Laboratorium Grafiki Komputerowej i Animacji

Ćwiczenie III

Biblioteka OpenGL - wprowadzenie, obiekty trójwymiarowe: punkty, linie, wielokąty

Sławomir Samolej

Rzeszów, 1999

1. Wprowadzenie

OpenGL nie jest językiem programowania, jest API (Application Programming Interface) - interfejsem programowym aplikacji. Jeśli kiedykolwiek mówimy o programach opartych na OpenGL lub o aplikacjach OpenGL to powinniśmy mieć na myśli, że został napisany w jakimś języku programowania (takim jak C, C++) i zawiera wywołania do jednej lub więcej bibliotek OpenGL.

Z punktu widzenia programisty OpenGL jest zestawem funkcji napisanych zgodnie z formalizmem przyjętym w języku C. Tym samym najbardziej naturalnym sposobem wykorzystania OpenGL jest umieszczanie wywołań funkcji w programie napisanym w języku C lub C++. Warto nadmienić, że inne języki programowania - tak zwane 4GL ("fourth-generation languages") języki czwartej generacji jak Visual Basic - są w stanie wywoływać funkcje z bibliotek napisanych w języku C i także mogą być wykorzystywane do budowania aplikacji OpenGL.

OpenGL dzieli się na trzy podstawowe biblioteki:

- biblioteka Auxiliary lub AUX ("pomocnicza") zawarta w pliku glaux.lib. Deklaracje funkcji zawartych w tej bibliotece znajdują się w pliku nagłówkowym glaux.h. Funkcje zawarte w tej bibliotece nie są zawarte w specyfikacji OpenGL. Są raczej zestawem narzędzi umożliwiających uruchamianie aplikacji OpenGL na dowolnej platformie sprzętowej i systemowej. Biblioteka AUX umożliwia w najszybszy sposób tworzenie programów wykorzystujących funkcje OpenGL bez konieczności wnikania w formalizm tworzenia oprogramowania na danej platformie. Wadą programów tworzonych przy pomocy tej biblioteki jest, że nie mogą one w pełni wykorzystywać specyficznych dla danego systemu funkcji. Funkcje należące do tej biblioteki mają przedrostek *aux*.

- biblioteka GL jest zestawem podstawowych funkcji zdefiniowanych w standardzie OpenGL. Zawarta jest w pliku opengl32.dll, jej plik nagłówkowy to gl.h. Funkcje należące do tej biblioteki mają przedrostek *gl*.

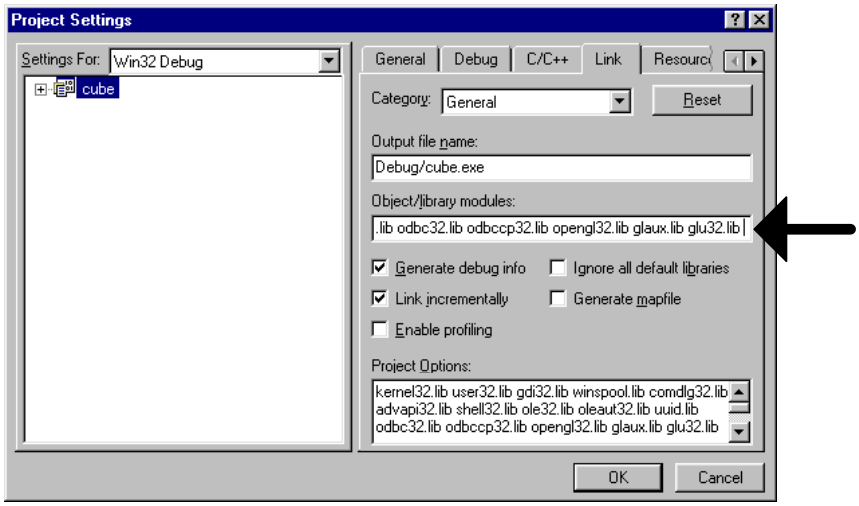
- biblioteka GLU (ang. utility library - biblioteka narzędziowa) zawarta w pliku glu32.dll. Jej plik nagłówkowy to glu.h. Biblioteka zawiera funkcje narzędziowe ułatwiające tworzenie aplikacji OpenGL, takie jak funkcje rysujące sfery, dyski, walce. Napisana jest przy użyciu komend OpenGL, co gwarantuje jej dostępność na wszystkich platformach, na których została zaimplementowana biblioteka OpenGL.

Nowsze systemy operacyjne jak Windows 95 OSR, Windows NT4.0/3.51, Windows 98 (?) posiadają wbudowaną bibliotekę OpenGL w postaci dwu wcześniej wymienionych plików: opengl32.dll i glu32.dll umieszczonych w katalogu "System". Dla starszych wersji systemu Windows 95 bibliotekę można doinstalować z plików dostępnych na stronach internetowych firm Siliocon Graphics (www.sgi.com), lub Microsoft (www.microsoft.com). Instalacja polega na przekopiowaniu bibliotek dll do katalogu "System".

Opracowanie omawia sposób tworzenia aplikacji w języku C, które wykorzystują rozkazy OpenGL. Zaprezentowano zarówno podstawy tworzenia programów z wykorzystaniem biblioteki AUX, jak i specyfikę programowania aplikacji zgodnych z Win32. Druga część opracowania omawia sposoby tworzenia podstawowych elementów graficznych w bibliotece OpenGL - tak zwanych prymitywów graficznych. Wszystkie pełne kody źródłowe i projekty omawianych w opracowaniu aplikacji znajdują się na załączonej dyskietce. Szczegóły dotyczące tworzenia oprogramowania dla systemu Windows znaleźć można w opracowaniach do ćwiczeń I i II z przedmiotu Grafika Komputerowa i Animacja.

2. Tworzenie aplikacji z wykorzystaniem biblioteki OpenGL.

Przed próbą uruchomienia programów zawierających wywołania funkcji OpenGL należy dokonać pewnych modyfikacji w ustawieniach kompilatora. Pierwszą z nich jest po stworzeniu projektu włączenie do zestawu bibliotek dostępnych do konsolidacji trzech dodatkowych plików: glaux.lib, glu32.lib i opengl32.lib. W pakiecie Visual C++ 5.0 należy wybrać opcję menu "Project" a następnie pole "Settings...". Po kilknięciu pola "Settings..." pojawi się okno dialogowe. Należy wybrać zakładkę "Link" i uzupełnić listę bibliotek w miejscu wskazanym na rysunku 2.1.



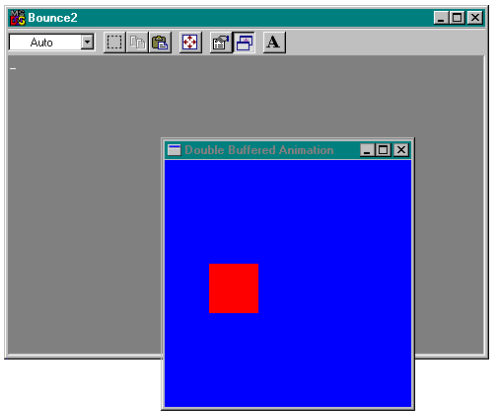
Rys 2.1 Sposób uzupełniania listy bibliotek dostępnych do konsolidacji

2.1 Biblioteka AUX

Biblioteka AUX pomyślana została jako narzędzie do szybkiego tworzenia aplikacji wykorzystujących funkcje OpenGL bez konieczności zaznajamiania się ze specyfiką programowania w danym środowisku. Główny zestaw funkcji biblioteki AUX dostępny jest niemal we wszystkich implementacjach OpenGL. Oznacza to w praktyce, że program zawierający wywołania funkcji OpenGL i napisany przy wykorzystaniu biblioteki AUX jest przenośny na dowolną platformę systemową (Wystarczy tylko kod źródłowy programu poddać kompilacji w nowym środowisku.). Podstawowe funkcje biblioteki AUX zajmują się tworzeniem i obsługą okien programowych oraz obsługą komunikatów użytkownika wysyłanych do programu. Pozostałe zaś zawierają kody pozwalające wyświetlać pewne kompletne gotowe trójwymiarowe obiekty.

Poniżej zostanie zaprezentowany i omówiony przykładowy program wykorzystujący bibliotekę AUX do stworzenia prostej animacji. Program tworzy okno z niebieskim kolorem wypełnienia i wyświetla w nim poruszający się czerwony kwadrat (rys 2.1.1) Kwadrat porusza się w taki sposób, że w chwili gdy napotyka "krawędź" okna to odbija się od niej. Kod omawianego programu znajduje się w katalogu "bounce2" na dołączonej do opracowania dyskietki. Wszystkie programy wykorzystujące podstawowe funkcje budujące okno i interfejs użytkownika przy pomocy biblioteki AUX mają strukturę jak typowe aplikacje zgodne z ANSI C. To znaczy główną funkcją programu jest main(). Aby dokonywać modyfikacji tych programów, lub aby tworzyć własne tego typu programy należy traktować je w środowisku Visual C++ jako *"Win 32 Console Application"* (Tworzenie tego typu projektów omówione

zostało w punkcie 3 opracowania do ćwiczenia I z przedmiotu Grafika Komputerowa i Animacja).



. Rys 2.1.1 Okno programu Bounce2.

