Нижегородский государственный университет
им. Н.И. Лобачевского

Механико - математический факультет

Кафедра теоретической механики

Февральских Л.Н., Маркина М.В.

ЛАБОРАТОРНЫЕ РАБОТЫ ПО КУРСУ «КОМПЬЮТЕРНАЯ ГРАФИКА»

(учебно-методическая разработка)

Н.Новгород, 2015

ГЛАВА 1. Работа над проектом с графическим интерфейсом

1. Создание проекта с графическим интерфейсом

Для создания нового проекта в Visual Studio необходимо:

1) В главном меню выбрать **File ► New ► Project** (рис. 1);

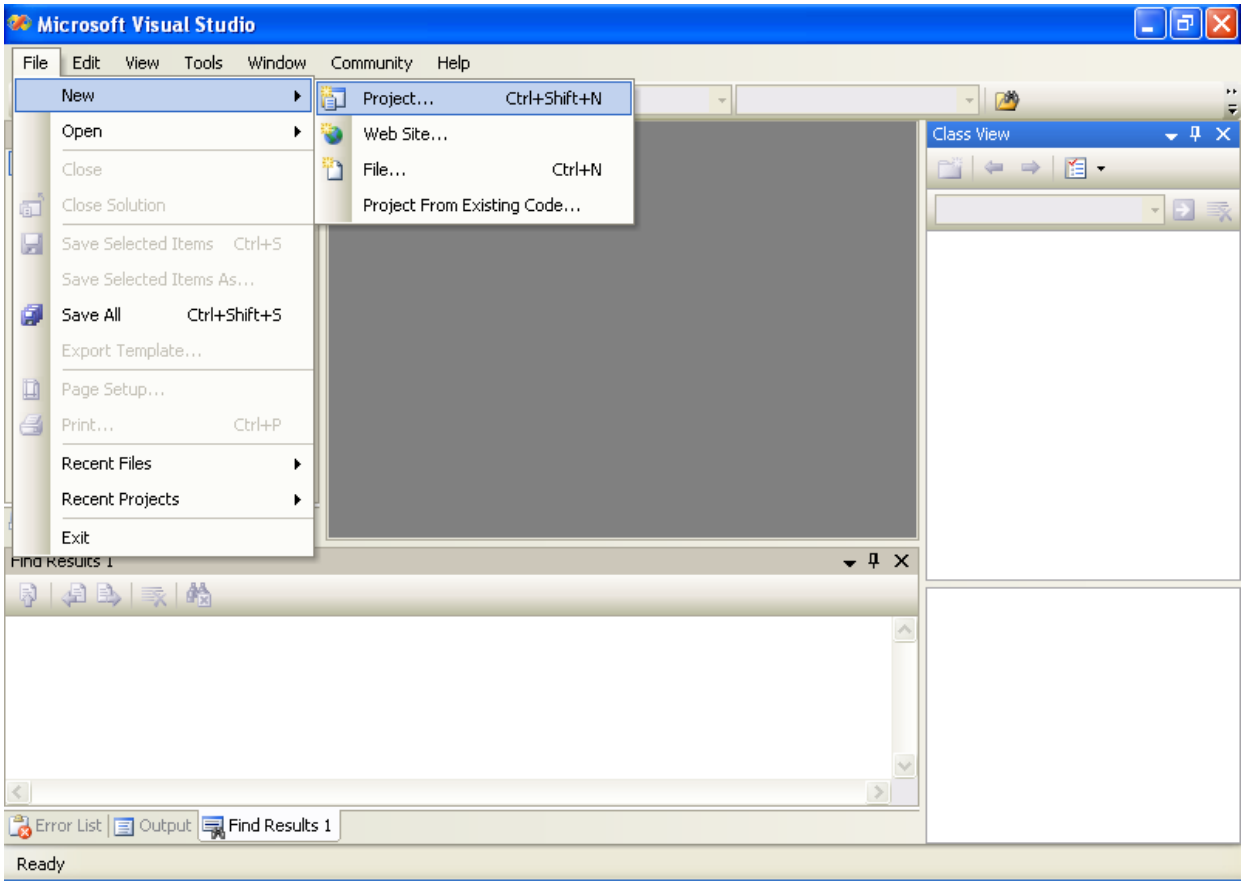


Рис. 1. Создание проекта с графическим интерфейсом. Шаг 1.

