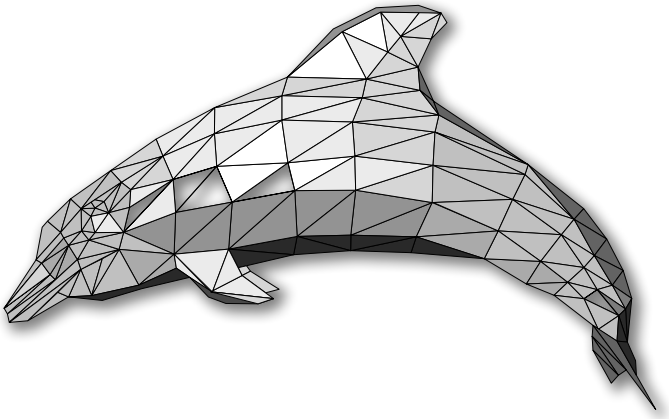
Trójwymiarowe animacje na LCD (2)

Tworzenie animacji 3D na wyświetlaczu LCD za pomocą STM32



Tekst jest kontynuacją artykułu „Animacje na LCD" opublikowanego w Elektronice Praktycznej nr 1/2012. Projekt przykładowy został zrealizowany z użyciem mikrokontrolera STM32F103VCT6 z rdzeniem Cortex-M3 taktowanego sygnałem o częstotliwości 72 MHz. Po odpowiednie schematy i opis interfejsu należy sięgnąć do wspomnianego artykułu (jest na CD).

Animacje trójwymiarowe są w rzeczywistości animacjami dwuwymiarowymi, ponieważ w ostateczności wyświetlane są na płaskich ekranach monitorów bądź innych wyświetlaczy. Złudzenie przestrzeni trójwymiarowej jest uzyskiwane poprzez zastosowanie pewnych trików i innych mechanizmów operujących na pikselach obrazu, dających w końcowej fazie złudzenie obrazu przestrzennego i głębi. Za prawdziwe animacje trójwymiarowe można by uznać np. projekcje holograficzne, które prezentują obiekty w rzeczywistej przestrzeni, mające szerokość, wysokość i głębokość. Tej ostatniej bez wątpienia brakuje obiektom prezentowanym na ekranie płaskim.

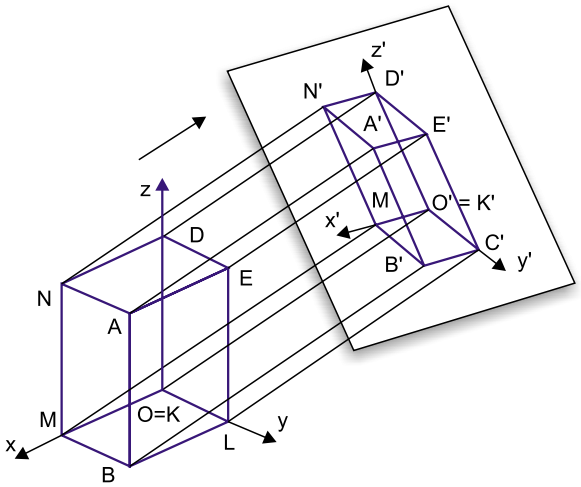
Reprezentacja

Animacje trójwymiarowe w rozumieniu klasycznym, tzn. animacji wyświetlanych na płaskich wyświetlaczach, ale posiadających złudzenie głębi, różni od animacji 2D proces ich generowania. Ponieważ formalinie niemożliwe jest ich odróżnienie, tj. po wymiarze, gdyż obie są dwuwymiarowe, to jednak w większości przypadkach odbiorca prawie natychmiast jest w stanie spostrzec różnice i odpowiednio wskazać tak zwane animacje 2D oraz 3D. Co więcej, dobry animator 2D jest w stanie uzyskać zadowalające efekty głębi, tak, że niektóre „oko" nie byłoby wstanie poprawnie rozpoznać, czy dana animacja jest tak zwaną 3D czy 2D. Mimo to, po wskazaniu techniki generowania owej animacji ta sama osoba bez wahania wskazałaby na 2D. Jak widzimy, rezultaty nie są dobrym kryterium identyfikującym. Sam proces tworzenia można jednak z większą dozą pewności uznać za bardziej rozróżnialny. Zazwyczaj w animacjach 2D rozpatrywany jest pewien obiekt posiadający tylko dwa wymiary tj. wysokość i szerokość, i wykonywane są na nim pewne operacje, przekształcenia, deformacje, przesunięcia itd. które następnie są prezentowane na wyświetlaczu również 2D. Natomiast w animacjach 3D obiekty posiadają trzy cechy, mianowicie wysokość, szerokość, i głębokość, które są poprzez różnie przekształcenia odpowiednio modyfikowane i podobnie jak w animacjach 2D prezentowane na ekranie. Tutaj jednak wykonywana jest dodatkowa operacja konwertująca trzy cechy do dwóch. Inaczej nie jest możliwością wyświetlenie ich na płaskim ekranie. Wyróżniamy zatem dwa podstawowe etapy tworzenia animacji 3D: modelowanie w przestrzeni trójwymiarowej oraz konwersja do odpowiednika płaskiego.