Poniżej zaprezentowano kod programu bounce2:

```
// bounce2.c
// Animacja z podwójnym buforowaniem

#include <windows.h>    // Plik nagłówkowy systemu Windows
#include <gl\gl.h>      // biblioteka OpenGL
#include <gl\glaux.h>   // biblioteka AUX

// Początkowa pozycja i rozmiar kwadratu
GLfloat x1 = 100.0f;
GLfloat y1 = 150.0f;
GLsizei rsize = 50;

// Rozmiar kroku w kierunku x i y
// (ilość pikseli o jaką ma się odbywać ruch
// za każdym razem)
GLfloat xstep = 1.0f;
GLfloat ystep = 1.0f;

// Zmienne do zachowywania rozmiarów okna programu
GLfloat windowWidth;
GLfloat windowHeight;

// Funkcja wywoływana przez AUX w chwili gdy nastąpiła
// zmiana rozmiaru okna programu
void CALLBACK ChangeSize(GLsizei w, GLsizei h)
      {
      // Zabezpieczenie przed dzieleniem przez 0 gdy okno
      // jest zbyt wąskie (dotyczy to tylko wysokości okna)
      if(h == 0)
            h = 1;

      // Ustaw viewport jako całe okno
    glViewport(0, 0, w, h);

      // Zeruj położenie układu współrzędnych
    glLoadIdentity();

      // Kod zapewniający kwadratowy kształt poruszającego się
      // obiektu niezależnie od proporcji pomiędzy wysokością i
      // szerokością okan programu
```

```
      // Zachowanie rozmiarów okna do dalszych obliczeń
      if (w <= h)
            {
            windowHeight = 250.0f*h/w;
            windowWidth = 250.0f;
            }
    else
            {
            windowWidth = 250.0f*w/h;
            windowHeight = 250.0f;
            }

      // Ustalenie odwzorowania przestrzeni
      glOrtho(0.0f, windowWidth, 0.0f, windowHeight, 1.0f, -1.0f);
      }

// Funkcja wywoływana przez AUX aby uaktualnić zawartość okna
void CALLBACK RenderScene(void)
      {
      // Ustal kolor tła na niebieski
      glClearColor(0.0f, 0.0f, 1.0f, 1.0f);

      // Wyczyść okno przy pomocy ustalonego koloru
      glClear(GL_COLOR_BUFFER_BIT);

      // Ustal kolor malowanych obiektów an czerwony i rysuj
      // kwadrat w ustalonej bieżącej pozycji
      glColor3f(1.0f, 0.0f, 0.0f);
      glRectf(x1, y1, x1+rsize, y1+rsize);

      glFlush();

      // Zawartość namalowanego obrazu prześlij na ekran
      auxSwapBuffers();
      }

// Funkcja wywoływana przez bibliotekę AUX gdy okno nie
// zmienia rozmiaru i nie jest przemieszczane po ekranie
void CALLBACK IdleFunction(void)
      {
      // Zmień kierunek, gdy obiekt osiągnie lewą lub prawą
      // krawędź okna
      if(x1 > windowWidth-rsize || x1 < 0)
            xstep = -xstep;

      // Zmień kierunek, gdy obiekt osiągnie górną lub dolną
      // krawędź okna
      if(y1 > windowHeight-rsize || y1 < 0)
            ystep = -ystep;

      // Sprawdź rozmiary okna na wypadek, gdyby po pomniejszeniu
      // okna kwadrat znalazł się poza obszarem wyświetlania
      if(x1 > windowWidth-rsize)
            x1 = windowWidth-rsize-1;

      if(y1 > windowHeight-rsize)
            y1 = windowHeight-rsize-1;

      // Przemieść kwadrat
      x1 += xstep;
      y1 += ystep;

      // Odrysuj scenę z nowymi koordynatami kwadratu
      RenderScene();
      }
```

```
// Funkcja główna programu
void main(void)
      {
      // Inicjalizacja okna przy pomocy biblioteki AUX
      auxInitDisplayMode(AUX_DOUBLE | AUX_RGBA);
      auxInitPosition(100,100,250,250);
      auxInitWindow("Double Buffered Animation");

      // Ustalenie funkcji, która ma być wywoływana w przypadku
      // zmiany rozmiaru okna
      auxReshapeFunc(ChangeSize);

      // Ustalenie funkcji, która ma być wywoływana, gdy program
      // nie przetwarza komunikatów od systemu
      auxIdleFunc(IdleFunction);

      // Uruchomienie głównej pętli programu
      auxMainLoop(RenderScene);
      }
```

Do kodu programu dołączono trzy plik nagłówkowe:

```
#include <windows.h>    // Plik nagłówkowy systemu Windows
#include <gl\gl.h>      // biblioteka OpenGL
#include <gl\glaux.h>   // biblioteka AUX
```

Plik windows.h zawiera prototypy funkcji wymaganych przez system windows do obsługi interfejsu użytkownika. Nagłówki gl.h i glaux.h zawierają odpowiednio funkcje OpenGL oraz funkcje biblioteki AUX.

Główna funkcja programu (main()) rozpoczyna się od wywołania funkcji biblioteki AUX definiujących okno programu i jego właściwości.

Linia:

```
auxInitDisplayMode(AUX_DOUBLE | AUX_RGBA);
```

ustala tryb wyświetlania podczas tworzenia okna programu. Flaga AUX_DOUBLE decyduje, że okno wyświetlane będzie w trybie podwójnego buforowania. Podwójne buforowania okna oznacza, że nowa zawartość okna przed wyświetleniem "rysowana" jest do bufora w pamięci komputera (bądź karty graficznej). Dopiero wywołanie odpowiedniej funkcji (w programie pisanym przy pomocy biblioteki AUX jest to funkcja auxSwapBuffers()) powoduje przesłanie do okna programu nowej, w pełni narysowanej klatki animacji. Tryb wyświetlania z podwójnym buforowaniem jest szczególnie przydatny do tworzenia animacji przy pomocy biblioteki OpenGL. Włączenie zamiast flagi AUX_DOUBLE stałej AUX_SINGLE spowoduje, że odnawianie zawartości okna odbywać się będzie zawsze bezpośrednio na ekranie. Flaga AUX_RGBA oznacza, że program będzie respektował definiowanie kolorów poprzez podawanie mu trzech składowych: czerwonej (R), zielonej (G) i niebieskiej (B).

Linia:

```
auxInitPosition(100,100,250,250);
```

ustala początkową pozycję okna na ekranie (100,100) oraz rozmiary okna (250×250 pikseli).

Wywołanie funkcji:

```
auxInitWindow("Double Buffered Animation");
```

tworzy na ekranie okno i decyduje o napisie jaki znajduje się w pasku tytułu programu.

Ostatnia funkcja wywoływana w funkcji głównej programu ma postać:

```
auxMainLoop(RenderScene);
```

Funkcja auxMainLoop() zapewnia pracę programu w systemie dopóki nie zostanie ona przerwana przez zamknięcie okna. Jej parametrem jest wskaźnik na funkcję, która ma być wywoływana jeśli kiedykolwiek zawartość okna programu ma być uaktualniona (pierwsze wyrysowanie zawartości okna, przesunięcie, bądź zmiana rozmiaru okna, odsłonięcie fragmentu okna uprzednio zasłoniętego przez inne okno).

Funkcją odpowiedzialną za wyrysowywanie zawartości okna w programie Bounce2 jest RenderScene(). W funkcji tej powinny znajdować się wywołania komend OpenGL odpowiedzialnych za tworzenie wyświetlanej sceny graficznej. W programie Bounce2 funkcja w kolejności ustala kolor tła (**glClearColor()**), czyści okno kolorem tła (**glClear()**) następnie ustala kolor rysowanego obiektu (**glColor3f()**) i rysuje w zadanym położeniu kwadrat (**glRectf()**). Funkcja **glFlush()** informuje OpenGL, że należy zakończyć rysowanie w oknie i umożliwić dalsze wykonywanie programu. Wywołanie funkcji **auxSwapBuffers()** powoduje przełączenie buforów programu i wyświetlenie obszarze roboczym okna programu nowej klatki animacji.

W prawie wszystkich środowiskach systemowych zawierających okna programów użytkownik może w każdej chwili zmieniać ich rozmiar. Najkorzystniejszym sposobem reakcji na zmianę rozmiaru okna w przypadku programów wyświetlających sceny graficzne jest automatyczne dostosowywanie rozmiarów rysowanej sceny do rozmiarów okna, przy czym ważne jest by kształty rysowanych obiektów nie uległy deformacji. Aby umożliwić programiście dokonanie niezbędnych transformacji w chwili zmiany rozmiaru okna biblioteka AUX udostępnia funkcję **auxReshapeFunc()**. Parametrem wywołania tej funkcji jest wskaźnik na funkcję, która ma być automatycznie uruchamiana w przypadku wykrycia zmiany rozmiarów okna programu. W programie Bounce2 funkcją wywoływaną w omówionej sytuacji jest ChangeSize(). Przejmuje ona od systemu aktualne rozmiary okna. Podstawowe znaczenie w funkcji odgrywają wywołanie: **glViewport()** oraz **glOrtho()**. Pierwsza z nich po modyfikacji rozmiarów okna określa pole widzenia obserwatora (ang viewport), w taki sposób, że pole widzenia obejmuje całe okno. Druga zaś definiuje na podstawie informacji o stosunku wysokości do szerokości okna taki ostrosłup widzenia, że zachowane pozostaną proporcje wymiarów animowanego obiektu.

Podstawowym elementem programów graficznych jest możliwość tworzenia animacji. Funkcją biblioteki AUX wspomagającą konstruowanie prostych animacji jest **auxIdleFunc()**. Parametrem wywołania tej funkcji jest wskaźnik na funkcję, która ma być wywoływana w chwili, gdy program nie przetwarza komunikatów systemowych dochodzących do niego. W programie Bounce2 jako parametr wywołania funkcji auxIdleFunc() znajduje się wskaźnik do funkcji IdleFunction(). Funkcja IdleFunction() na podstawie poprzedniego położenia kwadratu na scenie i na podstawie danych dotyczących rozmiarów okna programu oblicza nowe położenie animowanego obiektu, a następnie wywołuje funkcję RenderScene() w celu uaktualnienia zawartości okna programu.

Biblioteka AUX powstała jako narzędzie niezależne od platformy systemowej umożliwiające w sposób uniwersalny kreowanie graficznych aplikacji z wykorzystaniem funkcji OpenGL. Niewątpliwą zaletą tego rozwiązania jest możliwość tworzenia oprogramowania zgodnego z wieloma systemami operacyjnymi i platformami sprzętowymi. Wadą stosowania tej biblioteki jest niemożliwość korzystania z szeregu dodatkowych własności, narzędzi i zasobów, które oferują konkretne systemy. Stąd w szeregu systemach

operacyjnych utworzone zostały dodatkowe funkcje, które umożliwiają włączenie kodu OpenGL w kody aplikcji zgodnych z daną platformą.

2.2 Aplikacje OpenGL zgodne z Win32.

Aby umożliwić włączanie w kod programów zgodnych z Win32 wywołań funkcji OpenGL w systemie Microsoft Windows zdefiniowano sześć dodatkowych funkcji wchodzących w skład biblioteki OpenGL dla Windows (posiadają one przedrostki *wgl*) oraz pięć dodatkowych funkcji wspomagających graficzny interfejs użytkownika Windows. Poniżej zaprezentowany zostanie sposób tworzenia oprogramowania wykorzystującego standard OpenGL i formalizm tworzenia aplikacji zgodnych z Win32.