2) В открывшемся окне **New Project** в списке **Visual C++** выбрать **CLR** (Common Language Runtime) и вид приложения **Windows Forms Application** (рис. 2);
3) В поле Name ввести название проекта и нажать **OK**.

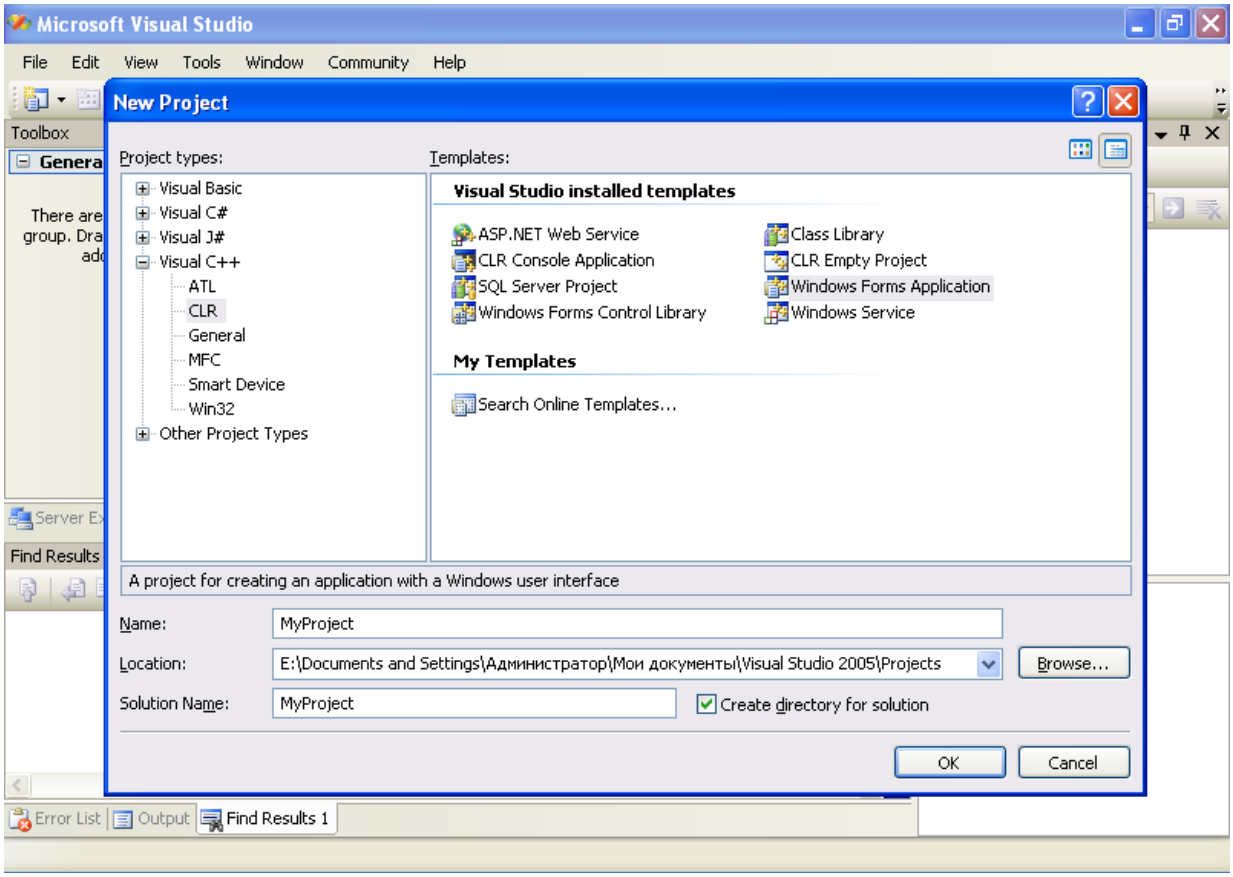


Рис. 2. Создание проекта с графическим интерфейсом. Шаг 2.