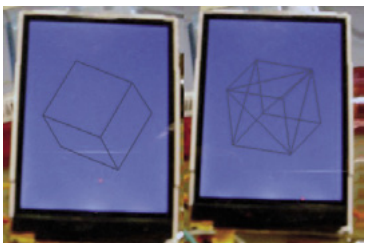
Dodatkowe materiały na CD/FTP:
ftp://ep.com.pl, user: 18453, pass: 5eyp1854
• pierwsza część kursu

Modelowanie

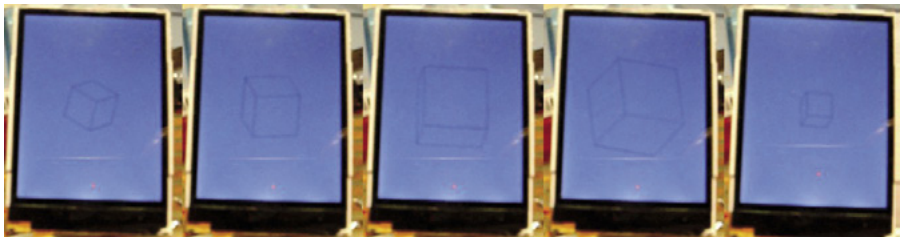
Punkty w przestrzeni trójwymiarowej są reprezentowane trzema liczbami (x,y,z), gdzie każda współrzędna x, y, z reprezentuje szerokość, wysokość i głębokość odpowiednio. Każde trzy różne niewspółliniowe punkty w przestrzeni wyznaczają niezdegenerowany trójkąt. Mając do dyspozycji wiele



Rysunek 1. Rzut równoległy na płaszczyznę



Rysunek 2. Bryła z widocznymi tylnymi ścianami (prawy) i niewidocznymi (lewy)



Rysunek 3. Sekwencja klatek animacji sześcianu

takich trójkątów można utworzyć praktycznie dowolną figurę w przestrzeni trójwymiarowej (bryłę). W zależności od ich ilości oraz wielkości figura będzie mniej lub bardziej przypominać nasz docelowy obiekt. Oczywiście zwiększanie ilości trójkątów wiąże się z zwiększeniem ilości obliczeń, co za tym idzie posiadaniem większej mocy obliczeniowej. Bryłę można zatem utożsamiać z skończonym zbiorem odpowiednio dobranych trójkątów (patrz rysunek z delfinem). Wybór trójkątów jako podstawowych elementów budulcowych nie jest koniecznością. Równie dobrze można wykorzystać inne figury, takie jak kwadraty lub wielokąty. Niestety każda z nich ma swoje ograniczenia jak i zalety. Dobór oczywiście jest zdeterminowany przeznaczeniem i narzuconymi wymaganiami.