2.2.1 Kontekst renderowania (ang. rndering context)

Gdy posługujemy się standardowym interfejsem graficznym Microsoft Windows przed jakąkolwiek próbą modyfikacji zawartości okna musimy uzyskać dostęp do tak zwanego kontekstu urządzenia (ang. device context) (Patrz materiały pomocnicze do ćwiczenia I z przedmiotu Grafika Komputerowa i Animacja). Malowanie w oknie w systemie Windows polega z punktu widzenia programisty na wysyłaniu rozkazów rysowania do kontekstu urządzenia. Dodatkowo kontekst urządzenia "pamięta" ustawienia trybu w jakim rysujemy na ekranie.

Odpowiednikiem kontekstu urządzenia graficznego interfejsu Windows jest kontekst renderowania (ang.rendering context) definiowany na potrzeby interfejsu OpenGL. Podobnie jak kontekstu urządzenia rolą kontekstu renderowania jest zapamiętywanie bieżących ustawień dotyczących sposobu prezentacji obiektów w oknie programu.

Kontekst renderowania nie został wymyślony przez twórców OpenGL, powstał on raczej jako dodatkowy element systemu Windows służący do wspomagania OpenGL. W celu umożliwienia tworzenia aplikacji OpenGL na platformie Windows do standartowego zestawu funkcji Windows dodano sześć nowych. Trzy najważniejsze z nich to:

```
HGLRC wglCreateContext(HDC hDC);
BOOL wglDeleteContext(HGLRC hrc);
BOOL wglMakeCurrent(HDC hDC, HGLRC hrc);
```

Nowym typem danych, który zdefiniowano jest HGLRC - uchwyt do kontekstu renderowania. Funkcja wglCreateContext() pobiera uchwyt kontekstu urządzenia i zwraca uchwyt do kontekstu renderowania OpenGL. Funkcja wglDeleteContext() służy do usunięcia kontekstu renderowania i powinna być wywołana przed zakończeniem działania aplikacji (najlepiej w obsłudze komunikatu WM_DESTROY). Funkcja wglMakeCurrent() ustala stworzony kontekst renderowania jako aktualnie obowiązujący dla danego urządzenia. (W programie można stworzyć sobie kilka kontekstów renderowania, lecz podczas rysowania bieżącym, obowiązującym może być tylko jeden z nich - ten ustalony przy pomocy funkcji wglMakeCurrent()). Przed zakończeniem działania programu należy odłączyć bieżący kontekst renderowania. Można tego dokonać wywołując funkcję wglMakeCurrent() z parametrem NULL w miejscu na uchwyt kontekstu renderowania.

2.2.2 Przygotowanie programu do rysowania przy pomocy komend OpenGL

Aby program napisany zgodnie ze standardem Win32 był w stanie wykorzystywać komendy OpenGL należy uzupełnić go o wywołania dodatkowych funkcji (*wgl*) przy

obsługiwaniu odpowiednich komunikatów systemu. Wymagane jest także wniesienie pewnych modyfikacji w sposób definiowania okien programowych oraz stworzenie specyficznych struktur danych przechowujących informacje dotyczące współpracy urządzeń graficznych systemu z biblioteką OpenGL.

Poniżej zaprezentowany zostanie sposób tworzenia i usuwania kontekstu renderowania w głównej procedurze obsługującej komunikaty systemu. Do tworzenia i niszczenia kontekstu renderowania najlepiej nadają się fragmenty kodu programu obsługujące komunikaty WM_CREATE i WM_DESTROY. Naturalnie kontekst renderowania tworzony jest w obsłudze komunikatu WM_CREATE i niszczony w obsłudze komunikatu WM_DESTROY. Poniższy fragment kodu prezentuje tworzenie i niszczenie kontekstu renderowania:

```
LRESULT CALLBACK WndProc(     HWND  hWnd,
                              UINT        message,
                              WPARAM      wParam,
                              LPARAM      lParam
                        )
      {
      static HGLRC hRC;       // Kontekst renderowania
      static HDC hDC;         // Kontekst urządzenia

      switch (message)
            {
            case WM_CREATE:
                  hDC = GetDC(hWnd);

                  // Utwórz kontekst renderowania i ustaw go jako
                        // bieżący
                  hRC = wglCreateContext(hDC);
                  wglMakeCurrent(hDC, hRC);
                  break;

            case WM_DESTROY:
                  // Odłącz kontekst renderowania i usuń go
                  wglMakeCurrent(hDC,NULL);
                  wglDeleteContext(hRC);
                  PostQuitMessage(0);
                  break;
            // ...
            }
    return (0L);
}
```

Rysowanie przy pomocy komend OpenGL powinno odbywać się, podobnie jak w typowych aplikacjach Windows podczas obsługiwania komunikatu WM_PAINT. Typowa obsługa komunikatu WM_PAINT może wyglądać jak poniżej:

```
case WM_PAINT:
                  {
                  // Funkcja rysująca przy pomocy komend OpenGL
                  RenderScene();
                  // Zabezpieczenie przed zbędnymi wywołaniami
                        // komunikatu WM_PAINT
                  ValidateRect(hWnd,NULL);
                  }
                  break;
```

Aby możliwe było rysowanie w danym oknie przy pomocy komend OpenGL, okno w momencie tworzenia (wywołanie funkcji CreateWindow()) musi mieć przekazane parametry

WS_CLIPCHILDREN i WS_CLIPSIBLINGS. Przykładowe wywołanie funkcji tworzącej okno zaprezentowano poniżej:

```
// ...
hWnd = CreateWindow(
                        lpszAppName,
                        lpszAppName,

                    // OpenGL wymaga WS_CLIPCHILDREN i WS_CLIPSIBLINGS
                          WS_OVERLAPPEDWINDOW |
                          WS_CLIPCHILDREN | WS_CLIPSIBLINGS,

                        100, 100,
                        250, 250,
                        NULL,
                        NULL,
                        hInstance,
                        NULL);
// ...
```

Rysowanie w oknie przy pomocy komend OpenGL wymaga także zdefiniowania tak zwanego formatu pikseli (ang. pixel format). Podobnie jak kontekst renderowania format pikseli nie jest częścią składową standardu OpenGL. Jest rozszerzeniem interfejsu Win32 dla wspomagania biblioteki OpenGL. Format pikseli ustawia takie właściwości kontekstu urządzenia powiązane z OpenGL jak bufor głębokości, sposób odwzorowywania kolorów oraz czy okno będzie wyświetlane w trybie podwójnego buforowania. Format pikseli dla danego kontekstu urządzenia musi zostać ustalony zanim zostanie on wykorzystany do stworzenia kontekstu renderowania. Do tworzenia formatu pixeli służą dwie funkcje:

```
int ChoosePixelFormat(HDC hDC, PIXELFORMATDESCRIPTOR *ppfd);
BOOL SetPixelFormat(HDC hDC,
                        int iPixelFormat,
                        PIXELFORMATDESCRIPTOR *ppfd
                      );
```

Tworzenie formatu pikseli odbywa się w trzech etapach. Po pierwsze należy wypełnić odpowiednimi wartościami strukturę typu PIXELFORMATDESCRIPTOR mając na uwadze w jaki sposób okno tworzone w programie ma współpracować z funkcjami OpenGL. Następnie należy przekazać stworzoną strukturę do funkcji ChoosePixelFormat(). Funkcja zwraca wartość całkowitą identyfikującą dostępny format pixeli kontekstu urządzenia najlepiej odpowiadający zestawowi danych przekazanych przez strukturę typu PIXELFORMATDESCRIPTOR. Liczba ta powinna być przekazana do funkcji SetPixelFormat() ostatecznie ustalającej format pixeli dla danego kontekstu urządzenia. Fragment kodu poniżej prezentuje prawidłową sekwencję czynności niezbędną do ustalenia formatu pixeli dla danego kontekstu urządzenia, na podstawie którego tworzony będzie kontekst renderowania:

```
PIXELFORMATDESCRIPTOR pixelFormat;
int nFormatIndex;
HDC hDC;

// Inicjalizuj strukturę pixelFormat
// ...

nFormatIndex=CoosePixelFormat(hDC,&pixelFormat);
SetPixelFormat(hDC, nPixelFormat, &pixelFormat);
```

Przykładowy sposób wypełnienia struktury PIXELFORMATDESCRIPTOR prezentuje fragment kodu poniżej:

```
void SetDCPixelFormat(HDC hDC)
{
int nPixelFormat;

static PIXELFORMATDESCRIPTOR pfd = {
            sizeof(PIXELFORMATDESCRIPTOR), // Rozmiar struktury
            1,                             // Wersja struktury (musi
                                              // być 1)
            PFD_DRAW_TO_WINDOW |           // Rysuj w oknie
                                           // nie do bitmapy
            PFD_SUPPORT_OPENGL |           // Wspomagaj wywołania
                                              // funkcji OpenGL
            PFD_DOUBLEBUFFER,              // Włącz podwójne
                                              // buforowanie
            PFD_TYPE_RGBA,                 // Tryb kolorów RGBA
            8,                             // 8 bitowy kolor
            0,0,0,0,0,0,                   // nie używane w ustawianiu
            0,0,                           // nie używane w ustawianiu
            0,0,0,0,0,                     // nie używane w ustawianiu
            16,                            // rozmiar bufora
                                              // głębokości
            0,                             // nie używane w ustawianiu
            0,                             // nie używane w ustawianiu
            PFD_MAIN_PLANE,                // rysuj w głównym planie
            0,                             // nie używane w ustawianiu
            0,0,0 };                       // nie używane w ustawianiu

      // Wybierz format pikseli najlepiej odpowiadający ustawieniom
      // w strukturze pdf
      nPixelFormat = ChoosePixelFormat(hDC, &pfd);

      // Ustaw format pikseli dla danego kontekstu urządzenia
      SetPixelFormat(hDC, nPixelFormat, &pfd);
      }
```