2. Объекты интерфейса и их свойства

Когда проект создан, на экране появляется конструктор форм с главным окном приложения – главной формой (**Form**) (рис. 3).

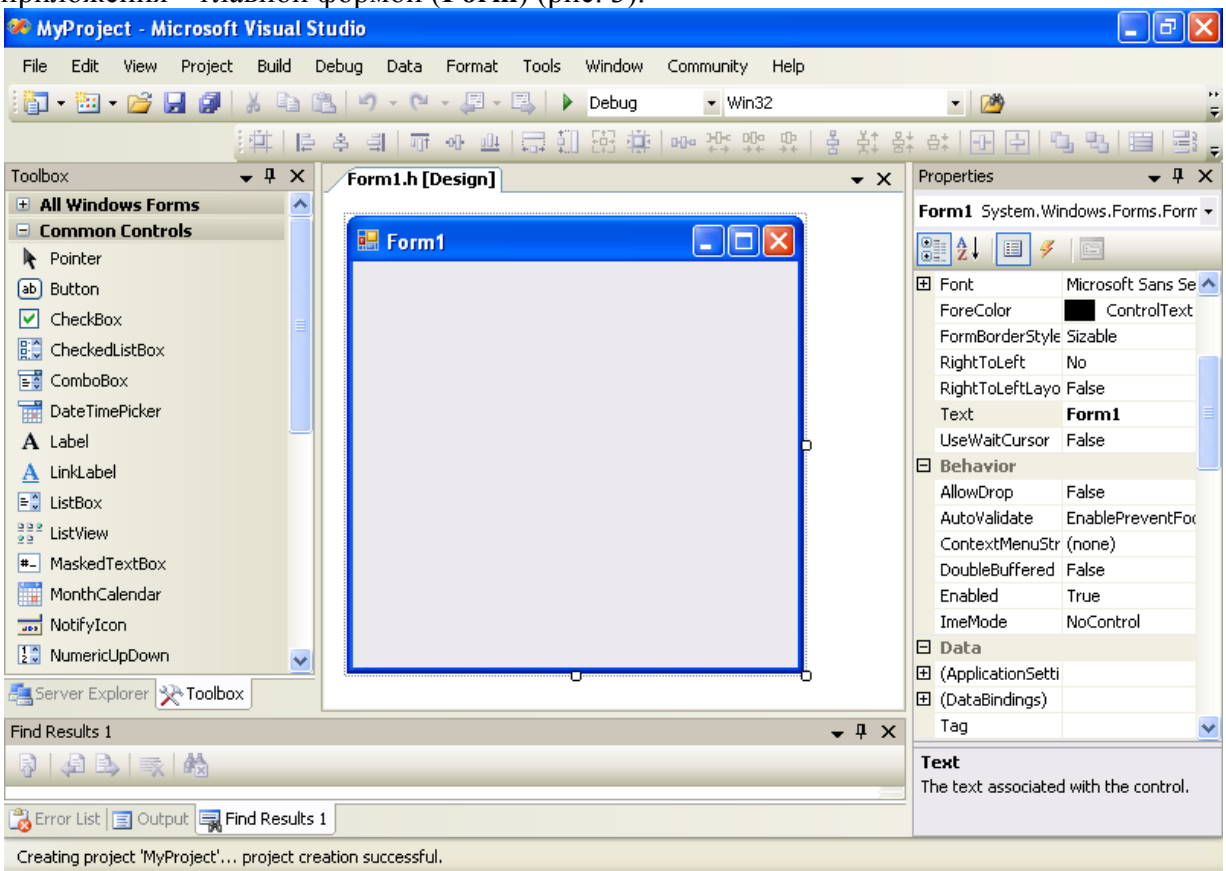


Рис. 3. Конструктор формы с окнами Toolbox и Properties.

На форму помещаются необходимые компоненты пользовательского интерфейса: поля ввода информации (**TextBox**), кнопки (**Button**), поля отображения текста (**Label**), окно вывода графики (**PictureBox**) и др. Они содержатся в окне **Toolbox** в списке **Common Control** (рис. 3). Для отображения окна следует в главном меню выбрать **View ► Toolbox**.