Podstawowe transformacje

Dany punkt w przestrzeni trójwymiarowej można przesunąć w dowolne inne miejsce w tej przestrzeni poprzez dodanie/odjęcie do ich współrzędnych odpowiednich wartości. Zastosowawszy tą operację do wszystkich punktów każdego trójkąta otrzymamy przesunięcie całej bryły. Operację tą nazywamy translacją o wektor. Za każdym razem, gdy bryła jest przemieszczana jej „przednia" część jest zawsze z „przodu" i nie ma żadnej możliwości obejrzenia jej tylnej, bocznej czy górnej części. Efekt taki można uzyskać za pomocą operacji obrotu wokół osi. Ponieważ działamy w przestrzeni trójwymiarowej wyróżniamy trzy podstawowe obroty wokół osi współrzędnych Ox,Oy,Oz. Pozostaje jeszcze jedna operacja: skalowanie – umożliwiająca powiększanie lub zmniejszanie obiektów zachowując proporcje. Powyższe operacje nie należy wyobrażać sobie, jako animacje obrotu czy przemieszczania bryły z jednego miejsca do drugiego. Są to operacje jednostkowe, powodujące przemieszczenie bryły „za jednym zamachem", „natychmiast" w inne miejsce.-

Powyższe trzy podstawowe transformacje wykonane w odpowiedniej kolejności i odpowiednią ilość razy umożliwiają wykonanie praktycznie dowolnej czynności, która będzie nam potrzebna do skonstruowana prostej animacji. Istnieją naturalnie przekształcenia, których nie da się uzyskać za pomocą tych trzech podstawowych operacji. Są to np. różnego rodzaju deformacje albo efekty wyrastania nowych części czy chociażby efekty rozłupywania brył na mniejsze. Te oraz inne efekty wykonywane są za pomocą innych bardziej złożonych technik.

Jak wprawić bryłę w ruch? Jak pamiętamy możemy wykonać pewne operacje na bryle w sposób „skokowy". Zmniejszając odległości tych „skoków" jesteśmy w stanie stworzyć złudzenie płynności. I właściwie to jest na tyle. Animacja jest gotowa! Odpowiedni dobór odległości oraz zarządzanie czasem wyświetlania daje efekt gładkości działania oraz dynamizmu do całej animacji. Natomiast, zły dobór skutkuje różnymi niepożądanymi zjawiskami, jak migotanie, brak płynności i inne, które mogą powodować niezadowalający rezultat.

Jak to wszystko wyświetlić na płaskim ekranie? Jak z animacji 3D zrobić animację 2D, która dawałaby złudzenie przestrzeni, mimo że jest całkowicie płaska? Proces ten nazywa się rzutowaniem. Wyróżnia się kilka sposobów rzutowania, i każdy ma swoje własności. Najprostszym jest rzutowanie równoległe niedające złudzenia perspektywy. Rzutowanie perspektywistyczne daje ową, ale za to jest bardziej złożone obliczeniowo. W większości przypadkach wykorzystywane są różne wersje rzutu perspektywistycznego dającego zdecydowanie lepsze rezultaty niż prosty rzut równoległy. Niemniej, do naszych potrzeb wystarczający jest ten prostszy. Rzutowanie jest – jak pokazuje **rysunek 1** – , procesem redukcji bryły trójwymiarowej na płaszczyznę.

Ukrywanie niewidocznych ścian

Jak sprawić, żeby bryła była widoczna tylko z tej strony, na którą patrzymy? Innymi słowy jak nie wyświetlać części, które nie są widoczne, bo są „z tyłu" albo są zasłaniane przez inne bryły czy inne części tej samej bryły? Na to pytanie nie ma ogólnej prostej odpowiedzi. W zależności od złożoności bryły techniki stają się coraz bardziej wyszukane i skomplikowane bądź, co istotne, są w miarę proste, ale wymagają dużej mocy obliczeniowej i sporej pamięci urządzenia. Animacja przeznaczona jest na prosty mikrokontroler, zatem musimy się spodziewać pewnych ograniczeń i nie oczekiwać rezultatów w stylu Hollywoodzkim. Ukrywanie ścian wykonywane jest w przypadku, gdy bryła uważana jest za nieprzezroczystą jak np. ściana, drzwi. Czasami pożądany jest jednak efekt przezroczystości, wtedy dopuszcza się pokazywanie niewidocznych części.

