Тема 2. СРЕДСТВА ЗА СЪЗДАВАНЕ НА ГС

Методи и средства за създаване на ГС. Използване на средите C++ и Visual C++ за създаване на графични приложения. Базов синтаксис. Библиотеки Примери в двете среди.

2.1 ГРАФИЧНА БИБЛИОТЕКА В СРЕДАТА НА C++
2.2. ГРАФИЧНА C++ БИБЛИОТЕКА - GRAPHICS.H
2.3 ПРОГРАМИРАНЕ НА ГРАФИЧНИ ПРИЛОЖЕНИЯ В VISUAL C
2. 4 МАТЕМАТИЧЕСКИ АПАРАТ, ИЗПОЛЗВАН ОТ АЛГОРИТМИТЕ В КОМПЮТЪРНАТА ГРАФИКА

упр2. Създаване на графични приложения с C++ в средата на Win BGI

2.1 ГРАФИЧНА БИБЛИОТЕКА В СРЕДАТА НА C++

Създаването на програми, които графично представят буквено-цифрова информация със средствата на компютърната графика се базира на създадената за съответния език графична библиотека от подпрограми (функции), реализиращи основните алгоритми на компютърната графика. В езика C++ тази библиотека се съдържа в заглавния файл **graphics.h**. Използването на подпрограмите, реализиращи различни графични алгоритми, изисква включването на графичната библиотеката (заглавния и файл) заедно с другите необходими за приложението библиотеки. Това налага при създаване на главна функция **main()**, която има графично приложение, заглавния файл **graphics.h** да бъде включен заедно с другите необходими на приложението заглавни файлове по начина показан с **Пример 2.1** Този пример илюстрира начина на включване в програмния текст на графично приложение на функции от библиотеката graphics.h**.**

Пример 2.1 Програмен текст на включване на графичната библиотека graphics.h и зареждане на графичен драйвер и режим за него с максимална разрешаваща способност, както и излизане от графичен режим след разглеждане на полученото изображение.

```
#include<iostream.h>
#include<graphics.h>  // Заглавен файл, съдържащ графичната библиотека
#include<conio.h>
#include<math.h>
#include<dos.h>
void main()
{
int gdriver = 0,gmode;
initgraph(&gdriver,&gmode,"C:\\borlandc\\bgi");
……
……
getche();         //
closegraph();
```

Пример 2.2 C++

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>

int main(void)
{

  int gdriver = VGA, gmode=VGAHI, errorcode;
```

```
   int xmax, ymax;


   initgraph(&gdriver, &gmode, 0);

    setcolor(getmaxcolor());
         line (100,200, 45,300);
   getch();
   closegraph();
   return 0;
}
```

Пример 2.3

```
#include <cstdlib>
#include <iostream>
#include <graphics.h>


#include <conio.h>
#include <math.h>
#include <cmath>

#include <stdlib.h>
#include <stdio.h>
using namespace std;

void draw_landscape(){

   setfillstyle(1, LIGHTBLUE);
   bar(0, 0, getmaxx(), getmaxy());

   //settextstyle(4, 0, 5);
   //char text[100] = "And this is how we Roll";
   //outtextxy(20, 20, text);
   setfillstyle(1,DARKGRAY);
   bar(0, getmaxy() - 250, getmaxx(), getmaxy());
   //setfillstyle(1, 15);
   //bar(0, getmaxy() - 175, getmaxx(), getmaxy() - 185);
}

int main(){
   initwindow(1024, 768, "Example");

  int initial_x = 20;
  int initial_y = getmaxy() - 280;

   draw_landscape();


   getch();
   return 0;
}
```

2.2. ГРАФИЧНА C++ БИБЛИОТЕКА - GRAPHICS.H

Според предназначението си функциите в графичната библиотека **graphics.h** могат да се определят в няколко групи. Основните групи графични функции са:

- ✓ инициализиращи графичния режим;

- ✓ проверяващи и променящи назначените параметри на графичното представяне;
- ✓ свързани с извеждането на текстове;
- ✓ изобразяващи графични примитиви и стандартни двумерни и тримерни обекти;
- ✓ съхраняване и възстановяване на изображения.

По-долу са разгледани най-често използваните функции като се описват техните имена, предназначение, параметри и примерно използване. При описанието на параметрите са възприети следните ***означения***:

color – означение цяла константа или променлива, съдържаща число в интервала [0;15], с което се указва поредния номер на цвета от цветовата палитра.
pattern – означение за цяла константа или променлива съдържаща число в интервала [0;11], с което се указва поредния номер на стила на щриховката а от разработените 12;
linestyle – означение за цяла константа или променлива, съдържаща число в интервала [0;3], с което се указва поредния номер на стила на линията а от разработените 4 такива;
x,y - означение за двойка цели константи или променливи, съответстващи на координатите в пиксели на една точка;
Dx,Dy - означение за двойка цели константи или променливи, съответстващи на отместване по координатните оси в пиксели;
thick={1,3} означение за цяла константа или променлива, указващо дебелината на линия;
page={0,1} означение за цяла константа или променлива, указващо номера на страница;
font = ={0,4} означение за цяла константа или променлива, указващо номера на стила на символите;
direction={0,1} означение за цяла константа или променлива, указващо последващо извеждане на текст дали ще бъде по хоризонтала или вертикала;
charsize - означение за цяла константа или променлива, указващо големината на символите;
horiz={0,1,2} – означение за цяла константа или променлива, указващо начина на центриране на текста при последващо изобразяване в хоризонтална посока;
vert={0,1,2}– означение за цяла константа или променлива, указващо начина на центриране на текста при последващо изобразяване във вертикална посока;
string – означение за константа или променлива, указваща текстова данна;
num – означение за цяла константа или променлива, указващо брой точки
arrxy – означение за масив от стандартни записи, съдържащи координати на точки;
sangle – означение за цяла константа или променлива, указваща начален ъгъл в градуси;
endangle – означение за цяла константа или променлива, указваща краен ъгъл в градуси;
R – означение за цяла константа или променлива, указваща радиус на окръжност в пиксели;
Rx, Ry - означение за цели константи или променливи, указващи радиуси на елипса в съответно на двете посоки в пиксели;

Фунции за назначаване на графичен режим

initgraph(GraphDriver, GraphMode, "\следа\...\...\BGI**") –** зарежда графичния драйвер и назначава режим за този драйвер. Параметрите GraphDriver, GraphMode са означения на цели променливи, указващи поредния номер на драйвера и режим за този драйвер. Назначаването автоматично на драйвер и режим за този драйвер изисква присвояването (GraphDriver = DETECT), като константата DETECT, съдържа стойност 0. Параметъра „\следа\....” е означение за символна данна, съдържащата описание на следата до папката, съдържаща файловете с разширение BGI.
closegrph() – освобождава паметта от графичния драйвер и възстановява предишния режим.
graphresult() – връща код указващ как е била изпълнена инструкцията **initgraph.**
grapherrormsg (ErrorCode) - връща текста на съобщение за грешка, съответстващо на параметъра **ErrorCode.**