У каждого компонента и формы приложения имеется набор свойств. Их редактирование осуществляется в окне **Properties** (рис. 3). Если оно не отображается, следует выполнить команду **View ► Properties Window** или нажать **F4**.

В табл. №1-5 приведены основные свойства объектов, которые могут быть полезны для выполнения практических заданий.

Табл. №1. Свойства объекта **Form**

Свойство	**Описание**
Name	Имя формы (Не отображается в окне Properties. Прописывается только в коде)
Text	Текст заголовка
Size	Размер формы **Width** – ширина, **Height** – высота.
Location	Положение формы на экране **X** – расстояние от левой границы формы до левой границы экрана, **Y** – расстояние от верхней границы формы до верхней границы экрана.
FormBorderStyle	Тип границы **Sizable** – стандартное окно с заголовком и кнопками свернуть, развернуть, закрыть, **FixedSingle** – стандартное окно фиксированного размера с тонкой границей, **Fixed3D –** стандартное окно фиксированного размера с объемной границей, **FixedDialog** – диалог, **SizeableToolWindow** – окно без кнопки Свернуть, **FixedToolWindow** – окно без кнопки Развернуть, **None** – окно без заголовка и границы.

Табл. №2. Свойства компонента **TextBox**

Свойство	**Описание**
Name	Имя поля ввода
Text	Текст, находящийся в поле
Size	Размер поля
Location	Положение поля на форме
Font	Шрифт, используемый для отображения текста
TextAlign	Выравнивание текста **Left** – по левому краю, **Right** – по правому краю, **Center** – по центру.
MaxLength	Максимальное количество символов в поле
Multiline	Разрешение на ввод многострочного текста **True** – разрешено, **False** – запрещено.
ReadOnly	Разрешение редактирования текста **True** – разрешено, **False** – запрещено.

ScrollBars	Отображение полос прокрутки **Horizontal** – горизонтальной, **Vertical** – вертикальной, **Both** – горизонтальной и вертикальной, **None** – не отображать.

Табл. №3. Свойства компонента **Label**

Свойство	**Описание**
Name	Имя компонента
Text	Отображаемый текст
Size	Размер компонента
Location	Положение компонента на форме
Font	Шрифт, используемый для отображения текста
TextAlign	Выравнивание текста **Left** – по левому краю, **Right** – по правому краю, **Center** – по центру.

Табл. №4. Свойства компонента **Button**

Свойство	**Описание**
Name	Имя кнопки
Text	Текст на кнопке
Size	Размер
Location	Положение кнопки на форме
Font	Шрифт, используемый для отображения текста
TextAlign	Положение текста на кнопке **MiddleCenter, MiddleLeft, MiddleRight** – в центре по вертикали, **TopCenter, TopLeft, TopRight** – прижат к верхнему краю, **BottomCenter, BottomLeft, BottomRight** – прижат к верхнему краю.
Enabled	Активность (работоспособность) кнопки **True** – активна, **False** – неактивна (не реагирует на нажатие).
Visible	Видимость кнопки **True** – видима, **False** – скрыта.

Табл. №5. Свойства компонента **PictureBox**

Свойство	**Описание**
Name	Имя компонента
Size	Размер
Location	Положение компонента на форме
SizeMode	Вариант масштабирования содержимого, если его размер не соответствует размеру компонента **Normal** – масштабирование не выполняется, **StretchImage** – содержимое занимает всю область компонента, **AutoSize** – размер компонента соответствует размеру содержимого, **CenterImage** – центрирование содержимого в поле компонента, **Zoom** – масштабирование содержимого под размеры компонента с сохранением пропорций содержимого.

3. Обработка событий

Для того чтобы программа выполняла некоторую работу в ответ на действия пользователя, нужно описать соответствующее событие.

Список возможных событий для выбранного объекта можно увидеть в окне **Properties** на вкладке **Events**, обозначаемой символом молнии  (рис. 4).