Realizacja transformacji

Punkty w przestrzeni trójwymiarowej przedstawiane są w postaci wektorowej. Dla punktu $p=(p_x, p_y, p_z)$ odpowiadający wektor ma następującą postać:

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

Dla operacji translacji, obrotu i skalowania wyznacza się macierze transformacji. Na **rys. 4** przedstawione są macierze: translacji o wektor (v_z,v_y,v_z), macierze obrotu o kąt α wokół osi Ox, Oy, Oz oraz macierz skalowania ze współczynnikiem proporcjonalności s.

$$T_v = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

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

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

$$S_v = \begin{bmatrix} v_x & 0 & 0 & 0 \\ 0 & v_y & 0 & 0 \\ 0 & 0 & v_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

W celu uzyskania współrzędnych punktu będącego wynikiem jednej z tych operacji wystarczy wymnożyć dany punkt **p** w postaci wektorowej z odpowiednią macierzą. Wykonywane jest tutaj mnożenie macierzy przez wektor. Otrzymany w wyniku tego wektor jest szukanym punktem. Operacje bardziej złożone można otrzymać poprzez składanie podstawowych operacji. Objawia się to zwykłym mnożeniem macierzy. Na przykład, dla wykonania translacji o wektor v, obrót wokół osi X o kąt α, ponownie wykonanie translacji o wektor v oraz obrót wokół osi Y o kąt β wystarczy obliczyć macierz:

Należy zwrócić uwagę na kolejność wykonywania operacji. Nie jest ona przemienna. Wynikiem powyższego mnożenia jest znowu macierz, co umożliwia budowanie dowolnie złożonych operacji nie zwiększając przy tym ani o bajt zużycia pamięci czy mocy obliczeniowej (przy założeniu, że macierz tą wyliczy się raz przed przystąpieniem do wykonywania animacji). Transformacja trójkąta czy całej bryły jest analogiczna. Wystarczy zastosować operację do każdego punktu bryły. Powodem używania zapisu macierzowego jest jego zwięzłość oraz przejrzystość. Te same wzory zapisane w klasycznym stylu, za pomocą formuł, zajęłyby znacznie więcej miejsca oraz stałyby się całkowicie nieczytelne. Co więcej, zapis taki zmniejsza ilość pracy wykonywanej przez człowieka przerzucając ją na maszynę.

Realizacja usuwania niewidocznych ścian

Usuwanie niewidocznych ścian realizowane jest za pomocą iloczynu wektorowego i iloczynu skalarnego wektorów (**rysunek 2**). Iloczyn wektorowy dwóch wektorów jest wektorem prostopadłym do płaszczyzny wyznaczonej przez te dwa wektory i razem z nimi tworzy układ „prawoskrętny". Iloczyn skalarny dwóch wektorów umożliwia nam wyznaczenie kąta, pod jakim są owe wektory do siebie ustawione. Dla dowolnego trójkąta ustalamy dwa boki i obliczamy iloczyn wektorowy wektorów wyznaczonych przez te boki. Jeżeli otrzymany wektor tworzy z wektorem [0,0,–1] kąt należący do przedziału (–90°,90°), to trójkąt ten jest ustawiony do obserwatora przednią częścią. W przeciwnym przypadku jest ustawiony tylną tj. jest niewidoczny. Należy podkreślić, że jest to jedna z najprostszych me-

Listing 1. Funkcja rzutująca
```
point projection_p(point3d p3){
  point p;  // punkt 2D
  p.x = p3.x;
  p.y = p3.y;
  return p;
}
```

Listing 2. Tablica elements
```
typedef struct {
  float x,y,z;
} point3d;
typedef struct {
  point3d p1,p2,p3;
} triangle3d;
triangle3d elements[12];
int elements_count;
```

tod tego typu. Nie stosuje się ona do bardziej złożonych brył albo do kilku brył, które mogą przysłaniać się nawzajem. W takim przypadku wykorzystywane są bardziej subtelne techniki analizujące strukturę i kształt obiektu oraz jego położenie w stosunku do innych obiektów czy do samego siebie. Inna techniką jest metoda Bufora Z, polegająca na obliczaniu dla każdego piksela odległości od ekranu. Dla dwóch nakładających się pikseli wyświetlany jest wtedy ten bliższy. Przy rozdzielczości wyświetlacza 320×240 pikseli oraz 3 bajtach na kolor wymagane by było co najmniej 230 kB pamięci RAM, co jest dosyć trudne do osiągnięcia dla mikrokontrolera (metoda *is_visible_t*).