```
initwindow(1024, 768, "Example");
initwindow(1024, 768, "Example");
```

Функции за промяна или проверка на назначените параметри на графичното представяне

setcolor(color) – назначава цвят на линия, указан с параметъра (color).
setbkcolor(color) – назначава цвят на фона, указан с параметъра (color).
setlinestyle(linestyle, pattern, thick) – назначава стил на линия (linestyle), стил на щриховка (pattern), и дебелина на линията (thick - 1 или 3 пиксела)
setfillstyle(pattern, color) – назначава стил на щриховка (pattern) и цвят, указан с параметъра (color).
setactivepage(page) – назначава активна страница.
setvisualpage(page) – назначава видима страница.
setviewport(x_1,y_1,x_2,y_2) – назначава графичен прозорец по зададени координати на горния му ляв (x_1,y_1) и долен десен (x_2,y_2) ъгъл.
getcolor() – връща цяло число, съответстващо на последно назначения цвят на линията.
getbkcolor() – връща цяло число, съответстващо на последно назначения цвят на фона.
getx() – връща цяло число, съответстващо на абцисата на текущата позиция на графичния курсор.
gety() – връща цяло число, съответстващо на ординатата на текущата позиция на графичния курсор.
getmaxx() – връща цяло число, съответстващо на максимално - възможната абцисна координата позволена от графичния режим и назначения за активната страница графичен прозорец.
getmaxy() – връща цяло число, съответстващо на максимално- възможната ординатна координата позволена от графичния режим и назначения за активната страница графичен прозорец.
getpixel(x,y) – – връща цяло число, съответстващо на цвета на точката зададена с координати (x,y).

Функции за работа с текстове в графичен режим

settextstyle(font, direction, charsize) – назначава стил (font) на символите , посока на извеждане (direction) и височина (charsize) на буквите при последващо извеждане на текст в графичен режим.
settextjustify(horiz, vert) – назначаване на центрирането на текст при последващо изобразяване съответно по хоризонтала и вертикала.
outtext(string) – извежда текста, указан с параметър (string) спрямо текущата позиция на графичния курсор.
outtextxy(x,y,string) – извежда текста, указан с параметър (string) спрямо точка, зададена с координати (x,y).
textheight(string) – връща цяло число, което съответства на височината на текста, указан като параметър (string) в пиксели.
textwidth(string) – връща цяло число, което съответства на ширината на текста, указан като параметър (string) в пиксели.

Функции изобразяващи основни графични обекти

putpixel(x, y, color) – изобразява пиксел с указаните координати (x,y) и цвят (color).
moveto(x,y) – премества графичния курсор до точка с координати (x,y).
moverel(Dx, Dy) – премества графичния курсор от текущата му позиция до точка отстояща на разстояние (Dx,Dy).
line(x_1,y_1,x_2,y_2) – изобразява отсечка между две точки по зададените им координати съответно (x_1,y_1) и (x_2,y_2).
lineto(x,y) – изобразява отсечка между текущата позиция на графичния курсор и точка с координати (x,y).

linerel(Dx,Dy) – изобразява отсечка между текущата позиция на графичния курсор и точка на разстояние (Dx,Dy).
drawpoly(num, arrxy) – изобразява полигон от (num) на брой точки, чийто координати са зададени с параметъра (arrxy).
fillpoly(num, arrxy) – изобразява затворен и защрихован полигон от (num) на брой точки, чийто координати са зададени с параметъра (arrxy).
floodfill(x, y, border) – генерира запълнена област с текущия стил и цвят на щриховка, започвайки от точката (x, y) до границата на затворения контур, чийто цвят е указан с параметъра (**border**)
rectangle(x_1,y_1,x_2,y_2) – изобразява правоъгълник по зададени координати на горен ляв ъгъл (x_1,y_1) и долен десен (x_2,y_2) ъгъл.
bar(x_1,y_1,x_2,y_2) – изобразява защрихован правоъгълник по зададени координати на горен ляв ъгъл (x_1,y_1) и долен десен (x_2,y_2) ъгъл.
bar3d(x_1,y_1,x_2,y_2, d, {topon/ topoff} – изобразява призма, за която са зададени координати на горен ляв ъгъл (x_1,y_1) и долен десен (x_2,y_2) ъгъл на челната стена, дебелината в пиксели (d) и това коя от двете основи да се изобрази съответно с избора на една от константите (topon/ topoff)**.**
circle(x,y, R) – изобразява окръжност с център (x,y) и радиуси R.
arc(x,y, sangle, endangle, R) – изобразява дъга от окръжност с център (x,y) и радиуси R. Дъгата от окръжността има за начало ъгъла (sangle) и за край ъгъла (endangle).
pieslice(x,y, sangle, endangle, R) – изобразява защрихован сектор от окръжност с център (x,y) и радиуси R. Дъгата на сектора има за начало ъгъла (sangle) и за край ъгъла (endangle).
ellipse(x,y, sangle, endangle, Rx, Ry) –изобразява дъга от елипса с център (x,y) и радиуси Rx, Ry. Дъгата има за начало ъгъла (sangle) и за край ъгъла (endangle).
fillelipse(x,y, Rx, Ry) – изобразява защрихована елипса с център (x,y) и радиуси Rx, Ry.
sector(x,y, sangle, endangle, Rx, Ry) – изобразява защрихован сектор от елипса с център (x,y) и радиуси Rx, Ry. Дъгата на сектора има за начало ъгъла (sangle) и за край ъгъла (endangle).

Пример 2.4 Главна функция която изобразява рибка, използвайки графичните функции за обекти със стандартна форма.

```
#include<iostream.h>
#include<graphics.h>
#include<conio.h>
#include<math.h>
void main()
{
int gdriver = 0,gmode;
initgraph(&gdriver,&gmode,"C:\\borlandc\\bgi");
int x0,y0,r,i=300,yt;
setbkcolor(1);
setfillstyle(1,YELLOW);
x0=500;y0=300;r=70;
while(i)
        {
        setcolor(YELLOW); //Назначаване на жълт цвят на линията
        setfillstyle(1,YELLOW); //Назначаване на жълт цвят на щриховката и стил – плътно
покриване
        ellipse(x0,y0,230,310,r,r);//Изобразяване на рибката      ellipse(x0,y0+107,50,130,r,r);
        floodfill(x0,y0+53,YELLOW); //Защриховане на тялото на рибката
        line(x0+45,y0+53,x0+65,y0+33);  //Контур -опашка
        line(x0+45,y0+53,x0+65,y0+73);
        line(x0+65,y0+33,x0+65,y0+73);
        floodfill(x0+50,y0+53,YELLOW); //Защриховане на опашката
        delay(10);
```