2.2.3 Przykładowy program

Tworzenie programu zgodnego z Win32 i wykorzystującego funkcje OpenGL polega na wykorzystaniu wybranych komunikatów systemowych do sterowania animacją, lub informacjami przekazywanymi przez użytkownika. Istotne znaczenie ma umiejętność wkomponowania w obsługę odpowiednich komunikatów wywołań funkcji tworzących programowy pomost pomiędzy biblioteką OpenGL i Windows. Poniżej zaprezentowany i omówiony zostanie przykładowy program dla Windows wykorzystujący funkcje OpenGL:

```
#include <windows.h>
#include <gl\gl.h>

// Początkowa pozycja kwadratu
GLfloat x1 = 100.0f;
GLfloat y1 = 150.0f;
GLsizei rsize = 50;

// Rozmiar kroku w kierunkach x i y
// (ilość pikseli do przemieszczenia w każdym kroku)
GLfloat xstep = 1.0f;
GLfloat ystep = 1.0f;
```

```
// Zmienne zachowujące rozmiary okna
GLfloat windowWidth;
GLfloat windowHeight;

static LPCTSTR lpszAppName = "GLRect";

// Deklaracja procedury okna
LRESULT CALLBACK WndProc(     HWND  hWnd,
                              UINT  message,
                              WPARAM      wParam,
                              LPARAM      lParam);

// Prototyp funkcji ustalającej format pikseli
void SetDCPixelFormat(HDC hDC);

void ChangeSize(GLsizei w, GLsizei h)
      {
      // Zabezpieczenie przed dzieleniem przez zero
      // Nie możesz utworzyć okna o zerowej szerokości
      if(h == 0)
            h = 1;

      // Ustaw viewport na całe okno
      glViewport(0, 0, w, h);

      // Zresetuj układ odniesienia przed modyfikacjami
      glLoadIdentity();


      // Fragment kodu zachowujący proporcje obrazu bez względu
      // na modyfikacje w rozmiarze okna
      if (w <= h)
            {
            windowHeight = 250.0f*h/w;
            windowWidth = 250.0f;
            }
    else
            {
            windowWidth = 250.0f*w/h;
            windowHeight = 250.0f;
            }

      // Ustal prostopadłościan widzenia
      glOrtho(0.0f, windowWidth, 0.0f, windowHeight, 1.0f, -1.0f);
      }


// Funkcja wywoływana przy obsłudze komunikatu WM_TIMER
void IdleFunction(void)
      {
      // Odwróć kierunek, kiedy obiekt dotknie krawędzi bocznych
      if(x1 > windowWidth-rsize || x1 < 0)
            xstep = -xstep;

      // Odwróć kierunek, kiedy obiekt dotknie krawędzi górnej i dolnej
      if(y1 > windowHeight-rsize || y1 < 0)
            ystep = -ystep;

      // Sprawdź krawędzie (W razie, gdy dokonano zmiany rozmiaru
      // okna i kwadrat jest poza obszarem zdefiniowanym jako
      // widoczny)
      if(x1 > windowWidth-rsize)
            x1 = windowWidth-rsize-1;
```

```
      if(y1 > windowHeight-rsize)
            y1 = windowHeight-rsize-1;

      // Przemieść kwadrat
      x1 += xstep;
      y1 += ystep;
      }

// Wywoływane podczas obsługi komunikatu WM_PAINT
void RenderScene(void)
      {
      // Ustal kolor tła
      glClearColor(0.0f, 0.0f, 1.0f, 1.0f);

      // Wyczyść okno bieżącym kolorem tła
      glClear(GL_COLOR_BUFFER_BIT);

      // Ustaw kolor rysowania na czerwony i rysuj kwadrat
      // w aktualnie ustalonej pozycji
      glColor3f(1.0f, 0.0f, 0.0f);
      glRectf(x1, y1, x1+rsize, y1+rsize);

      glFlush();
      }

// Ustal format pikseli dla danego kontekstu urządzenia
void SetDCPixelFormat(HDC hDC)
      {
      int nPixelFormat;

      static PIXELFORMATDESCRIPTOR pfd = {
            sizeof(PIXELFORMATDESCRIPTOR),// Rozmiar struktury
            1,                            // Wersja struktury
            PFD_DRAW_TO_WINDOW |          // Rusuj w oknie nie w itmapie
            PFD_SUPPORT_OPENGL |          // Wspomagaj wywołania funkcji
                                          // OpenGL
            PFD_DOUBLEBUFFER,             // Tryb podwójnego buforowania
            PFD_TYPE_RGBA,                // Tryb kolorów RGBA
            8,                            // 8-bitowy kolor
            0,0,0,0,0,0,                  // Nie używane do wybrania trybu
            0,0,                          // Nie używane do wybrania trybu
            0,0,0,0,0,                    // Nie używane do wybrania trybu
            16,                           // Rozmiar bufora głębokości
            0,                            // Nie używane do wybrania trybu
            0,                            // Nie używane do wybrania trybu
            PFD_MAIN_PLANE,               // Rysuj w głównym planie
            0,                            // Nie używane do wybrania trybu
            0,0,0 };                      // Nie używane do wybrania trybu

      // Wybierz format pikseli najlepiej odpowiadający opisanemu
      // w strukturze pdf
      nPixelFormat = ChoosePixelFormat(hDC, &pfd);

      // Ustaw format pikseli dla kontekstu urządzenia
      SetPixelFormat(hDC, nPixelFormat, &pfd);
      }

// Główna funkcja programu
int APIENTRY WinMain(   HINSTANCE   hInstance,
                        HINSTANCE   hPrevInstance,
                        LPSTR             lpCmdLine,
                        int               nCmdShow)
      {
      MSG         msg;
      WNDCLASS    wc;
```

```
	HWND		hWnd;


	wc.style			= CS_HREDRAW | CS_VREDRAW;
	wc.lpfnWndProc		= (WNDPROC) WndProc;
	wc.cbClsExtra		= 0;
	wc.cbWndExtra		= 0;
	wc.hInstance		= hInstance;
	wc.hIcon			= NULL;
	wc.hCursor			= LoadCursor(NULL, IDC_ARROW);

	// Dla OpenGL nie potrzeba pędzla tła okna
	wc.hbrBackground  = NULL;

	wc.lpszMenuName		= NULL;
	wc.lpszClassName  = lpszAppName;

	if(RegisterClass(&wc) == 0)
		return FALSE;

	hWnd = CreateWindow(
			lpszAppName,
			lpszAppName,

			// OpenGL wymaga WS_CLIPCHILDREN i WS_CLIPSIBLINGS
			WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN | WS_CLIPSIBLINGS,

			// Pozycja i rozmiar okna
			100, 100,
			250, 250,
			NULL,
			NULL,
			hInstance,
			NULL);

	if(hWnd == NULL)
		return FALSE;

	ShowWindow(hWnd,SW_SHOW);
	UpdateWindow(hWnd);

	while( GetMessage(&msg, NULL, 0, 0))
		{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
		}

	return msg.wParam;
	}

LRESULT CALLBACK WndProc(	HWND  hWnd,
					UINT  message,
					WPARAM	wParam,
					LPARAM	lParam)
	{
	static HGLRC hRC;		// Kontekst renderowania
	static HDC hDC;			// Kontekst urządzenia

	switch (message)
		{

		case WM_CREATE:
			// Zachowaj kontekst urządzenia
			hDC = GetDC(hWnd);
```

```
            // Wybierz format pikseli
            SetDCPixelFormat(hDC);

            // Utwórz kontekst renderowania i ustaw go jako
            // bieżący
            hRC = wglCreateContext(hDC);
            wglMakeCurrent(hDC, hRC);

            // Utwórz Timer odpalający co 1 milisekundę
            SetTimer(hWnd,101,1,NULL);
            break;


        case WM_DESTROY:
            // Zniszcz Timer
            KillTimer(hWnd,101);

            // Odłącz kontekst renderowania i zniszcz go
            wglMakeCurrent(hDC,NULL);
            wglDeleteContext(hRC);

            PostQuitMessage(0);
            break;

        case WM_SIZE:
            // Wywołaj funkcję modyfikującą obraz pod wpływem
            // zmiany rozmiarów okna
            ChangeSize(LOWORD(lParam), HIWORD(lParam));
            break;

        // Komunikat jest obsługiwany gdy nadchodzi "przerwanie"
        // od zegara. Wykonywana jest funkcja zmieniająca koordynaty
        // poruszającego się kwadratu, a następnie unieważniająca
        // okno, co wywoła obsługę komunikatu WM_PAINT
        case WM_TIMER:
            {
            IdleFunction();

            InvalidateRect(hWnd,NULL,FALSE);
            }
            break;

        case WM_PAINT:
            {
            // Wywołaj funkcję rysującą
            RenderScene();

            // wywołaj funkcję zamieniającą bufory
            SwapBuffers(hDC);

            // Zabezpieczenie przed niepotrzebnymi wywołaniami
            // komunikatu WM_PAINT
            ValidateRect(hWnd,NULL);
            }
            break;

    default:
        return (DefWindowProc(hWnd, message, wParam, lParam));

    }

return (0L);
  }
```

Zaprezentowany program ,podobnie jak Bounce2, tworzy okno w systemie, a następnie realizuje animację poruszającego się kwadratu, który odbija się od krawędzi okna. Poszczególne elementy systemu i biblioteka OpenGL współpracują w programie ze sobą w następujący sposób:

- Obsługa komunikatu WM_CREATE polega na zainicjalizowaniu formatu pikseli, stworzeniu kontekstu renderowania i ustawieniu go jako bieżący. W obsłudze tego komunikatu utworzony zostaje również zegar poprzez wywołanie funkcji:

  ```
  SetTimer(hWnd,101,1,NULL);
  ```

  Zegar w systemie Windows jest urządzeniem wejścia, które cyklicznie informuje aplikację o upływie określonego interwału czasu. Co określony czas (padany jako parametr wywołania funkcji SetTimer) zegar generuje komunikat WM_TIMER. W prezentowanym programie zegar służy do odmierzania czasu pomiędzy wyświetlaniem kolejnych klatek animacji.

- Obsługa komunikatu WM_DESTROY polega na zakończeniu działania zegara (funkcja KillTimer()), a następnie na odłączeniu i zniszczeniu kontekstu renderowania oraz wysłaniu komunikatu do systemu pozwalającego na zakończenie pracy aplikacji (PostQuitMessage()).

- Komunikat WM_SIZE obsługiwany jest w programie przez identyczną jak w programie Bounce2 funkcję Change Size().

