiennej, wskaźnik do zmiennej oraz instrukcje obsługi.
Oczywiście narzędzie to może być znacznie rozbudowane, jednak cel projektu był inny, a skorzystanie z takiego rozszerzenia jedynie ciekawostką, wzbogaceniem interfejsu.

8. Struktura programu

W tej części zostanie w skrócie opisana struktura programu:

- deklaracja dyrektyw preprocesora oraz bibliotek
- deklaracja zmiennych globalnych (w tej części umieszczono min. zmienne odpowiadające za rotacje oraz ruch obiektów)
- deklaracja procedury okna
- deklaracja procedury okna „dialogowego" *AboutBox*
- funkcja ustawiająca *Pixel Format*
- funkcja *ReduceToUnit* sprowadzająca wektor składający się z 3 punktów do wektora o długości 1
- funkcja *calcNormal* zwracająca normalną z dwóch wektorów o długości 3
- funkcja *ChangeSize* dostosowująca scenę do zmiany wymiarów okna
- funkcja *SetupRC* w której następuje zainicjowanie wszystkich obiektów dotyczących renderowaną scenę np.: światło, kolor tła, używane narzędzia (AntTweakBar) itp.
- funkcja *LoadBitmapFile* zwracająca wskaźnik do wczytanej tekstury
- funkcje odpowiadające za określenie położenia oraz tekstur składowych czołgu oraz otoczenia. W skrócie, w tej części tworzone są wszystkie obiekty.
- funkcja *RenderScene*, w której programista umieszcza wszystkie obiekty, które powinny się pojawić w oknie. Mowa tu o narysowaniu wszystkich obiektów (zarówno składowych modelu jak i AntTweakBar). Ważne jest aby odpowiednio ustawić rotację oraz ruch obiektów
- funkcja *SetDCPixelFormat* odpowiadająca za wybranie odpowiedniego format pikselowego dla danego środowiska urządzenia.
- funkcja *GetOpenGLPalette* w razie potrzeby tworzy paletę kolorów o wielkości 3x3x2 dla środowiska danego urządzenia
- funkcja *WndProc* obsługująca wszystkie procedury oraz zdarzenia zachodzących w trakcie działania programu. W tej części programista deklaruje nazwy tekstur do załadowania, przyciski na które ma reagować dana zmienna itp.
- procedura okna „dialogowego"

9. Podsumowanie

Zgodnie ze wstępnymi założeniami udało się zrealizować model graficzny niemieckiego czołgu „Panther”. Czołg jest w stanie wykonywać większość ruchów możliwych w rzeczywistości. Udało się również stworzyć iluzję poruszającej się gąsienicy zmieniając kolor jej fragmentów. Program umożliwia również poruszanie kamerą w wybranym przez użytkownika kierunku. Taki zabieg umożliwia obejrzenie modelu z wielu perspektyw.
Warto zaznaczyć, że model był rozbudowywany konsekwentnie z jednych zajęć na następne, o czym świadczy dokładny opis przebiegu pracy zawarty w części III dokumentacji.
Możliwa jest rozbudowa projektu, dodając:

- bardziej płynne animacje
- lepsze tekstury
- zmiana podłoża (tworząc je np. w programie Blender)
- wprowadzenie zdarzeń (np. wybuchy, strzał, kolizje)
- możliwość obejrzenia środka maszyny
- wprowadzenie fabuły

Repozytorium z kodem programu oraz niezbędnymi plikami potrzebnymi do jego funkcjonowania można odnaleźć pod adresem:

https://github.com/jpaslawski/Panther-OpenGL