```
getche();
closegraph();
    }
```

2.3 ПРОГРАМИРАНЕ НА ГРАФИЧНИ ПРИЛОЖЕНИЯ В VISUAL C

Типове данни в Windows

В Windows – програмите вместо стандартните за C и C++ типове данни се използват типове, определени в библиотечни файлове (например, в WINDOWS.H). Често се използват следните типове:

- HANDLE - 32-разрядно цяло в качество на дескриптор (числов идентификатор на някакъв ресурс);
- HWND – дескриптор на прозореца (32- разрядно цяло);
- BYTE – 8-разрядно без-знаково символно значение;
- WORD – 16 – разрядно без-знаково цяло;
- DWORD, UINT – 32 разрядно без-знаково цяло;
- LONG – 32 – разрядно цяло със знак;
- BOOL – цяло, използва се за обозначаване на истина (1 - TRUE)
- или лъжа (0 – FALSE)
- LPSTR 32 – разряден указател към символен низ.

Структура на Windows – приложения

Работата на Windows – програма се основава на обработка на съобщения, постъпващи от потребителя, от операционната система и от други програми. Структурата на приложението обезателно включва главна функция WinMain, еднакво устроена за всички приложения. С нея започва изпълнението на всички Windows – програми. WinMain изпълнява следните основни действия:

1. Определя и зарегистрира клас прозорец в Windows.
2. Създава и изобразява прозорец, определен от даден клас.
3. Стартира цикъл за обработка на съобщения.

Всяко приложение взаимодейства с Windows чрез Application Programming Interface (API). API съдържа няколко стотин функции, които реализират всички необходими системни действия (отделяне на памет, създаване на прозорец, извеждане на екран и т.н.). Функциите API се съдържат в библиотеките с динамично зареждане (Dinamik Link Libraries (DLL)), които се зареждат в паметта само в момента, когато към тях възникне обръщение.
Едно от подмножествата на API е GDI (Graphic Device Interface- интерфейс на графичните устройства). Задачата на GDI е обезпечаване на апаратно независима графика. Благодарение на функциите на GDI Windows – приложението може да се изпълнява на различни компютри.

Библиотека Microsoft Foundation Class Library

Библиотеката MFC е базов набор от класове, написани на език C++ и предназначени за упростяване и ускоряване на процеса на програмиране под Windows. Библиотеката представлява йерархия от класове на много нива, обезпечаващи възможност за създаване на Windows проиложения на основата на обектно ориентирания подход. Предимство на MFC е, че с използването на класове се избягва голяма част рутинна работа и значително се съкращава обема на програмния текст. Интерфейсът, обезпечаван от библиотеката практически не зависи от реализиращите го детайли. Поради това програмите, написани на основа на MFC лесно се адаптират към нови версии на Windows.

Обработка на съобщения

Съобщението е уникално 32 битово цяло значение. Във файла WINDOWS.H за всички съобщения са определени стандартни имена. Често съобщенията се съпровождат с параметри, носещи допълнителна информация (координати на курсора, код на натиснат клавиш и т.н.). Всеки път, когато възникне събитие, касаещо програмата, Windows изпраща към нея съответно съобщение. Съобщенията, за които не е предвидена специална обработка от програмиста, се обработват в MFC програмата по стандартен начин.

СЪЗДАВАНЕ НА ГРАФИЧНИ ПРИЛОЖЕНИЯ
Генератор на приложения AppWizard

AppWizard е инструментално средство на Visual C++, упростяващо значително процеса на създаване на Windows – приложения на основата на MFC. При работа с този инструмент се появява последователност от диалогови прозорци, в които AppWizard задава на програмиста въпроси. В процеса на диалога потребителят определя типа и характеристиките на разработвания проект. Определяйки необходимите за този проект класове от библиотеката MFC, AppWizard създава проект на приложението и построява всички нужни производни класове. По-нататъшната работа на програмиста се състои в определяне на свойствата и поведението на обектите от тези класове:

- Създава се йерархия на класовете на графичните обекти – фигури. В основата на йерархията се поставя базовия клас, който отразява свойства и методи, общи за цялата фигура.
- От базовия клас се пораждат производни класове, които отговарят за рисуване на различни фигури. Всяка проста фигура може да се представи с обект от съответен клас.
- Въвежда се механизъм за съхраняване на фигурите.
- Допълва се методът за съхраняване и зареждане на изображението от файл със средства за коректна работа с версиите за формати на файлове.
- Допълва се интерфейса с команди за създаване на различни фигури.

Съхраняване на рисунката във файл

Рисунката се съхранява чрез запазване на координатите на опорните точки. Най-прост вариант е статически масив с достатъчно голям размер. За съхраняване на масива с координати обикновено се използва архитектура Document-View. Под документ се подразбира всякаква съвкупност от данни: текст, звук, изображение и т.н. Приложенията, построени с използване на архитектурата Document-View са два типа – еднодокументни (SDI-Single Document Interface) и многодокументни (MDI – Multiple Document Interface). SDI –приложенията позволяват едновременно редактиране на единствен документ (програми Notepad и Paint от набора стандартни програми на Windows). MDI приложенията позволяват редактиране на няколко приложения (MS Word) и поддържат едновременно различни типове документи. Предимство на архитектурата Document-View е, че тя позволява отделянето на данните от тяхното визуално представяне. В такива приложения едни и същи данни, съхранявани в обекта-документ, могат да се представят едновременно в различна форма с помощта на различни обекти-облици. Например даден график (един и същ набор данни) може в един прозорец да изглежда като таблица със значения, в друг прозорец - като крива, в трети - като диаграма. Обликът е обект, предназначен за изобразяване на данните от документа на екрана и обезпечаване на взаимодействието с потребителя. Потребителят наблюдава визуализирани с обекта-облик, рисунки и изпълнява редактиране. Обектът-облик (cview) приема управляващите действия на потребителя, свързва се с обекта-документ(cdocument) и изменя данните, използвайки методите на документа.

Опростена схема на взаимодействие на обектите в Windows –приложението:

- Създава се обект-приложение, който инициира създаването на обект-документ и обект-облик.
- Обектът-приложение съхранява данните, отнасящи се към цялото приложения и организира работата на документите и облиците.
- Обектът-документ съхранява данните, за обработката на които е предназначена програмата и притежава методи за тяхната модификация
- Обектът-облик визуализира данните, комуникира с потребителя и изменя данните.

За съхраняване на данните отговаря обектът-документ.

- При съхраняване първо се записва идентификатор на формата, а след това – данните
- При зареждане първо се чете идентификатора и в зависимост от значението му се избира схема на зареждане.
- Идентификатор на формата може да бъде всяко значение, например версията на програмата. Ако искаме програмата правилно да чете файлове с различни формати трябва да се предвиди правилно зареждане на данните, обусловено от идентификатора на формата.

Извеждане на рисунката на печат и предварителен преглед

Разликата в размера на рисунката на екрана и на лист хартия се обяснява с различната разрешаваща способност на двете устройства. Разрешаващата способност на устройството, се измерва в количествоот пиксели на единица размер (обикновено dot per inch – dpi).

Проблемът за съотношението на размерите на екрана и принтера може да се разреши с прилагането на следния алгоритъм:

1. Узнава се разрешението на екрана
2. Узнава се разрешението на принтера
3. Изчислява се съотношението на разрешението и се мащабира рисунката.

Чрез средствата на библиотеката MFC може да се установи такъв режим на изображение на обекта-облик, който позволява работа в пространството на логическите координати.

Пример 2.4