- Jak już wspomniano, komunikat WM_TIMER jest wysyłany przez uaktywniony na początku działania programu zegar. Obsługa komunikatu WM_TIMER polega na wywołaniu funkcji modyfikującej położenie poruszającego się kwadratu IdleFunction(). Zawartość funkcji jest podobna do odpowiadającej funkcji w programie Bounce2. Nie wywołuje ona tylko funkcji RenderScene(). Po zakończeniu działania funkcji IdleFunction() wywołana zostaje funkcja InvalidateRect() co automatycznie powoduje wysłanie do systemu komunikatu WM_PAINT.

- Obsługa komunikatu WM_PAINT składa się z dwu etapów. Pierwszym z nich jest wywołanie funkcji RenderScene() o kodzie identycznym jak w programie Bounce2. Drugim etapem obsługi komunikatu WM_PAINT jest wywołanie funkcji SwapBuffers(). Funkcja ta jest odpowiednikiem auxSwapBuffers() i realizuje wyświetlenie zawartości załadowanej do bufora klatki animacji (zawartość dotychczas niewidocznego bufora pojawia się na ekranie, a bufor do tej pory wyświetlany przygotowany jest na przesłanie do niego następnej klatki animacji).

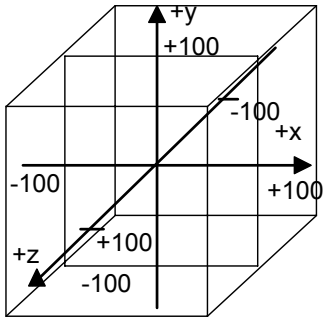
Omówiony powyżej program umieszczony jest w katalogu o nazwie *Glrect1* na dołączonej do opracowania dyskietce. Stanowić on może szkielet dla dowolnych aplikacji łączących w sobie możliwości trójwymiarowej biblioteki OpenGL i graficznego interfejsu Microsoft Windows.

3. Obiekty trójwymiarowe: linie, punkty, wielokąty

Podstawowymi elementami składowymi sceny tworzonej przy pomocy biblioteki OpenGL są tak zwane prymitywy graficzne - najprostsze elementy graficzne, z których można tworzyć dowolnie złożone obiekty. Wszystkie prymitywy są jedno lub

dwuwymiarowymi obiektami. Najprostszymi z nich są punkty, najbardziej skomplikowanymi wielokąty.

Umieszczone poniżej przykłady stanowić będą prezentacje podstawowych definicji prymitywów graficznych OpenGL. W opracowaniu znajdują się odwołania do programów umieszczonych na załączonej do opracowania dyskietce. Podawana nazwa programu jest równocześnie katalogiem zawierającym pliki projektowe i źródłowe aplikacji. Wszystkie obiekty sytuowane będą w zdefiniowanym prostopadłościanie widzenia o rozmiarach 100 × 100 × 100 jednostek (rys 3.1). Zagadnienia dotyczące definiowania ostrosłupów i prostopadłościanów widzenia będą treścią dalszych materiałów pomocniczych do przedmiotu Grafika Komputerowa i Animacja, stąd zawarte modyfikacje w funkcji ChangeSize() nie będą szczegółowo omówione. Tworzone przykładowe prymitywy graficzne można w dostarczonych programach "obracać" w przestrzeni przy pomocy klawiszy strzałek na klawiaturze komputera.



Rys 3.1 Prostopadłościan widzenia 100 × 100 × 100

Modyfikacje w funkcji ChangeSize() prezentuje wydruk poniżej:

```
void ChangeSize(GLsizei w, GLsizei h)
      {
      GLfloat nRange = 100.0f;

      if(h == 0)
            h = 1;

      glViewport(0, 0, w, h);

      // Wyzeruj stos macierzy projekcji
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();

      // Oblicz prostopadłościan widzenia
      if (w <= h)
      glOrtho (-nRange, nRange, -nRange*h/w, nRange*h/w, -nRange, nRange);
      else
      glOrtho (-nRange*w/h, nRange*w/h, -nRange, nRange, -nRange, nRange);

      // Wyzeruj stos macierzy modelowania / obserwacji
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      }
```

3.1 Punkty

Podstawową, najczęściej wykorzystywaną funkcją standardu OpenGL jest funkcja definiująca wierzchołek: **glVertex3f()**. Podane poniżej wywołanie funkcji definiuje wierzchołek w przestrzeni i umieszcza go w punkcie o współrzędnych (50.0,50.0,0.0):

```
glVertex3f(50.0f,50.0f,0.0F)
```

Ale dopiero podanie dodatkowych informacji (Czy wierzchołek ma być samodzielnym punktem wyświetlanym na ekranie, czy ma też wyznaczać wielokąt lub odcinek) pozwala na wyświetlenie prymitywu na ekranie. Pojedynczy prymityw graficzny OpenGL tworzy się przez podanie listy wierzchołków zawartych pomiędzy komendami **glBegin()** a **glEnd()**. Parametr wywołania komendy glBegin() ustala rodzaj zależności pomiędzy podanymi wierzchołkami. Najprostszym prymitywem jest punkt. Fragment programu:

```
glBegin(GL_POINTS);                 // Wybierz rodzaj prymitywu : punkty
      glVertex3f(0.0f,0.0f,0.0f);   // Podaj położenie punktu
      glVertex3f(50.0f,50.0f,50.0f);// Podaj położenie następnego punktu
glEnd();                            // Zakończ rysowanie punktów
```

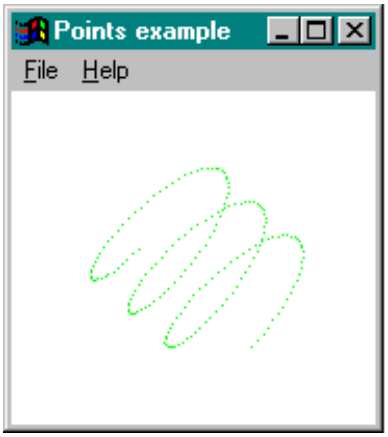
poprzez podanie parametru **GL_PIONTS** powoduje, że podane wierzchołki interpretowane są jako pojedyncze punkty. Pomiędzy komendami glBegin() a glEnd() można umieścić dowolną liczbę prymitywów graficznych, oczywiście gdy chcemy żeby był to zbór prymitywów tego samego rodzaju.

Podany poniżej fragment kodu programu POINTS (umieszczonego na dyskietce) tworzy spiralę złożoną z pojedynczych punktów:

```
void RenderScene(void)
      {
      GLfloat x,y,z,angle;// Zmienne pomocnicze do przechowywania położeń
                          // punktów

      // Wyczyść okno
      glClear(GL_COLOR_BUFFER_BIT);

      // Zachowaj stan macierzy i wykonaj rotację
      glPushMatrix();
      glRotatef(xRot, 1.0f, 0.0f, 0.0f);
      glRotatef(yRot, 0.0f, 1.0f, 0.0f);

      // Definiuj prymityw
      glBegin(GL_POINTS);

      z = -50.0f;
      for(angle = 0.0f; angle <= (2.0f*GL_PI)*3.0f; angle += 0.1f)
            {
            x = 50.0f*sin(angle);
            y = 50.0f*cos(angle);

            // Podaj położenie punktu i przesuń współrzędną z punktu
            glVertex3f(x, y, z);
            z += 0.5f;
            }

      // Zakończ rysowanie punktu
      glEnd();

      // Przywróć transformacje
      glPopMatrix();

      glFlush();
```

```
}
```

Istotne znaczenie ma fragment programu zawarty pomiędzy komendami glBegin() i glEnd(). Funkcje wywoływane poza definicją prymitywów służą miedzy innymi do mazania tła okna oraz do umożliwienia dokonywania transformacji przestrzennych narysowanego obiektu i będą omówione w dalszych opracowaniach. Łatwo zauważyć, że pomiędzy komendami glBegin() a glEnd() oprócz zastrzeżonego zbioru komend OpenGL można umieszczać dowolne fragmenty kodu w języku C/C++.
Przykładowe okno programu prezentuje rysunek 3.1.1 Klawisze strzałek na klawiatura umożliwiają obracanie obiektu w przestrzeni.



Rys 3.1.1 Okno programu POINTS

Domyślnym rozmiarem punktu jest jeden piksel. Funkcja **glPointSize(GLfloat size)** umożliwia zmianę rozmiaru punktu. Zakres liczb opisujących dostępne rozmiary punktu zależy od implementacji biblioteki OpenGL, a można go uzyskać przy pomocy fragmentu kodu:

```
GLfloat sizes[2];  // do zachowania zakresu dostępnych rozmiarów punktów
GLfloat step;      //  do  zachowania  minimalnego  dostępnego  "skoku"  w
rozmiarze

// Pobierz dostępne rozmiary punków i "skok"
glGetFloatv(GL_POINT_SIZE_RANGE,sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY,&step);
```

Program POINTSZ wykorzystuje możliwość definiowania rozmiarów punktów do wyrysowania spirali składającej się z punków o różnej wielkości. Zaskoczeniem może być, że "powiększone" punkty mają kształt kwadratów. Wynika to ze specyfikacji standardu OpenGL. Oczywiście możliwe jest zmuszenie programu aby wyświetlał okrągłe punkty poprzez wywołanie funkcji:

```
glEnable(GL_PIONT_SMOOTH);
```

Jak już wspomniano, często traktuje się OpenGL jako swego rodzaju system operacyjny, bądź środowisko systemowe służące do tworzenia trójwymiarowej grafiki komputerowej. Jako "system operacyjny" OpenGL w momencie inicjalizacji programu powołuje do życia zbiór zmiennych systemowych przechowujących informacje o stanie programu. Funkcja **glEnable()** oraz pokrewna jej **glDisable()** służą do modyfikacji tych zmiennych systemowych. Zmienne systemowe decydują o szeregu właściwościach tworzonej

sceny i w dalszej części opracowania oraz w kolejnych materiałach pomocniczych z przedmiotu Grafika Komputerowa i Animacja będą one sukcesywnie prezentowane.