Так, например, для формы **Form** наиболее часто используют события **Load** (загрузка формы), **Resize** (изменение размера окна), для кнопки **Button** – событие **Click** (нажатие на кнопку), для компонента **PictureBox** – событие **Paint** (рисование в области компонента).

Каждому событию в коде программы соответствует функция обработки этого события. Реализация этих функций прописывается в файле формы с расширением **.h (Form1.h)**. Для того чтобы функция обработки некоторого события была включена в код, нужно в окне **Properties** на вкладке **Events** сделать двойной щелчок левой кнопкой мыши по пустому полю рядом с названием этого события (рис. 4).

Например, сделаем возможным реакцию программы на нажатие кнопки с именем **button1** (событие **Click**). В этом случае в код главного модуля приложения будет добавлена функция обработки события вида:

```
private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
{
}
```

Функция возвращает тип `System::Void` (ничего не возвращает), имя функции `button1_Click`, аргументы функции `sender` и `e` – указатели (обозначаются символом `^`) на объекты системных типов `System::Object` и `System::EventArgs` соответственно.

Последовательность действий, выполняемых при нажатии на кнопку, нужно прописать в теле функции обработки.

Если по каким-то причинам функция оказалась ненужной, следует знать, что удаления функции из кода **недостаточно!** При создании события в файл **Form1.h** также добавляется строка вида:

```
this->button1->Click += gcnew System::EventHandler(this,
&Form1::button1_Click);
```

Без детального разъяснения отметим, что эта строчка связывает событие `Click` и функцию с именем `button1_Click`. При наличии этой строки в коде компилятор будет «искать» объявление функции с таким именем и, если не найдет, выдаст несколько ошибок.

Событие будет удалено, если вместе с функцией его обработки удалить строку-связь этой функции с названием события.