```
.....
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
        int wmId, wmEvent;

        PAINTSTRUCT ps;
        HDC hdc;
        TCHAR szHello[MAX_LOADSTRING];
        LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);
        POINT pp[3];
        static HBRUSH oldBrush;
        static HPEN oldPen;
        static HPEN newPen;
   static POINTS ptsPrevEnd;
switch (message)
        {
                case WM_COMMAND:
.....
        break;
                case WM_PAINT:
                {
                        hdc = BeginPaint(hWnd, &ps);
                        // TODO: Add any drawing code here...
                        RECT rt;
                        GetClientRect(hWnd, &rt);
                //      HPEN hpen=CreatePen(PS_SOLID, 10, RGB(255,255,255));
                        SetBkColor(hdc,RGB(120,220,180 ));
                        DrawText(hdc, szHello, strlen(szHello), &rt, DT_CENTER);
                // Painting 2 rectangles
                        HBRUSH hBrush1=CreateSolidBrush(RGB(0xFF,0xFF,0x00));
                        SelectObject(hdc, hBrush1);
                        Rectangle(hdc, 134, 456, 324, 190);
```

```
                HBRUSH hBrush2=CreatePatternBrush(LoadBitmap(hInst, (const
char*)IDB_BITMAP1));
                SelectObject(hdc, hBrush2);
                Rectangle(hdc, 634, 456, 340, 190);
        // Fill ellipse with user witmap image
                HBITMAP hBmp=LoadBitmap(hInst, (const char*)IDB_BITMAP2);
                HBRUSH hBrush4=CreatePatternBrush(hBmp);
                SelectObject(hdc, hBrush4);
                Ellipse(hdc, 134, 456, 324, 190);