3.2 Linie

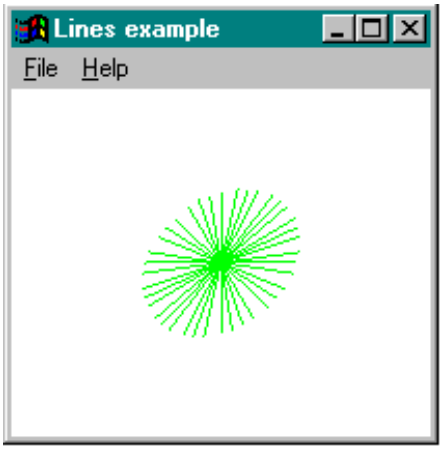
Dwa punkty w przestrzeni definiują odcinek. Fragment kodu:

```
glBegin(GL_LINES);
      glVertex3f(0.0f,0.0f,0.0f);
      glVertex3f(50.0f,50.0f,50.0f);
glEnd();
```

rysuje odcinek wyznaczony przez podane dwa wierzchołki. Należy pamiętać, że przy definiowaniu tego rodzaju prymitywu pod uwagę będzie brana parzysta liczba wierzchołków. W przypadku podania nieparzystej liczby punków, ostatni z nich będzie ignorowany. Program LINES tworzy pęk odcinków łączących naprzeciwległe punkty okręgu. Poniższy fragment kodu tworzy wspomniany pęk odcinków:

```
      glBegin(GL_LINES);

      z = 0.0f;
      for(angle = 0.0f; angle <= GL_PI*3.0f; angle += 0.5f)
            {
            // Górna połowa okręgu
            x = 50.0f*sin(angle);
            y = 50.0f*cos(angle);
            glVertex3f(x, y, z);

            // Dolna połowa okręgu
            x = 50.0f*sin(angle+3.1415f);
            y = 50.0f*cos(angle+3.1415f);
            glVertex3f(x, y, z);
            }

      glEnd();
```

Rysunek 3.2.1 prezentuje przykładowe okno programu.



Rys 3.2.1 Okno programu LINES

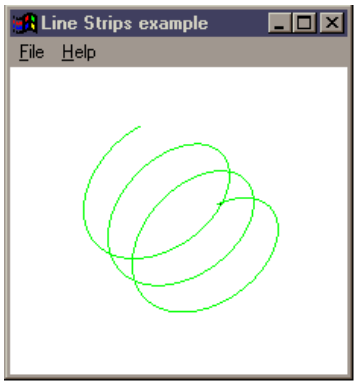
Następne dwa prymitywy OpenGL pozwalają na budowę łamanej lub zamkniętej łamanej poprzez podanie wierzchołków przez które ma przechodzić. Jeśli jako parametr wywołania komendy glBegin() wstawiony zostanie **GL_LINE_STRIP**, to podawane kolejne wierzchołki łączone będą odcinkami prostymi. Kod:

```
glBegin(GL_LINE_STRIP);
      glVertex3f(0.0f, 0.0f, 0.0f);       //V0
      glVertex3f(50.0f, 50.0f, 0.0f);     //V1
      glVertex3f(50.0f, 100.0f, 0.0f);    //V2
glEnd();
```

pozwala na wyrysowanie dwu linii opisanych przy pomocy trzech wierzchołków. Jeśli obiekt tworzony jest jako prymityw typu **GL_LINE_LOOP**, to podobnie jak w przypadku GL_LINE_STRIP kolejne wierzchołki wyznaczają kolejne odcinki łamanej, przy czym wierzchołek ostani i pierwszy są automatycznie łączone ze sobą, tak, że powstały obiekt jest łamaną zamkniętą. Parametr GL_LINE_LOOP służyć może do najprostszego definiowania figur zamkniętych w standardzie OpenGL. Zaprezentowany sposób tworzenia łamanych służyć może także do tworzenia przybliżeń krzywych. Najbardziej efektywnym sposobem tworzenia krzywych w bibliotece OpenGL jest wyznaczenie serii punktów, przez które krzywa ma przebiegać, a następnie wykorzystanie prymitywu GL_LINE_STRIP do automatycznego połączenia wierzchołków. Poniższy fragment kodu prezentuje sposób stworzenia ciągłej spirali zbudowanej na bazie wierzchołków definiowanych w programie POINTS:

```
glBegin(GL_LINE_STRIP);

z = -50.0f;
for(angle = 0.0f; angle <= (2.0f*3.1415f)*3.0f; angle += 0.1f)
      {
      x = 50.0f*sin(angle);
      y = 50.0f*cos(angle);

      // Specify the point and move the Z value up a little
      glVertex3f(x, y, z);
      z += 0.5f;
      }

// Done drawing points
glEnd();
```

Rysunek 3.2.1 prezentuje okno programu LSTRIPS wykorzystującego powyższy kod.



Rys. 3.2.2 Okno programu LSTRIPS.

Podobnie jak można definiować wielkość punktów, możliwe jest w bibliotece OpenGL zdefiniowanie grubości linii. Służy do tego funkcja **glLineWidth(GLfloat width)**. Paramet wywołania funkcji jest przybliżoną grubością linii w pikselach. Podobnie jak w przypadku rozmiaru punktów, nie wszystkie grubości linii są rozpoznawane przez funkcję. Podany

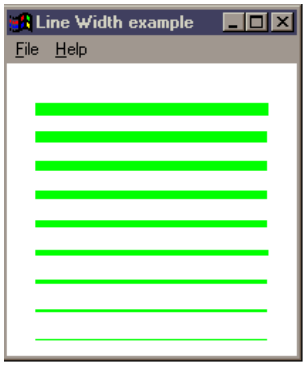
poniżej kod pozwala określić zakres parametru wywołania funkcji glLineWidth() dla danej implementacji biblioteki OpenGL:

```
GLfloat sizes[2];  // do zachowania zakresu dostępnych grubości linii
GLfloat step;      //  do  zachowania  minimalnego  dostępnego  "skoku"  w
grubości

// Pobierz dostępne rozmiary punków i "skok"
glGetFloatv(GL_LINE_WIDTH_RANGE,sizes);
glGetFloatv(GL_LINE_WIDTH_GRANULARITY,&step);
```

Program LINESW wyrysowuje na ekranie zestaw równoległych linii o narastającej grubości. Poniższy fragment kodu programu LINESW prezentuje wykorzystany sposób rysowania i definiowania grubości linii:

```
void RenderScene(void)
      {
      GLfloat y;                          //   Storeage   for   varying   Y
coordinate
      GLfloat fSizes[2];                  // Line width range metrics
      GLfloat fCurrSize;                  // Save current size

      // ...

      // Get line size metrics and save the smallest value
      glGetFloatv(GL_LINE_WIDTH_RANGE,fSizes);
      fCurrSize = fSizes[0];

      // Step up Y axis 20 units at a time
      for(y = -90.0f; y < 90.0f; y += 20.0f)
            {
            // Set the line width
            glLineWidth(fCurrSize);

            // Draw the line
            glBegin(GL_LINES);
                  glVertex2f(-80.0f, y);
                  glVertex2f(80.0f, y);
            glEnd();

            // Increase the line width
            fCurrSize += 1.0f;
            }

      // ...
      }
```

Rysunek 3.2.3 prezentuje przykładowe okno programu LINESW.



Rys 3.2.3 Okno programu LINESW.

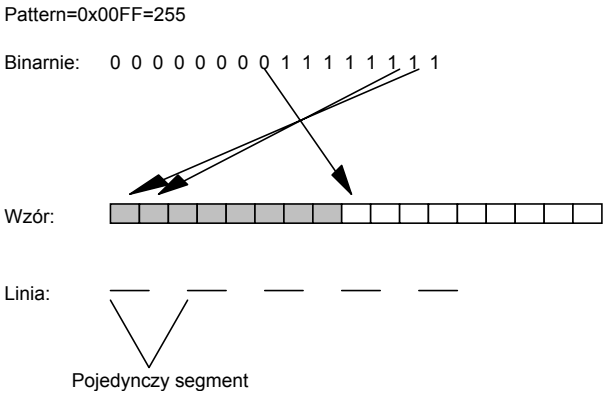
Oprócz nadawania liniom grubości, standard OpenGL umożliwia utworzenie linii o zadanym wzorze (np. kreskowana, kropkowana itp.). Aby możliwe było nadawanie "wzorów" liniom uaktywniona musi być odpowiednia zmienna systemowa OpenGL:

```
glEnable(GL_LINE_STIPPLE);
```

Definicją wzoru linii steruje funkcja o prototypie:

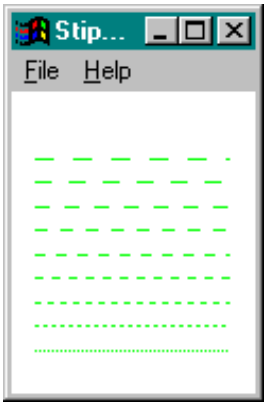
void glLineStipple(GLint factor, GLushort pattern)

Parmetr *pattern* jest 16 bitową wartością, która definiuje wzór rysowanej linii. Każdy bit wartości odpowiada fragmentowi linii, który jest wyświetlany lub nie. Domyślnie każdy bit odpowiada pojedynczemu pikselowi na ekranie, ale parametr *factor* pełni rolę mnożnika określającego ile razy każdy bit z parametru *pattern* ma zostać powielony. Przykładowo ustalenie parametru *factor* na 5 spowoduje, że każdy bit w parametrze *pattrern* reprezentował będzie pięć pikseli które mają być albo zaświecona albo zgaszone. Dodatkowo bit zerowy (LSB) parametru *pattern* jest używany jako pierwszy do specyfikowania wzoru (Dokładniej wyjaśnia to rys 3.2.4).



Rys 3.2.4 Sposób odwzorowania wzoru linii (parametru *pattern*) w standardzie OpenGL

Dołączony do opracowania program LSTIPPLE (rys 3.2.5) wykorzystuje możliwość tworzenia linii o określonym wzorze do wyświetlenia zestawu linii o tym samym wzorze ale o zmieniającym się parametrze *factor* funkcji glLineStipple().



Rys 3.2.5 Okno programu LSTIPPLE.