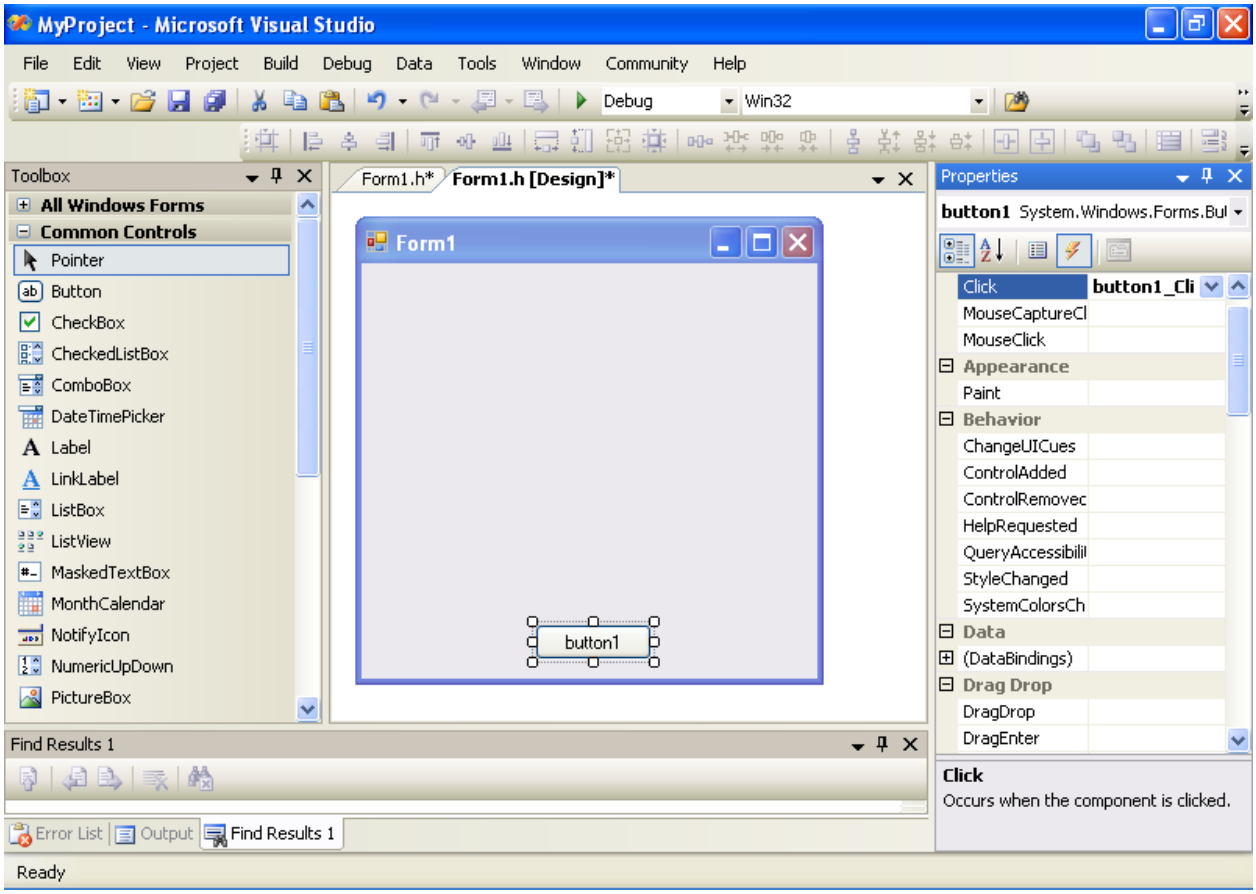


Рис. 4. Конструктор формы с кнопкой и списком событий для нее.

4. Доступ к объекту

Каждый объект интерфейса программы (**Form**, **Button**, **TextBox** и т.д.) реализован с помощью класса. Класс – это некоторый тип данных. Например, главное окно приложения является объектом класса `Form1` (изначально совпадает с именем формы), наследуемым от базового класса `System::Windows::Forms::Form`, кнопка на форме – объект класса `System::Windows::Forms::Button`, поле ввода текста – объект класса `System::Windows::Forms::TextBox`.

При добавлении в проект нового элемента графического интерфейса определяется указатель на объект соответствующего класса. Сам объект создается при старте приложения и его адрес заносится в указатель, имя которого известно. **Все операции с элементами интерфейса осуществляются через указатели!**

В частности, когда компонен помещается на форму, в класс `Form1` добавляется поле, содержащее указатель на объект соответствующего класса. Доступ к этому компоненту можно получить с помощью оператора стрелка (`->`) через указатель на форму, в области которой расположен компонент. Этот указатель имеет имя `this`.

Например, если форма содержит кнопку с именем `button1`, то доступ к ней осуществляется инструкцией `this->button1`. В действительности `button1` – это указатель, содержащийся в классе `Form1`. Через указатель `button1` можно получить доступ к свойствам кнопки, например изменить свойство `Text` (текст на кнопке):

```
this->button1->Text = "OK";
```

5. Пространство имен и полезные функции пространства System::Convert

В модуле формы Form1.h в инструкции `using namespace` перечислены пространства имен, которые использует программа. Например:

```
using namespace System;
using namespace System::Windows::Forms;
using namespace System::Drawing;
```

В действительности пространство имен – это контейнер, который предоставляет программе свои объекты (типы, функции, константы и др.)

Например, пространство имен `System::Windows::Forms` содержит объекты `Form, TextBox, Button, Label` и т.д.

Для получения доступа к объекту пространства имен следует указать идентификатор пространства имен, которому принадлежит объект, затем оператор области видимости (`::`), и наконец, имя объекта, т.е.
`Идентификатор_Пространства_Имен::Имя_Объекта.`

Полезным является пространство имен `System::Convert`. Оно содержит функции конвертации из строки в число и обратно.

Рассмотрим функции перевода строки в числа типа `double` и `int`:

```
double System::Convert::ToDouble(System::String^ str);
int System::Convert::ToInt32(System::String^ str);
```

В качестве аргумента в функции передается указатель (^) на строку. Под строкой здесь и далее понимается объект типа `System::String`. Например, содержимое поля ввода с именем `textBox1`, т.е. объект `this->textBox1->Text`, является указателем на объект типа `System::String`.