                HBRUSH hBrush3=CreateHatchBrush(HS_CROSS,RGB(0x88,0x88,0x20));
                SelectObject(hdc, hBrush3);
                Ellipse(hdc, 590, 410, 370, 230);
        // ouput text in graphical window
                char* bla=" I can paint ";
                SetTextColor(hdc,RGB(0x5F,0x3F,0x11));
                SetBkColor(hdc,RGB(0x48,0xAB, 0xFF));  // yellow RGB(255,255,0)
                TextOut(hdc, 150, 100, bla, strlen(bla));
        // four_angle
                HBITMAP hBmp1=LoadBitmap(hInst, (const char*)IDB_BITMAP4);
                HBRUSH hBrush14=CreatePatternBrush(hBmp1);
                SelectObject(hdc, hBrush14);
                pp[0].x=760; pp[0].y=500;
                pp[1].x=925; pp[1].y=380;
                pp[2].x=990; pp[2].y=500;
                pp[3].x=890; pp[3].y=600;
                Polygon(hdc, pp,4);
        // END
                EndPaint(hWnd, &ps);
                break;
        }
```

2. 4 МАТЕМАТИЧЕСКИ АПАРАТ, ИЗПОЛЗВАН ОТ АЛГОРИТМИТЕ В КОМПЮТЪРНАТА ГРАФИКА

Математическите аспекти на алгоритмите в компютърната графика са изграждат върху основата на линейната алгебра и аналитична геометрия като надстройка, която е създадена за целите на компютърната графика. В тази глава се разглеждат основни математически понятия, така както се третират в областта на компютърната графика.

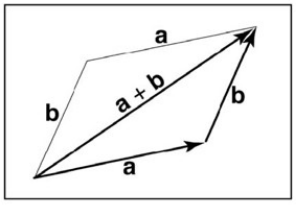
Вектори

Векторът е насочена отсечка. По долу ще бъдат използвани следните означения при работа с вектори и Фиг. 5.1

P,Q – крайни точки на отсечката;
a, b, c - вектори;
0 - вектор с нулева деължина;
-a - вектор с дължина $|a|$, насочен в посока обратна на тази на вектора a;
p,m - **реални числа;**
$|a|$ - дължина на вектора, равна на разстоянието между крайните точки на отсечката.

Свойства на векторите са представени със следните уравнения:

a + b = b + a
(a + b) + c = a + (b + c)
a + 0 = a



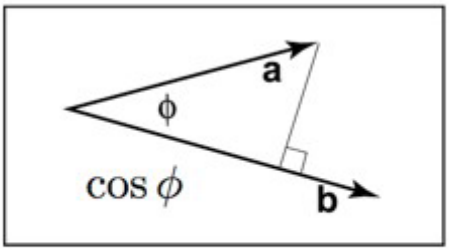
Сума на вектори

a + (-a) = 0
p(a + b) = pa +pb
(p + m)a = pa +ma
1a = a
0a = 0

Нормалите представляват вектори, които са перпендикулярни на лицето на даден полигон. Нормалните вектори са нужни при изчисленията на динамичната светлина. Ако не използвате готови обекти от помощните библиотеки (които са с определени нормални вектори), а такива създадени от вас, определянето на нормали за страните на обектите е задължително. За да изчислите нормалния вектор е нужно просто да вземете един от върховете на страната (която може да бъде триъгълник, квадрат или друга), да построите два вектора към съседните два върха и да ги **умножите векторно**.

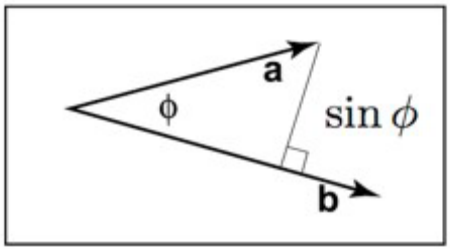
Събиране и

Скаларно умножение на векттори



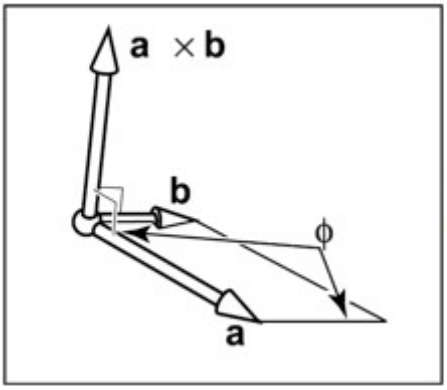
$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\phi$$

Скаларно пресичащо се умножение на векттори



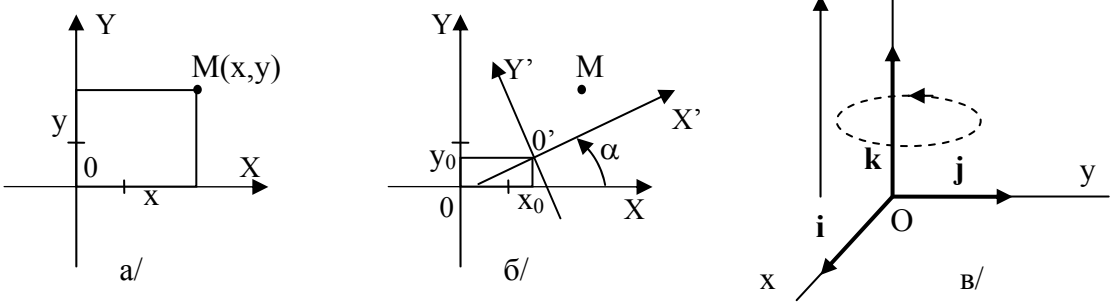
$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin\phi$$

Векторно умножение на вектори



$$\mathbf{c} = \mathbf{a} \times \mathbf{b}$$

$$\begin{aligned} x_c &= y_a z_b - z_a y_b \\ y_c &= z_a x_b - x_a z_b \\ z_c &= x_a y_b - z_a x_b \end{aligned}$$



Фиг 2. Координатни системи, математически елементи и особености
а/ Двумерна Декартова координатна система; б/ Трансформации на 2D координатна система; в/ Дясно-ориентирана 3D координатна система.

Координатни системи

Основните координатни системи, използвани в компютърната графика, бяха дадени по-горе в параграф 1.2. Тук ще обърнем внимание на някои математически подробности, относно координатните сиетми в Компютърната графика.

Двумерната, декартова координатна система в равнината е осев кръст, получен от пресичането на две перпендикулярни оси с дадени мащаби, които могат да бъдат равни или различни. Всяка точка от тази равнина се определя еднозначно от две числа (x,y), наречени координати на точката съответстващи на алгебричните мерки на проекциите на радиус вектора на точката съответно върху оста x и оста y. Пример на положителна (дясно) ориентирана декартова координатна система и еднозначно представяне на точка M е представена на Фиг. 2. а/.

Трансформации на координатни системи

Данните за графичните изображения се представят като съставени от обекти, описани със своите реперни точки. Смяната на една координатна система с друга променя координатите на точките от изображението по трансформационни формули, които изразяват новите координати в зависимост от старите, включващи параметрите определящи положението на новата координатна система спрямо старата. Пример за трансформация на координатна система е представен на Фиг 2 б/. Ако новата координатна система има начало определено с 0''(x_0,y_0) и осите и слючват ъгъл α със съответните оси на първоначалната декартова коодинатна система, решението включва промяна на началото (преобразуване транслация) и завъртане на осите около новото начало (преобразуване ротация). Зависимостите при смяна на началото са представени с уравнение (2.1), а ротацията около началото на координатната система с уравнение (2.2).

$$\begin{aligned} x'' &= x - x_0 \\ y'' &= y - y_0 \end{aligned} \qquad (2.1)$$

$$\begin{aligned} x' &= (x - x_0)\cos\alpha + (y - y_0)\sin\alpha \\ y' &= -(x - x_0)\sin\alpha + (y - y_0)\cos\alpha \end{aligned} \qquad (2.2)$$

където: x_0,y_0- координати на центъра на новата координатна система, α- ъгъл между осите на новата и съответните оси на първоначалната декартова коодинатна система, x,y- координати на точка спямо старата координатна система и x',y'- координати на точка спямо новата координатна система.

Тримерната координатна система се задава с тройка перпендикулярни единични вектори. Тримерната координатна системата се нарича дясноориентирана (Фиг. 2 в/), ако при въртене от вектор **i** към вектор **j** на 90° посоката на вектора k съвпада с постъпателното движение на винт с дясна резба. Началото на координатната система се означава с O. Всеки вектор V може да бъде записан във вид на линейна комбинация на i, j,k по следния начин:

V=xi + yj + zk, където x, y, z са координати на крайната точка P на вектора V. Векторът може да бъде записан и в матрична форма с уравнение (2.3).

$$V = [x, y, z] \text{ или } \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad (2.3)$$

Записи с използването на Еднородни координати

За да се избегне зависимостта на координатите от размерите на изображението се използват еднородни координати, които описват всяка точка с три координати (x, y, w), като w е единица на измерването на координатите, наричана мащабен множител. Въвеждането на еднородни координати се използва за вътрешно изменение на мащаба и има редица математически достойнства.

Уравнението: aX + bY +c = 0, описващо права в двумерното пространство. Ако се замени точката (X, Y), с (x/w, y/w) то ще получи вида (5.4):

ax + by + ch = 0 (2.4)

Това уравнение се нарича еднородно, (x, y, w) се наричат еднородни координати на точката (X, Y). Ако w=1 то уравнението (2.4) описва равнина минаваща през координатното начало и

зададената права линия. Ако се счита, че записа (x, y, w) е друг начин да се запише (x/w, y/w), то трябва w≠0.

Използването на еднородните координати позволява да се представи каквото и да е графично преобразование с уравнения включващи само матрични умножения. Ако едно двумерно преобразуване в декартови координати би могло да се запише като [x',y'] = [x y] A, където A е матрица на преобразуването. $A = \begin{bmatrix} a_1 & a_2 \\ b_1 & b_2 \end{bmatrix}$

В еднородни координоинати в двумерното пространство то може да се запише във вида:

[x' y' z'] = [x y z] A, където $A \begin{bmatrix} a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \\ c_1 & c_2 & c_3 \end{bmatrix}$

Използването на еднородни координати позволяват да се представят в матрична форма чрез матрични умножения (2.5) и (2.6) съответно на преобразованията транслация представено с уравнение (2.1) и последваща ротация -уравнение (2.2).

$$[u_1 \quad v_1 \quad 1] = [x \quad y \quad 1]T', \text{ където } T' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_0 & -y_0 & 1 \end{bmatrix} \qquad (2.5)$$

$$[u_2 \quad v_2 \quad 1] = [u_1 \quad v_1 \quad 1]R_0, \text{ където } R_0 = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (2.6)$$

Общия запис на трансформацията ще има вида (2.7), уравнение включващо само матрични умножения:

$$[x \quad y \quad 1] = [x \quad y \quad 1]\, T'\, R_0 \qquad (2.7)$$

Уравнение на права (2.8), зададена с две точки $P_1(x_1, y_1, w_1)$, $P_2(x_2, y_2, w_2)$, в еднородни координати

$$\det \begin{bmatrix} x & x_1 & x_2 \\ y & y_1 & y_2 \\ w & w_1 & w_2 \end{bmatrix} = 0 \text{ или } x(y_1w_2 - y_2w_1) + y(x_2w_1 - x_1w_2) + w(x_1y_2 - x_2y_1) \qquad (2.8)$$

Координатите на точка, получена от пресичането на две прави $L_1(a_1,b_1,c_1)$ и $L_2(a_2,b_2,c_2)$ се задават от членовете в скобите на уравнение (2.9).

$$a\,(b_1c_2 - c_1b_2) + b\,(c_1a_2 - a_1c_2) + c\,(a_1b_2 - b_1a_2) = 0 \qquad (2.9)$$

Положение на точка относно отсечка и многоъгълник

Положение на точка P(X,Y,W) спрямо отсечка L, чиито крайни точки са: (x_1,y_1,w_1) и (x_2,y_2,w_2). Ако W, w_1, w_2 са положителни, то точката P е разположена отдясно на правата, определена от отсечката L, само ако е изпълнено неравенството (2.10).

$$X(y_1w_2 - w_1y_2) + Y(x_2w_1 - w_2x_1) + W(x_1y_2 - y_1x_2) < 0 \qquad (2.10)$$

При определяне на положението на точката P(X,Y,W) спрямо многоъгълник се възможни следните случаи според това дали е изпъкнал или не. Ако многоъгълникът е изпъкна и върховете му са подредени по посока на часовниковата стрелка, то точката е разположена вътре в него, ако винаги се намира в дясно от наблюдателя при извършването на обход по страните на многоъгълника според подредбата на върховете му. В случай, че многоъгълникът не е изпъкнал, то търсенето на решение става по-трудно. Един от начините е да се раздели многоъгълника на няколко на брой изпъкнали многоъгълници и след това да се изследва последователно положението на точката спрямо всеки един от тях. Друг начин да се определи положението на точка спрямо многоъгълник е да се построи права през точката P(X,Y,W) и да се намерят точките на пресичане на тази права със страните на многоъгълника. След това чрез прилагането на критерия за четност върху получените пресечни точки със страните на многоъгълника се определя положението на точката. Този

втори начин е твърде трудоемък т.к. за да се определят пресечните точки трябва да се реши система с уравнения, получените точки трябва да се сортират за да се приложи критерия за четност. По тази причина този начин се използва само за не-изпъкнали многоъгълници.