Realizacja rzutowania

Rzutowanie równoległe jest zwykłą konwersją punktu (x,y,z) na punkt (x,y,0), który można interpretować jako punkt (x,y) i bezpośrednio wyświetlić na wyświetlaczu (**listing 1**).

Przykład animacji sześcianu

Sekwencję klatek wyświetlanych podczas animowania ruchu sześcianu na ekranie LCD pokazano na **rysunku 3**. Pierwszym krokiem jest stworzenie bryły – sześcianu. Jak wiemy

Listing 3. Funkcja generująca sześcian
```
void init_cube()
{
  angle=0;
  //front up, front down
  triangle3d fd = create_triangle3d(
                  create_point3d(-1,1,-1),
                  create_point3d(-1,-1,-1),
                  create_point3d(1,-1,-1));
  triangle3d fu = operation_t(rotation_z_matrix(180*pi/180),fd);
  // back down, back up
  triangle3d bd = operation_t(rotation_y_matrix(pi),fd);
  triangle3d bu = operation_t(rotation_y_matrix(pi),fu);
  // right down, right up
  triangle3d rd = operation_t(rotation_y_matrix(pi/2),fd);
  triangle3d ru = operation_t(rotation_y_matrix(pi/2),fu);
  // left down, left up
  triangle3d ld = operation_t(rotation_y_matrix(-pi/2),fd);
  triangle3d lu = operation_t(rotation_y_matrix(-pi/2),fu);
  // up down, up up
  triangle3d ud = operation_t(rotation_x_matrix(-pi/2),fd);
  triangle3d uu = operation_t(rotation_x_matrix(-pi/2),fu);
  // down down, down up
  triangle3d dd = operation_t(rotation_x_matrix(pi/2),fd);
  triangle3d du = operation_t(rotation_x_matrix(pi/2),fu);
  elements[0]=fd;
  elements[1]=fu;
  elements[2]=bd;
  elements[3]=bu;
  elements[4]=rd;
  elements[5]=ru;
  elements[6]=ld;
  elements[7]=lu;
  elements[8]=ud;
  elements[9]=uu;
  elements[10]=dd;
  elements[11]=du;
  elements_count=12;
}
```

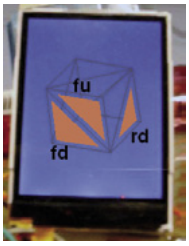
sześcian składa się z ośmiu wierzchołków i sześciu kwadratowych ścian, co daje nam 12 trójkątów i 36 punktów do wyznaczenia. Tablica *elements* (**listing 2**) przechowuje wszystkie trójkąty naszej bryły. Funkcja init_cube generuje bryłę. Sześcian będzie położony w początku układu współrzędnych, tj. jego środek będzie punktem (0,0,0). Bok będzie długości 2.0, rozciągając się od wartości –1.0 do +1.0. Pierwszy trójkąt reprezentujący przednią ścianę bryły jest trójkątem pokrywającym lewą dolną połówkę ściany (**rysunek 4**).

Żeby otrzymać drugą połówkę wystarczy obrócić pierwszy trójkąt o 180 stopni wokół osi Z. Oś Z jest osią prostopadłą do powierzchni monitora, o wartościach dodatnich za monitorem, i wartościach ujemnych przed tj. bliżej użytkownika. Oś X jest pozioma, oś Y pionowa (**rysunek 5**).

Drugą ścianę (tylnią) otrzymamy przez obrót pierwszej o kąt 180 stopni wokół osi. Trzecią ścianę (prawą) otrzymujemy poprzez obrót pierwszej o 90 stopni wokół osi Y. Ścianę lewą otrzymamy poprzez obrót pierwszej ściany o kąt –90 stopni wokół osi Y. Ścianę górną otrzymamy poprzez rotację pierwszej o kąt –90 stopni wokół osi X. Ostatecznie ściana dolna jest obrazem ściany przedniej w obrocie o kąt 90 wokół osi X. Wszystkie trójkąty zapisujemy do tablicy i ustawiamy jej wielkość na 12. Praktyczną realizację wymienionych korków działania programu pokazano na **listingu 3**.

Funkcje *rotation_x_matrix()*, *rotation_y_matrix()*, *rotation_z_matrix()* i *translation_matrix()* tworzą macierze przekształceń, natomiast funkcja *operation()* wymnaża wektor przez macierz. Mamy już bryłę, ożywmy ją zatem.