Существует несколько функций перевода из числа в строку с общим названием `ToString`. Функции возвращают указатель на объект типа `System::String` и отличаются друг от друга типом аргумента:

```
System::String^ System::Convert::ToString(double num);
System::String^ System::Convert::ToString(int num);
```

Рассмотрим пример использования функций конвертации.

Пример 1. Использование функций `ToInt32, ToDouble, ToString`

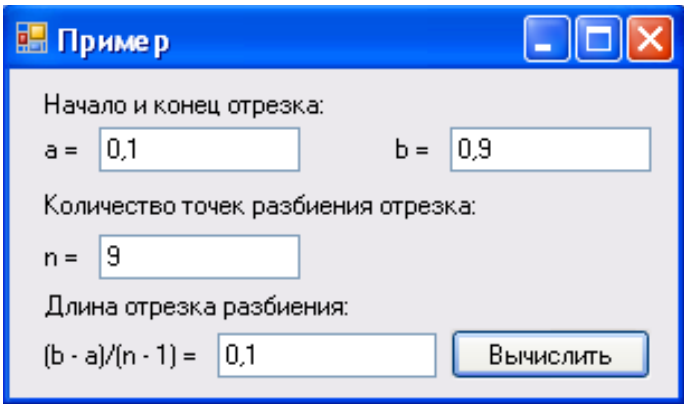


Рис. 5. Интерфейс программы.

Пусть имеется отрезок $[a, b]$. Разобьем его на равные части n точками $x_i = a + i\Delta x$, $i = 0, \dots, n-1$ таким образом, чтобы $x_0 = a$, $x_{n-1} = b$, и вычислим длину отрезка разбиения Δx.

Создадим проект **Windows Forms Application**. На главной форме приложения разместим четыре поля ввода, подписи к ним и кнопку, как показано на рис 5. Числа a, b и n вводятся пользователем в текстовые поля `textBox1, textBox2` и `textBox3`

соответственно. После нажатия кнопки button1 в поле textBox4 выводится длина отрезка разбиения, вычисленная по формуле:

$$\Delta x = \frac{b-a}{n-1}.$$

Создадим событие Click кнопки button1. В теле функции button1_Click применим функции конвертации.

```
private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
{
     %Текст из полей ввода конвертируем в числа нужного формата.
Сохраняем числа в переменных a, b и n.%
     double a = System::Convert::ToDouble(this->textBox1->Text);
     double b = System::Convert::ToDouble(this->textBox2->Text);
     int n = System::Convert::ToInt32(this->textBox3->Text);
     %Длину отрезка переводим в строку и выводим в textBox4.%
     this->textBox4->Text = System::Convert::ToString((b - a)/(n
- 1));
}
```

6. Проверка правильности ввода в TextBox

Во избежание ошибок пользователя необходимо проводить проверку правильности ввода данных. Для примера 1 простейшая проверка предусматривает выполнение неравенства a < b и условия непустоты строк в текстовых полях, например this->textBox1->Text != "".

Для проверки более высокого уровня требуется контролировать, что введены именно числа, причем того формата, который в данном случае необходим. Это можно сделать при помощи функции-метода TryParse, находящегося в системных структурах Int32 и Double:

```
System::Boolean Int32::TryParse(System::String^ s, Int32 n);
System::Boolean Double::TryParse(System::String^ s, Double a);
```

Первый аргумент s - строка, преобразуемая в число, второй аргумент – переменная, в которую заносится число, если преобразование прошло успешно. В противном случае ей присваивается значение, равное нулю. В качестве второго аргумента функций вместо переменных типа Int32 и Double можно использовать переменные стандартных типов int и double.

Функция TryParse возвращает true, если строка s успешно преобразована в число требуемого типа, во всех остальных случаях – false.