```
/*
--------------------------------------------------------------------------------
-----------------
////////////////////////// Създаване на OpenGL прозорец с Win32 API
//////////////////////////////
--------------------------------------------------------------------------------
-----------------

Изграждането на OpenGL прозорец с Win32 API
За разлика от
Всяко windows базирано приложение установява цикъл на съобщенията във своята
WinMain( ) функция. Когато Windows изпрати съобщение ( то може да бъде най-
различно - от съобщение за натискане на даден клавиш до съобщение за активиране
на screensaver ) вашата програма прочита това съобщение ( ако има други след
него - чакат на опашка ) и го изпраща отново на Windows. Особеното тук, е че
вашия код трябва да съдържа една специална CALLBACK функция ( ще видите след
малко ), която се извиква многократно от Windows. Тя приема като аргумент
последното съобщение, което е било прочетено от вашата програма и върнато
обратно на Windows. Процесорното време трябва да бъде правилно разделено между
всички изпълняващи се приложения.
При създаването на проекта с Visual C++ изберете задължително Win32 Application,
а не Win32 Console Application както при GLUT.


#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>

void Render( ); // прототипът на рендериращата ни функция
bool Init( ); // прототипът на инициализиращата ни функция

// дефинираме променлива от тип HGLRC, в която ще се съдържа манипулатор за
OpenGL контекст
// на рендериране
HGLRC hRC;

// дефинираме променлива от тип HDC, в която ще се съдържа манипулатор за
контекст на устройство
HDC hDC;

/*
Не гледайте отчаяно. Контекста на устройството представлява просто структура,
която съдържа в себе
си някои други обекти като четка, писалка, шрифт и др. Можете да ги използвате,
когато изчертавате
нещо върху клиентската област от прозореца или върху т.н. битмап. Битмапът е
най-важния елемент,
понеже той играе ролята на платно, върху което рисуваме. За да получим достъп до
тези ресурси,
трябва да изискаме от Windows манипулатор за контекст на устройство на нашия
прозорец ( ще
видите как става това по-нататък ). Важно е да знаете, че всеки прозорец трябва
да има свой
собствен манипулатор за контекст на устройство.
А сега да обясня за другия манипулатор. След като получите ( ще видите как )
манипулатор за OpenGL
контекст на рендериране, свързан към контекста на устройството на съответния
прозорец, е нужно да
направите OpenGL контекста на рендериране текущ. Какво означава това? Ами много
просто. Представете
```

```
си че имате 10 прозореца, всеки със свой манипулатор за контекст на устройство и
по още един OpenGL
манипулатор за контекст на рендериране. Правейки някой от тези OpenGL контексти
на рендериране
текущ ( чрез съответния манипулатор ), вие избирате върху кой от прозорците да
се рендерира
вашата 3D графика. Може би обяснението ми не е върха, но е по-добре отколкото
нищо...
За да не се объркате тотално, е желателно да продължите с WinMain() функцията,
която се намира
някъде по-надолу!
*/


void SetupPixelFormat( HDC hDC )
{
  int nPixelFormat; // тук ще съхраним индентификатора на формата на пикселите

  // Създаваме структурната променлива pfd и попълваме нейните елементи. Не
обръщайте голямо
  // внимание на игнорираните елементи на структурата. Някои от тях по стандарт
дори вече не се
  // използват.
  static PIXELFORMATDESCRIPTOR pfd = {
        sizeof( PIXELFORMATDESCRIPTOR ),   // големина на структурата
        1,                                 // версия на структурата, в случая 1.0
        PFD_DRAW_TO_WINDOW |               // поддържа се рисуване в/у прозореца
        PFD_SUPPORT_OPENGL |               // поддържа се OpenGl
        PFD_DOUBLEBUFFER,                  // поддържа се двоен буфер
        PFD_TYPE_RGBA,                     // RGBA режим за цветовете
        32,                                // дълбочина на цвета
        0, 0, 0, 0, 0, 0,                  // игнорираме цветовите битове
        0, 0,                              // няма alpha буфер
        0,                                 // няма accumulation буфер
        0, 0, 0, 0,                        // игнорираме accumulation битовете
        16,                                // дълбочина на z-буфера
        0,                                 // няма stencil буфер
        0,                                 // няма auxiliary буфери
        0,                                      // игнорираме тази променлива
        0,                                 // резервираме определен брой слоеве
за изрисуване
        0, 0, 0 };                         // игнорираме маските на слоевете за
изрисуване

  // Функцията ChoosePixelFormat избира формат за пикселите, който съответства
на избраните от нас
  // в по-горната структура характеристики и се поддържа от контекста на
устройството. Функцията
  // връща индетификатор на този формат.
  nPixelFormat = ChoosePixelFormat( hDC, &pfd );

  // Установяваме избрания формат на пикселите.
  SetPixelFormat( hDC, nPixelFormat, &pfd );

  // Ако вече нищо не разбирате, значи всичко е наред. Това е най-малкото което
може да ви се случи
  // при Windows програмирането :)) Та общо взето това е при пикселния формат -
избирате режим на
  // рендериране, само дето тука за разлика от GLUT не става с една функция...
}


// Това е CALLBACK функцията ( или функцията на прозореца ), която се извиква от
Windows. Тя приема
```

```
// от Windows първите четири аргумента от структурата MSG, която видяхте във
функцията WinMain( ).
// Ако се чудите за какво говоря, значи нищо не сте гледали, веднага отидете на
WinMain( ), за да
// не идвам аз! :))
LRESULT CALLBACK WindowFunc( HWND hwnd, UINT message, WPARAM wParam, LPARAM
lParam )
{
  int width, height; // Ширина и дължина на прозореца, нужни за негово правилно
оразмеряване.