W funkcji *main()* naszego programu umieszczamy następujący kod pokazany na **listingu 4**. Funkcja *draw_cube()* jest odpowiedzialna za generowanie i wyświetlanie pojedynczej klatki animacji. Każda iteracja pętli generuje pojedynczą klatkę animacji (**listing 5**).



Rysunek 4. Triangulacja sześcianu (fd = front down, fu = front up, rd = right down)

Listing 4. Szkielet animacji bryły
```
init_cube();
while(true){
// czyść ekran
  ...
draw_cube();
...
// sleep
}
```

Listing 5. Funkcja odpowiedzialna za wygenerowanie pojedynczej klatki animacji i jej wyświetlenie

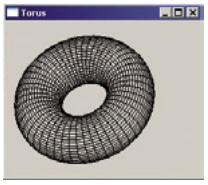
```
#define foreachi(i,n) for(int i=0;i<n;++i)
void main_function(){
  matrix bigger = scale_matrix(distance_z,distance_z,distance_z);
  matrix rotate_x = rotation_x_matrix(angle*pi/180);
  matrix rotate_y = rotation_y_matrix(angle*pi/180);
  matrix rotate_z = rotation_z_matrix(angle*pi/180);
  matrix move = translation_matrix(120,160,100/*distance_z*/);
  matrix anim = multiply_m(move, multiply_m(rotate_z,
    multiply_m(rotate_y, multiply_m(rotate_x,bigger))));
  point3d direction = create_point3d(0,0,1);
  foreachi(i,elements_count){
    if( is_visible_t(direction,operation_t(anim,elements[i]))) {
      drawTriangle(projection_t(operation_t(anim,elements[i])));
    }
  }
}
```

Listing 6. Tworzenie i wymnażanie macierzy przekształceń podstawowych

```
void main_function(){
  // tworzymy macierze przekształceń podstawowych
  matrix bigger = scale_matrix(distance_z,distance_z,distance_z);
  matrix rotate_x = rotation_x_matrix(angle*pi/180);
  matrix rotate_y = rotation_y_matrix(angle*pi/180);
  matrix rotate_z = rotation_z_matrix(angle*pi/180);
  matrix move = translation_matrix(120,160,100/*distance_z*/);
  // generujemy macierz przekształcenia
  matrix anim =
    multiply_m(move,
      multiply_m(rotate_z,
        multiply_m(rotate_y,
          multiply_m(rotate_x,bigger))));
  // wykonujemy obliczenia dla każdego trójkąta
  foreachi(i,elements_count){
  // sprawdzamy czy trójkąt jest widoczny
  if( is_visible_t(direction,operation_t(anim,elements[i]))) {
    drawTriangle(projection_t(operation_t(anim,elements[i])));
  }
}
```

Listing 7. Funkcja generująca torus

```
void init_torus2()
{
  #define CMAX 9
  point3d circle_plus[CMAX];
  point3d circle_minus[CMAX];
  float angle=0;
  foreachi(i,CMAX){
    angle= (((float)i)/CMAX)*(180.0) * pi/180*2;
    float x = cos(angle);
    float y = sin(angle);
    float z = 0;
    circle_plus[i] = create_point3d(x,y,z);
    circle_minus[i] = create_point3d(x,y,z);
  }
  #define XMAX 9
  matrix translation = translation_matrix(2,0,0);
  matrix rotate = rotation_y_matrix((0.5/XMAX)*pi*2);
  matrix anim = multiply_m(rotate,translation);
  foreachi(i,CMAX){
    circle_plus[i]  = operation_p(anim,circle_plus[i]);
  }
  rotate = rotation_y_matrix((-0.5/XMAX)*pi*2);
  anim = multiply_m(rotate,translation);
  foreachi(i,CMAX){
    circle_minus[i]  = operation_p(anim,circle_minus[i]);
  }
  //triangle3d ring[CMAX+1];
  foreachi(i,CMAX-1){
    triangle3d trl = create_triangle3d(
    circle_plus[i],
    circle_minus[i],
    circle_plus[i+1]);
    triangle3d trr = create_triangle3d(
      circle_minus[i+1],
      circle_plus[i+1],
      circle_minus[i]);
    elements[2*i] = trl;
    elements[2*i+1]= trr;
  }
  elements[CMAX*2]  = create_triangle3d(
    circle_plus[CMAX-1],
    circle_minus[CMAX-1],
    circle_plus[0]);
    elements[CMAX*2+1] = create_triangle3d(
      circle_minus[0],
      circle_plus[0],
      circle_minus[CMAX-1]);
  elements_count=2*(CMAX)+2;
  foreachi(q,XMAX-1){
    rotate = rotation_y_matrix((((float)(q+1))/XMAX)*pi*2);
    foreachi(i,elements_count){
      elements[i+elements_count*(q+1)]=operation_t(rotate,elements[i]);
    }
  }
  elements_count *= XMAX;
}
```