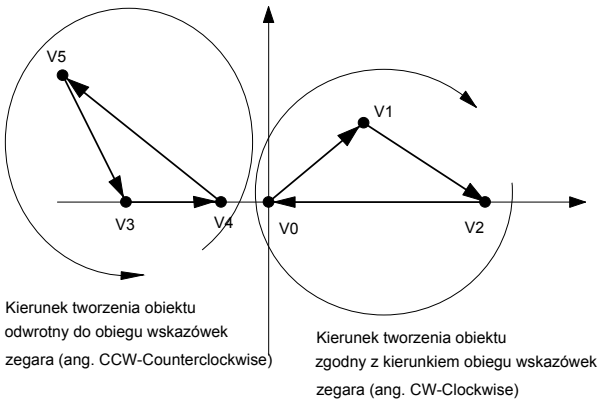
3.3 Trójkąty

Jakkolwiek w standardzie OpenGL możliwe jest tworzenie figur zamkniętych z zestawu odcinków to żaden z tak stworzonych obiektów nie można wypełnić żadnym kolorem - pozostaje on zawsze zbiorem odcinków. Najprostszą figurą zamkniętą (taką którą można zamalować kolorem, uczynić ją półprzeźroczystą itp.) w bibliotece OpenGL jest trójkąt. Do tworzenia trójkątów służy prymityw **GL_TRIANGLES**. Pobiera on trzy kolejne wierzchołki z podanej listy i łączy w jeden obiekt. Przykładowy fragment kodu:

```
glBegin(GL_TRIANGLES);
      glVertex2f(0.0f, 0.0f);            //V0
      glVertex2f(25.0f, 25.0f);          //V1
      glVertex2f(50.0f, 0.0f);           //V2

      glVertex2f(-50.0f, 0.0f);          //V3
      glVertex2f(-75.0f, 50.0f);         //V4
      glVertex2f(-25.0f, 0.0f);          //V4

glEnd();
```

tworzy dwa trójkąty używając sześciu wierzchołków. Należy pamiętać, że trójkąt wypełniony zostanie aktualnie ustalonym kolorem rysowania. Jeśli przed rysowaniem obiektu nie ustalono żadnego koloru rysowania, to nie można przewidzieć wyniku działania programu, bo w OpenGL nie ma domyślnego koloru rysowania. Trójkąty są najbardziej popularnymi prymitywami służącymi do tworzenia trójwymiarowych obiektów w OpenGL. Każdy wielokąt skomponować można ze zbioru odpowiednio dobranych trójkątów. Trójkąty są najprostszymi figurami do zdefiniowania w trójwymiarowej przestrzeni (wierzchołki trójkąta zawsze mieszczą się w jednej płaszczyźnie). Większość akceleratorów sprzętowych OpenGL posiada wbudowane sprzętowe funkcje wspomagające wyświetlanie trójkątów.

Ważną cechą w standardzie OpenGL jest kierunek rysowania prymitywów zamkniętych. Ilustruje to rysunek 3.3.1.



Rys 3.3.1 Kierunki rysowania prymitywów na płaszczyźnie

Kierunek rysowania figur zamkniętych pozwala rozpoznawać "przednią" i "tylnią" powierzchnię obiektu. Domyślnie w standardzie OpenGL przednią (frontową) stroną płaskiego prymitywu jest ta strona, w której obieg punktów podczas rysowania jest odwrotny do obiegu wskazówek zegara (CCW). Na rysunku 3.3.1 oznacza to, że trójkąt (V0,V1,V2) zwrócony jest do nas tyłem, a trójkąt (V3,V4,V5) - przodem. Wyznaczenie stron jest o tyle ważne, że w OpenGL każdej ze stron wielokąta można nadać inne właściwości (kolor,

przeźroczystość itp.). W trakcje konstruowania obiektów graficznych można w każdej chwili nakazać OpenGL, aby przednią stroną stała się ta która była domyślnie tylną. Odpowiedzialna za to jest funkcja **glFrontFace()**. Wywołanie funkcji:

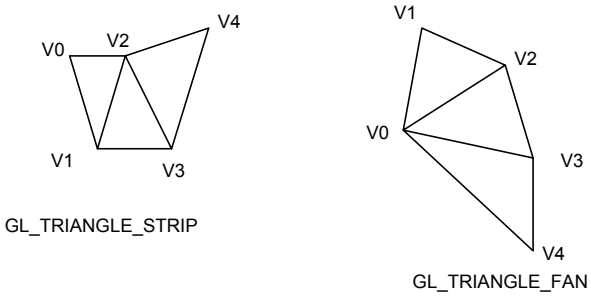
```
glFrontFace(GL_CW);
```

powoduje odwrócenie przyjętego domyślnie porządku wyznaczania frontowych ścian prymitywów, a wywołanie:

```
glFrontFace(GL_CCW);
```

"przywraca" domyślny porządek. Umiejętne manipulowanie funkcją glFrontFace() znacznie ułatwia tworzenie zamkniętych obiektów przy pomocy bardziej zaawansowanych prymitywów graficznych, w których "kierunek obiegu punktów" ustalany jest z góry.

Do stworzenia wielu powierzchni i kształtów doskonale nadaje się ciąg połączonych w odpowiedni sposób ze sobą trójkątów. Stąd OpenGL przewiduje dwa dodatkowe prymitywy pozwalające budować tego typu obiekty: **GL_TRIANGLE_STRIP** oraz **GL_TRIANGLE_FAN**. Pierwszy z nich pozwala na tworzenie swego rodzaju pasu z połączonych ze sobą trójkątów, drugi zaś automatycznie kreuje "wachlarz" złożony z trójkątów (rys 3.3.2).

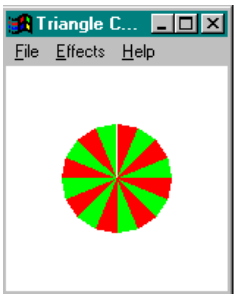


Rys. 3.3.2 Interpretacja prymitywów GL_TRIANGLE_STRIP i GL_TRIANGLE_FAN

Należy zauważyć, że prymityw GL_TRIANGLE_FAN tworzony jest automatycznie w tki sposób, że zwrócony jest do nas tylnią powierzchnią.

3.4 Tworzenie zamkniętych brył

Zaprezentowany poniżej program TRIANGLE (umieszczony na załączonej dyskietce) pozwala na zapoznanie się z podstawowymi aspektami tworzenia zamkniętych brył złożonych z trójkątów i innych wielokątów w standardzie OpenGL. Program przy pomocy dwu "wachlarzy" trójkątów tworzy w przestrzeni przybliżenie stożka. Dla odróżnienia kolejne trójkąty prymitywów rysowane są na przemian kolorem zielonym lub czerwonym. Rysunek 3.4.1 przedstawia początkowy wygląd programu TRIANGLE. Ponieważ oś stożka pokrywa się z osią "z" układu współrzędnych na ekranie widoczne jest tylko "koło" złożone z "wachlarza" trójkątów.



Rys 3.4.1 Początkowy wygląd okna programu TRIANGLE

Poniżej znajduje się wydruk kodów funkcji SetupRC() i RenderScene() programu TRIANGLE, które zawierają istotne modyfikacje w stosunku do dotychczasowo prezentowanych:

```
// This function does any needed initialization on the rendering
// context.
void SetupRC()
      {
      // Black background
      glClearColor(0.0f, 0.0f, 0.0f, 1.0f );

      // Set drawing color to green
      glColor3f(0.0f, 1.0f, 0.0f);

      // Set color shading model to flat
      glShadeModel(GL_FLAT);

      // Clock wise wound polygons are front facing, this is reversed
      // because we are using triangle fans
      glFrontFace(GL_CW);
      }

// Called to draw scene
void RenderScene(void)
      {
      GLfloat x,y,angle;  // Storage for coordinates and angles
      int iPivot = 1;         // Used to flag alternating colors

      // Clear the window and the depth buffer
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

      // Turn culling on if flag is set
      if(bCull)
            glEnable(GL_CULL_FACE);
      else
            glDisable(GL_CULL_FACE);

      // Enable depth testing if flag is set
      if(bDepth)
            glEnable(GL_DEPTH_TEST);
      else
            glDisable(GL_DEPTH_TEST);

      // Draw back side as a polygon only, if flag is set
      if(bOutline)
            glPolygonMode(GL_BACK,GL_LINE);
      else
            glPolygonMode(GL_BACK,GL_FILL);


      // Save matrix state and do the rotation
      glPushMatrix();
```

```
glRotatef(xRot, 1.0f, 0.0f, 0.0f);
glRotatef(yRot, 0.0f, 1.0f, 0.0f);


// Begin a triangle fan
glBegin(GL_TRIANGLE_FAN);

// Pinnacle of cone is shared vertex for fan, moved up Z axis
// to produce a cone instead of a circle
glVertex3f(0.0f, 0.0f, 75.0f);

// Loop around in a circle and specify even points along the circle
// as the vertices of the triangle fan
for(angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI/8.0f))
      {
      // Calculate x and y position of the next vertex
      x = 50.0f*sin(angle);
      y = 50.0f*cos(angle);

      // Alternate color between red and green
      if((iPivot %2) == 0)
            glColor3f(0.0f, 1.0f, 0.0f);
      else
            glColor3f(1.0f, 0.0f, 0.0f);

      // Increment pivot to change color next time
      iPivot++;

      // Specify the next vertex for the triangle fan
      glVertex2f(x, y);
      }

// Done drawing fan for cone
glEnd();


// Begin a new triangle fan to cover the bottom
glBegin(GL_TRIANGLE_FAN);

// Center of fan is at the origin
glVertex2f(0.0f, 0.0f);
for(angle = 0.0f; angle < (2.0f*GL_PI); angle += (GL_PI/8.0f))
      {
      // Calculate x and y position of the next vertex
      x = 50.0f*sin(angle);
      y = 50.0f*cos(angle);

      // Alternate color between red and green
      if((iPivot %2) == 0)
            glColor3f(0.0f, 1.0f, 0.0f);
      else
            glColor3f(1.0f, 0.0f, 0.0f);

      // Increment pivot to change color next time
      iPivot++;

      // Specify the next vertex for the triangle fan
      glVertex2f(x, y);
      }

// Done drawing the fan that covers the bottom
glEnd();

// Restore transformations
glPopMatrix();
```

```
// Flush drawing commands
glFlush();
}
```

Na uwagę zasługuje opcja menu *Effects*, która pozwala na zmianę pewnych parametrów decydujących o sposobie wyświetlania obiektów graficznych na scenie.