  switch( message )
  {
    // WM_CREATE е първото съобщение, което нашия прозорец получава веднага след
неговото създаване
    case WM_CREATE:
         // Изискваме контекст на устройство за нашия прозорец. Ако всичко
премине успешно
             // получаваме манипулатор към този ресурс.
         hDC = GetDC( hwnd );
             // Определяме формат на пикселите. Разгледайте функцията
SetupPixelFormat по-нагоре и ще
         // видите за какво става въпрос...
         SetupPixelFormat( hDC );
             // Изискваме OpenGL контекст на рендериране към контекста на
устройството. Ако всичко
         // премине успешно получаваме манипулатор към този ресурс.
         hRC = wglCreateContext( hDC );
             // Правим OpenGL контекста на рендериране на нашия прозорец текущ.
         wglMakeCurrent( hDC, hRC );
         return 0;
    break;

    // Получаваме съобщението WM_SIZE при промяна на размерите на прозореца.
    case WM_SIZE:
         // получаваме новите ширина и дължина на прозореца
         height = HIWORD( lParam );
         width = LOWORD( lParam );

         if( height==0 ) height = 1;

         glViewport( 0, 0, width, height );
         glMatrixMode( GL_PROJECTION );
         glLoadIdentity( );

         // определяме новата 3D перспектива на виждане
         gluPerspective( 45.0f, (GLfloat)width/(GLfloat)height, 0.1, 200.0 );

         glMatrixMode( GL_MODELVIEW );
         glLoadIdentity( );

         return 0;
    break;

    // При затваряне на прозореца...
    case WM_CLOSE:

    case WM_DESTROY:
         wglMakeCurrent( hDC, NULL ); // изключваме текущия OpenGL контекст за
рендериране
         wglDeleteContext( hRC ); // унищожаваме OpenGL контекста за рендериране
             // изискваме терминиране на приложението, чрез изпращане на WM_QUIT
съобщение от страна
         // на Windows
         PostQuitMessage(0);
```

```
          return 0;
     break;

     // При натискане на бутон от клавиатурата...
     case WM_KEYDOWN:

          switch(wParam)
          {
            // ако натиснатият бутон е "Escape", изискваме терминиране на
приложението,
            // чрез изпращане на WM_QUIT съобщение от страна на Windows
            case VK_ESCAPE:
                 PostQuitMessage(0);
            break;
          }
     break;
  }

  // Всички необработени съобщения от нашата CALLBACK функция се изпращат отново
на Windows с
  // помоща на функцията DefWindowProc( ) за обработка по подразбиране.
  return (DefWindowProc( hwnd, message, wParam, lParam ));
}


/*
Windows базираните програми притежават WinMain() функция, която не се различава
единствено по
името си с функцията main(), която доскоро използвахте във вашите програми.
Самата функция трябва
да се компилира посредством конвенцията за извикване използвана от Windows -
WINAPI или APIENTRY,
като двете са напълно аналогични. Ако не разбирате това, не се затормозявайте,
защо е така, а
просто го използвайте. Запазете търпението си за нещо по-смислено :) На
функцията WinMain()
се предават четири аргумента. Първият се отнася за текущата инстанция на
програмата. Вторият - за
предишната инстанция на програмата и винаги е равен на NULL, понеже Windows е
многозадачен и може
да изпълнява повече от една инстанция едновременно ( с изключение на Windows 3.1
). Третият
аргумент представлява указател към низ, съдържащ аргументите от командния ред,
зададени при
стартирането на програмата. Последният аргумент определя начина по който ще се
изобрази прозореца
на екрана.
*/

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
CommandLine, int nWinMode)
{
  // В самото начало на WinMain() функцията дефинираме някои променливи:
  // - структурна променлива от тип MSG, в която ще се съхраняват съобщенията
към прозореца
  // Ето какво представлява структурата MSG:
  /*
  typedef struct tagMSG
  {
  HWND hwnd; // манипулатора на прозореца, за който е отправено съобщението
  UINT message; // самото съобщение
  WPARAM wParam; // информация, свързана със съобщението
  LPARAM lParam; // още информация...
  DWORD time; // времето, през което е изпратено съобщението
```

```
  POINT pt; // това е структурна променлива, която съдържа x и y
местоположението на мишката
            // в момента когато е изпратено съобщението
  } MSG;
  */
  MSG msg;

  // - структурна променлива от тип WNDCLASSEX, с която ще дефинираме класа на
прозореца
  // Тази ще си я видите самички в хелп-а :)
  WNDCLASSEX wcl;

  // - променлива от тип HWND, която ще съдържа манипулатора на нашия прозорец
  HWND hwnd;

  // Може би се чудите какви са тези манипулатори и глупости, но след малко ще
видите... Отсега ви
  // казвам, че можете да използвате Help - а на вашия компилатор, за да
получите напълно
  // изчерпателна информация за типа на всички променливи, които виждате в този
код.

  // - променлива от тип DWORD, която ще използваме впоследствие за определяне
на типа на прозореца
  DWORD dwStyle = WS_OVERLAPPEDWINDOW;

  int WindowWidth = 640;      // - ширина на прозореца
  int WindowHeight = 480;     // - дължина на прозореца
  int BitDepth = 32;          // - дълбочина на цвета
  int DisplayFrequency = 85; // - честота на опресняване на екрана

  bool Finished;              // - в процес на рендериране ли сме?
  bool Fullscreen = false;     // - рендериране на цял екран?


  // Следващото нещо, което трябва да направим е да попълним елементите на
структурната променлива
  // wcl от тип WNDCLASSEX, която ще използваме при регистрирането на прозореца.


  // определяме големина на нашата WNDCLASSEX структура
  wcl.cbSize = sizeof( WNDCLASSEX );
  // определяме стил на прозореца. В случая използваме три аргумента, с които
изискваме съответно
  // прерисуване на прозореца при промяна на неговата дължина, ширина и
създаване на уникален
  // манипултор на устройство за всеки прозорец, регистриран с този клас. За
повече стилове можете
  // да проверите в Help - а, за структурата WNDCLASSEX...
  wcl.style = CS_HREDRAW | CS_VREDRAW | CS_OWNDC;
  // определяме функцията на прозореца
  wcl.lpfnWndProc = WindowFunc;
  // няма нужда от допълнителна информация за класа на прозореца и за самия
прозорец
  wcl.cbClsExtra = 0;
  wcl.cbWndExtra = 0;
  // посочваме манипулатора на текущата инстанция
  wcl.hInstance = hInstance;
  // определяме иконата на приложението
  wcl.hIcon = LoadIcon( NULL, IDI_APPLICATION );
  // определяме малката икона на изображението
  wcl.hIconSm = LoadIcon( NULL, IDI_APPLICATION );
  // определяме курсора на мишката
  wcl.hCursor = LoadCursor(NULL, IDC_ARROW);
  // цвят на фона ( в случая - никакъв )
```