Rysunek 5. Torus z gęstą triangulacją

Pozostała ostatnia kwestia i animacja będzie gotowa. Jak wcześniej zostało powiedziane do stworzenia prostej animacji wystarczą podstawowe trzy operacje. Składając odpowiednią ich ilość jesteśmy w stanie utworzyć dość ciekawe i złożone ruchy, na pewno wystarczające dla naszych potrzeb. W tym przykładzie animacja składać się będzie z obrotu sześcianu o jeden stopień we wszystkich trzech kierunkach. W rezultacie otrzymamy obracającą się kostkę w pozornie losowy sposób. Na początku tworzymy macierze przekształceń podstawowych. Następnie wymnażamy macierze w celu otrzymania jednej, reprezentującej całą sekwencję poprzednich. Ostatecznie dla wszystkich trójkątów z tabeli *elements*, reprezentującą naszą bryłę wykonujemy operację *anim*, jednocześnie sprawdzając czy dana ściana po transformacji jest widoczna. Operacje te pokazano na **listingu 6**.

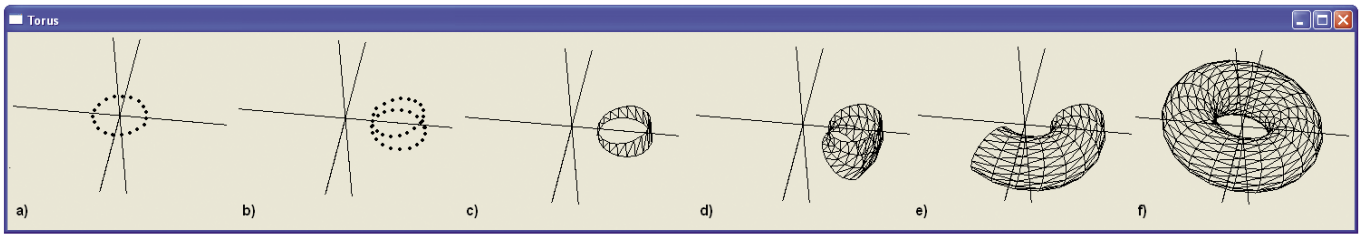
Przykład animacji torusa

Przykład z torusem (**rysunek 5**) jest bardziej złożony. Wygenerowanie odpowiednich trójkątów składających się na całego torusa jest dosyć żmudną pracą.

Na początku wygenerujemy dwa identyczne zestawy punktów na okręgu znajdującym się w środku układu współrzędnych jak pokazuje **rysunek 6a**. Następnie odsuniemy je troszkę od środka i pierwszy okrąg obrócimy odrobinę w prawo, drugi odrobinę w lewo – **rysunek 6b**. Następnie połączymy odpowiednie punkty na obydwu okręgach tworząc trójkąty. Otrzymamy w ten sposób mały skrawek docelowego torusa – **rysunek 6c**. Na końcu wystarczy powielić tę część odpowiednią ilość razy by otrzymać gotową bryłę – **rysunki 6d…6f**. Sposób definiowania bryły torusa pokazano na **listingu 7**. Proces animacji pozostaje taki sam jak w przypadku sześcianu. Podążając zgodnie z powyższymi metodami tworzenia i animowania figur można stworzyć dowolną bryłę tj. klin pryzma, oktaedr, ostrosłup, kula, walec, stożek, torus podwójny, wstęgę Mobiusa (powierzchnia). Dowolne łączenie figur może skutkować lepszym efektem jak klasyczny przykład dzbanka 3D. Ograniczeniem mogą być nieregularne kształty bryły, których punkty podczas tworzenia będzie trzeba ręcznie wygenerować.