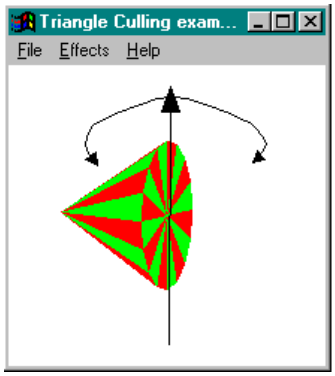
W standardzie OpenGL jeśli definiowany jest kolor to dotyczy on wierzchołków prymitywów. Oznacza to, że w trakcie rysowania prymitywu możemy zmieniać kolor jego wierzchołków. Do ustalenia w jaki sposób będą interpretowane kolory wierzchołków prymitywu służy funkcja **glShadeModel()**. Jeśli chcemy aby kolor prymitywu był taki jak kolor jego ostatniego wierzchołka, to przed rysowaniem prymitywu należy wywołać funkcję w następujący sposób:

```
glShadeModel(GL_FLAT);
```

, natomiast jeśli chcemy aby kolor każdego wierzchołka prymitywu mógł być określany indywidualnie (powoduje to, że w drodze od jednego wierzchołka do drugiego OpenGL stara się w sposób ciągły interpolować kolor) to funkcja wywołana powinna być w sposób:

```
glShadeModel(GL_SMOOTH);.
```

Jeśli próbowalibyśmy używać klawiszy strzałek do poruszania narysowanego stożka, to okazuje się, że nie jest on w stanie obracać się wokół osi równoległej do jego podstawy. Zamiast tego sprawia on wrażenie, jakby odbywał ruch podobny do wahadła jak na rysunku 3.4.2.



Rys 3.4.2 Początkowy efekt obrotu stożka w programie TRIANGLE

Efekt taki następuje dlatego, że podstawa stożka jest wyświetlana zawsze po narysowaniu jego tworzącej i zasłania poprzednio narysowany prymityw. Aby zapobiec tego typu problemom, należy uaktywnić w OpenGL tak zwany z-bufor. Mechanizm działania tego bufora jest bardzo prosty. Każdy wyrysowany na ekranie piksel otrzymuje dodatkową wartość określającą jego odległość od kamery. Jeśli wartość bieżąco rysowanego piksel jest mniejsza od wartości tego samego piksela już wcześniej wyświetlanego, to piksel pojawia się na ekranie. W przeciwnym wypadku jest on w myśl OpenGL zasłonięty i nie jest rysowany. Aby uaktywnić obliczanie głębokości położenia pikseli należących do figur w OpenGL należy wywołać funkcję:

```
glEnable(GL_DEPTH_TEST);
```

Uaktywnianie i wyłączanie bufora głębokości (z-bufora) wykonywane jest w funkcji RenderScene():

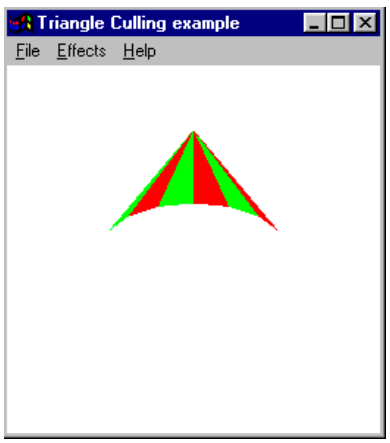
```
if(bDepth)
      glEnable(GL_DEPTH_TEST);
else
      glDisable(GL_DEPTH_TEST);
```

W zależności od stanu zmiennej bDepth modyfikowanej pod wpływem wybrania opcji *Depth Test* z menu *Effects* w programie TRIANGLE włączane lub wyłączone jest obliczanie bufora głębokości. Włączenie bufora głębokości powoduje, że trójwymiarowy obiekt może być prawidłowo obracany i obserwowany ze wszystkich stron.

Pomimo, że na ekranie uzyskujemy zadowalający efekt, to program zużywa znaczą ilość czasu na wyznaczanie widocznych i niewidocznych pikseli. Czasami istnieje taka sytuacja, że jesteśmy pewni, że pewna powierzchnia nigdy nie będzie na ekranie wyświetlana. Przykładowo, jeśli animowaną figurą jest bryła zamknięta, to na ekranie nigdy nie będą wyświetlane "wewnętrzne" powierzchnie prymitywów tworzących obiekt (Co rozumiane jest w myśl standardu OpenGL przez wewnętrzną-tylną i zewnętrzną-frontową powierzchnię wielokąta omówione zostało w punkcie 3.3 opracowania). W OpenGL istnieje możliwość wyłączenia wyświetlania (a więc eliminację konieczności przeprowadzania dodatkowych obliczeń) wybranych powierzchni prymitywów graficznych. W programie TRIANGLE włączanie i wyłączanie wyświetlania wewnętrznych powierzchni prymitywów odbywa się w liniach:

```
glFrontFace(GL_CW);

// ...

if(bCull)
      glEnable(GL_CULL_FACE);
else
      glDisable(GL_CULL_FACE);
```

W pierwszym kroku przy pomocy funkcji glFrontFace() ustalone jest w programie, które z powierzchni prymitywów uznawane są za przednie. Następnie, w zależności od stanu zmiennej bCull (zmienianej pod wpływem wybierania opcji menu *Effects* → *Cull Back*), włączane lub wyłączane jest wyświetlanie wewnętrznych powierzchni prymitywów. W oknie programu po wyłączeniu wyświetlania wewnętrznych powierzchni prymitywów widoczna jest tylko zewnętrzna powierzchnia tworzącej stożka (rys. 3.4.3).



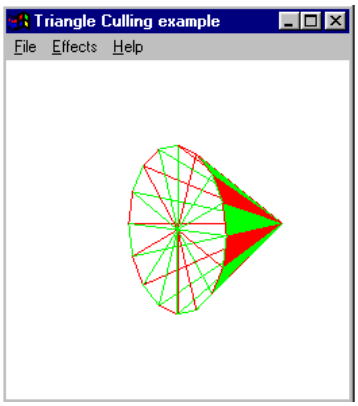
Rys. 3.4.3 Okno programu TRIANGLE

po wyłączeniu wyświetlania wewnętrznych powierzchni prymitywów.

Wielokąty OpenGL nie muszą być wypełniane kolorami. Domyślnie rysowane są one jako figury wypełnione, ale możliwe jest wyświetlanie ich jako zbiry krawędzi (w postaci linii) lub jako zbiór punktów. Służy do tego funkcja **glPolygonMode()**. Funkcja umożliwia zdefiniowanie w jaki sposób wyświetlana będzie dana strona wielokąta (przednia lub tylna). Można przykładowo zdefiniować, że przednia strona prymitywu będzie wypełniona, podczas gdy strona tylnia będzie widoczna tylko jako zestaw wierzchołków. Fragment kodu programu TRIANGLES:

```
if(bOutline)
      glPolygonMode(GL_BACK,GL_LINE);
else
      glPolygonMode(GL_BACK,GL_FILL);
```

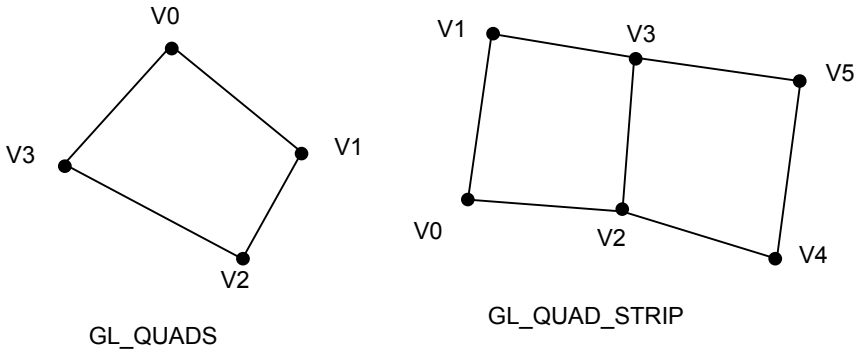
podaje, że w zależności od stanu zmiennej bOutline (ustawianej pod wpływem wybrania opcji *Effects→Outline Back*) wewnętrzne (tylnie) strony prymitywów będą albo wypełnione, albo tworzyć będą tylko siatkę złożoną z krawędzi (rys. 3.4.4).



Rys 3.4.4 Efekt wykorzystania funkcji glPolygonMode() w programie TRIANGLES

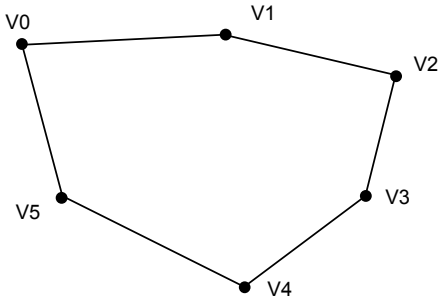
3.5 Pozostałe prymitywy - czworokąty, wielokąty.

OpenGL pozwala na tworzenie bardziej złożonych prymitywów niż trójkąty. Prymityw **GL_QUADS** pobiera z listy wierzchołków cztery kolejne i tworzy czworokąt. Podobnie jak w przypadku trójkątów istnieje możliwość utworzenia prymitywu złożonego z ciągu przylegających do siebie czworokątów. Powołuje się go do życia przy pomocy identyfikatora **GL_QUAD_STRIP**. Rysunek 3.5.1 prezentuje przykładowe prymitywy powołane przy pomocy identyfikatorów GL_QUADS i GL_QUAD_STRIP.



Rys 3.5.1 Przykłady definicji czworokątnych prymitywów w OpenGL

Prymityw **GL_POLYGON** służy do tworzenia prymitywów graficznych w postaci wielokąta o dowolnej ilości wierzchołków (rys. 3.5.2).



Rys 3.5.3 Przykład prymitywu GL_POLYGON

Podczas tworzenia obiektów składających się z wielu wielokątów należy pamiętać o dwu podstawowych zasadach. Po pierwsze **każdy z wielokątów musi być tak zdefiniowany, aby wszystkie jego wierzchołki leżały w jednej płaszczyźnie**. Po drugie zaś **zdefiniowane wielokąty nie mogą być wielokątami wklęsłymi**.

Bibliografia:

[1] R. S. Wright Jr., M. Sweet, *OpenGL Superbible*, The Waite Group Press, 1996
[2] *OpenGL Programming Guide*, Addison-Wessley, 1993
[3] E. Angel, *Interactive Computer Graphics*, Addison-Wesley, 1997
[4] R. Fosner, *OpenGL Programming for Windows and Windows NT*, Addison-Wesley, 1997
[5] R. Leniowski, *Wykłady z przedmiotu Grafika Komputerowa i Animacja*
[6] *Guide to OpenGL® on Windows® From Silicon Graphics®*, 1997