```
  wcl.hbrBackground = NULL;
  // няма меню
  wcl.lpszMenuName = NULL;
  // име на класа на прозореца
  wcl.lpszClassName = "OpenGL Window Class";

  // Регистрираме класа на прозореца. Ако регистрацията не премине успешно,
Известяваме за грешка
  // с диалогов прозорец. WinMain() функцията връща нула и приложението се
затваря..
  if( !RegisterClassEx( &wcl ) )
  {
    MessageBox( NULL, "Window class registration unsuccessful!", "Error",
                    MB_OK | MB_ICONWARNING );
    return 0;
  }

  // Преди да изобразим самия прозорец, питаме потребителя с диалогов прозорец,
дали желае
  // приложението да се изпълнява на цял екран ( например всички професионално
направени игри )
  if( IDYES == MessageBox( NULL, "Do you want to switch to fullscreen mode?",
"Fullscreen",
                                          MB_YESNO | MB_ICONQUESTION |
MB_DEFBUTTON1 ) ) Fullscreen = true;

  // следващите инструкции се узпълняват, ако е избрано рендериране на цял екран

  if( Fullscreen )
  {

    DEVMODE dmScreenSettings;

      // Със следващата функция зареждаме текущите настройки на дисплея(
резолюция, дълбочина на
      // цвета и други ) в структурата dmScreenSettings. Вижте в хелп-а за
структурата от
      // предифиниран тип DEVMODE, ако исктате да се информирате по-подробно.
      if( !EnumDisplaySettings( NULL, ENUM_CURRENT_SETTINGS, &dmScreenSettings )
)
      {
            MessageBox(NULL, "Could Not Enum Display Settings", "Error", MB_OK |
MB_ICONWARNING);

      }

      // Задаваме стойностите на настройките, които ще променяме:
      dmScreenSettings.dmPelsWidth = WindowWidth; // ширина
      dmScreenSettings.dmPelsHeight = WindowHeight; // дължина
      dmScreenSettings.dmBitsPerPel = BitDepth; // дълбочина на цвета
      dmScreenSettings.dmDisplayFrequency = DisplayFrequency; // честота на
опресняване на монитора

      // Заменяме настройките на дисплея с тези от структурата dmScreenSettings.
Ако функцията
      // ChangeDisplaySettings() върне различна стойност от макроса
DISP_CHANGE_SUCCESSFUL извеждаме
      // съобщение за грешка.
      if( DISP_CHANGE_SUCCESSFUL != ChangeDisplaySettings( &dmScreenSettings,
CDS_FULLSCREEN ) )
      {
        MessageBox( NULL, "Change to fullscreen unsuccessful! Program will
continue in normal mode.", "Error",
                        MB_OK | MB_ICONWARNING );
```

```
    }
      // Ако сме сменили успешно настройките, определяме стил на прозореца - без
рамка( все пак при
      // режим на цял екран това е напълно ненужно ) и скриваме курсора на
мишката.
      else {

     dwStyle = WS_POPUP;

       ShowCursor( false );
    }
  }


  // Следващата функция създава самия прозорец и връща манипулатор към този
прозорец
  hwnd = CreateWindow( "OpenGL Window Class", // името на регистрирания вече
клас на прозореца
                                "Created with Win32 API", // определяме име на
прозореца
                                dwStyle, // определяме стила на прозореца в
зависимост от dwStyle
                        0, 0, // x и y координати на прозореца
                        WindowWidth, // определяме ширина на прозореца
                        WindowHeight, // определяме дължина на прозореца
                        NULL, // манипулатор към родителски прозорец ( няма такъв
)
                        NULL, // манипулатор към меню ( няма такова )
                        hInstance, // манипулатор на текущата инстанция на
програмата
                        NULL);   // указател към допулнителна информация ( няма )

  // Ако сме получили NULL, вместо валиден манипулатор на прозорец, значи
прозореца не може да
  // бъде създаден. Известяваме за грешка с диалогов прозорец. След това
WinMain() функцията
  // връща нула и приложението се затваря.
  if( !hwnd )
  {
    MessageBox( NULL, "Creating window unsuccessful!", "Error",
                      MB_OK | MB_ICONWARNING );
      return 0;
  }

  ShowWindow( hwnd, SW_SHOW ); // изобразяваме създадения от нас прозорец
  UpdateWindow( hwnd );        // указваме, че прозореца трябва да пъде
прерисуван, за всеки случай

  // Разбрахте ли вече за какво са тези манипултори? Както виждате те са един
вид имена. Например
  // ако имате повече от един прозорец и искате да скриете точно определен, няма
как да направите
  // това, без да кажете за кого се отнася заповедта т.е. използвате съответния
прозорец.

  Finished = false; // това го разбирате нали :)

  // Извикваме инициализиращата ни функция и ако тя върне нула, значи нещо не е
наред и няма
  // да рендерираме нищо.
  if( !Init() ) Finished = true;

  // Това е цикъла на съобщенията, за който стана на въпрос в самото начало. Той
продължава докато
  // променливата Finished не получи стойност false.
```

```
  while( !Finished )
  {
      // Зареждаме следващото съобщения от опашката ( ако има такова ) в
структурната променлива msg
    // от тип MSG. Ако няма съобщение на опашката функцията връща нула.
      if( PeekMessage( &msg, 0, 0, 0, PM_REMOVE ) )
    {
      // В случай, че сме получили съобщения за затваряне на програмата,
променяме стойността на
      // Finished, така че цикъла на съобщенията да завърши.
      if( msg.message == WM_QUIT ) Finished = true;

        // Преобразуваме суровия клавиатурен код в знакови съобщения
      TranslateMessage( &msg );
      // Изпращаме последното прочетено и преобразувано съобщение обратно на
Windows
      DispatchMessage( &msg );
    }
    // Извикваме рендериращата ни функция, ако няма съобщения на опашката,
чакащи да бъдат
    // обработени и върнати на Windows.
    else Render( );
  }

  // Финцкията WinMain( ) завършва, като изпраща стойността msg.wParam,
  // съдържаща кода на WM_QUIT съобщението при затварянето на програмата.
  return msg.wParam;
}


bool Init( )
{
  glClearColor( 0.0, 0.0, 0.0, 0.0 );

  return true;
}


void Render( )
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ); // изчистваме буферите
  glLoadIdentity( ); // зареждаме първоначалната матрица

  glTranslatef( 0.0, 0.0, -10.0 ); // преместваме сцената 10 единици напред

  // изрисуваме триъгълник
  glBegin(GL_TRIANGLES);

      glVertex3f( -2.0, -2.0, 0.0 );
      glVertex3f( 0.0, 2.0, 0.0 );
      glVertex3f( 2.0, -2.0, 0.0 );

  glEnd();

  // Разменяме двата буфера за изрисуване. Забележете че за аргумент подаваме
контекста
  // на устройството на нашия прозорец.
  SwapBuffers( hDC );
}
```