Optymalizacja

W związku z ograniczoną mocą obliczeniową mikrokontrolera jak i szybkością obsługi LCD animacje oddają wrażenie migotania. Można ten efekt zadowalająco zniwelować



Rysunek 6. Etapy tworzenie przykładowej bryły – torus

Listing 8.Struktury wykorzystywane w programie

```
typedef struct{
  float x,y;
} point;
typedef struct {
  float x,y,z;
} point3d;
typedef struct {
  point p1,p2,p3;
} triangle;
typedef struct {
  point3d p1,p2,p3;
} triangle3d;
typedef struct {
  float value[4][4];
} matrix;
typedef struct {
  float value[4];
} vector;
```

Listing 9. Wybrane metody wykorzystywane w procesie generowania animacji

```
matrix translation_matrix(float x, float y, float z){
  matrix m = identity_matrix();
  m.value[0][3] = x;
  m.value[1][3] = y;
  m.value[2][3]= z;
  return m;
}

matrix rotation_x_matrix(float alpha){
  matrix m = null_matrix();
  m.value[0][0]=1;
  m.value[1][1]=cos(alpha);
  m.value[2][2]=cos(alpha);
  m.value[2][1]=-sin(alpha);
  m.value[1][2]=sin(alpha);
  m.value[3][3]=1;
  return m;
}
```

Listing 10. Metoda rozstrzygająca widoczność ścian (trójkąta)

```
int is_visible_t(point3d direction, triangle3d tr){
  point3d first_vector,second_vector,multiplied;
  first_vector = create_vector3d(tr.p1,tr.p2);
  second_vector = create_vector3d(tr.p1,tr.p3);
  // iloczyn wektorowy
  multiplied = vector_multiplication(first_vector,second_vector);
  // iloczyn skalarny
  float angle = scalar_multiplication(direction,multiplied);
  if( angle < 0 ){
    return 0; // ściana niewidoczna
  }else {
    return 1; // ściana widoczna
  }
}
```

wyświetlając figury w następujący sposób. Wyświetlamy figurę, obracamy ją o ustalony kąt i wyświetlamy ją jeszcze raz. Teraz na ekranie mamy dwie figury niejako rozmazane. Zmazujemy pierwszą figurę np. przez narysowanie jej w kolorze tła, po czym obracamy o kąt i rysujemy kolejną. Co nam daje taki sposób? Na ekranie zawsze jest narysowana jedna albo dwie figury, dlatego proces rysowania kolejnej przesuniętej o dany kąt nie jest tak widoczny. W praktyce dzięki takiemu podejściu wyświetlanie animacji bryły na całym ekranie LCD jest w pełni płynne.

Uwagi końcowe

Na **listingu 8** pokazano struktury wykorzystywane w programie. Jak można zauważyć, służą one do deklarowania typów zmiennych zawierających współrzędne punktu na płaszczyźnie, punktu w przestrzeni lub wektora w przestrzeni. Definiowanie struktur tego typu ułatwia pracę nad programem, ponieważ wszystkie parametry (np. współrzędne punktu) są przechowywane w pojedynczej zmiennej. Na **listingach 9** i **10** zamieszczono, odpowiednio – przykłady funkcji zawierających wybrane metody generowania animacji oraz algorytm rozstrzygający widoczność ścian animowanego trójkąta. Po lekturze artykułu, samodzielna analiza sposobu ich działania nie powinna nastręczać żadnego problemu.

W artykule omówiono podstawowe metody tworzenia animacji za pomocą przekształceń brył. Dosyć użyteczną metodą i jednocześnie stosunkowo prostą do wykonania, ale wymagającą sporych zasobów pamięci, jest przechowywanie w pamięci mikrokontrolera gotowych klatek (obrazów) do wyświetlenia, utworzonych na przykład za pomocą programów graficznych. Ta metoda jest jednak łatwa do samodzielnego zaimplementowania po opanowaniu 1-szej części tego kursu.

Tomasz Rzędowski
tomasz.rzedowski@gmail.com