В частности для примера 1 проверка правильности ввода может иметь вид:

```
if (Double::TryParse(textBox1->Text, a) &&
Double::TryParse(textBox2->Text, b) &&
Int32::TryParse(textBox3->Text, n))
{
     %Конвертация параметров задачи.%
     ...
}
else
     %Предупреждение об ошибочном или незавершенном вводе
данных.%
     ...
```

7. Диалоговое окно

Если пользователь ошибся при вводе, можно предупредить его вызовом сообщения при помощи функции `Show` из пространства имен `MessageBox`:

```
MessageBox::Show(System::String^ text, System::String^ caption,
System::Windows::Forms::MessageBoxButtons buttons,
System::Windows::Forms::MessageBoxIcon icon);
```

Первые два аргумента содержат текст сообщения (`text`) и заголовок диалогового окна (`caption`). Третий аргумент `buttons` определяет набор стандартных кнопкок, содержащихся в окне. Список констант, соответствующих наборам кнопок, содержатся в пространстве имен `MessageBoxButtons`. Они сведены в табл. №6.

Табл. №6. Стандартные наборы кнопок диалогового окна

Константы	Кнопки, содержащиеся в окне
AbortRetryIgnore	Прервать, Повторить и Пропустить
OK	Ок
OKCancel	Ок и Отмена
RetryCancel	Повторить и Отмена
YesNo	Да и Нет
YesNoCancel	Да, Нет и Отмена

Можно обозначить характер сообщения, добавив в диалоговое окно один из четырех системных значков. Это позволяет сделать четвертый аргумент `icon`, который должен совпадать с одной из стандартных констант пространства имен `MessageBoxIcon`, приведенных в табл. №7.

Табл. №7. Виды сообщений

Константы	Характер сообщения	Значок, отображаемый в окне
Error, Hand, Stop	Сообщение об ошибке	
Information, Asterisk	Информационное сообщение	
Question	Вопрос	
Warning, Exclamation	Предупреждение	
None	Нейтральное сообщение	

Таким образом, предупреждение об ошибке ввода может иметь вид (рис. 6):

```
MessageBox::Show("Ввод данных произведен неверно!","Ошибка
ввода",MessageBoxButtons::OK, MessageBoxIcon::Error);
```

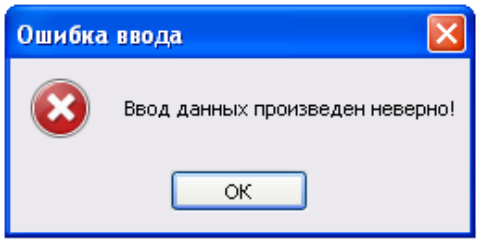


Рис. 6. Сообщение об ошибке ввода

Для полноценного диалога с пользователем нужно контролировать, какая кнопка в окне с сообщением была нажата. Это можно сделать, сравнив значение, возвращаемое функцией `MessageBox::Show`, с «кодом» кнопки. Каждой стандартной кнопке диалогового окна соответствует определенная константа («код»), которая содержится в пространстве имен `System::Windows::Forms::DialogResult`. Так, например, можно задать действия для повторного ввода данных (рис. 7):

```
%Если нажата кнопка Повтор%
if(MessageBox::Show("Ввод данных произведен неверно!","Ошибка
ввода", MessageBoxButtons::AbortRetryIgnore,
MessageBoxIcon::Warning) ==
System::Windows::Forms::DialogResult::Retry)
{
     %Очистка поля, в котором сделана ошибка ввода.%
     this->textBox1->Text = "";
}
```

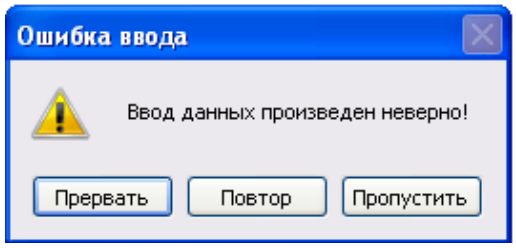


Рис. 7. Диалоговое окно, позволяющее выбрать действие

ГЛАВА 2. Графика

1. Отображение графики

Прорисовку графических объектов удобно производить в области специального компонента **PictureBox**. С этой областью связана локальная система координат с началом в левом верхнем углу и осями Ox, Oy, ориентированными согласно рис. 8. Единица, отсчитываемая по осям координат, равна 1 пикселю. Таким образом каждой точке области `pictureBox` ставится в соответствие пара чисел (x, y).

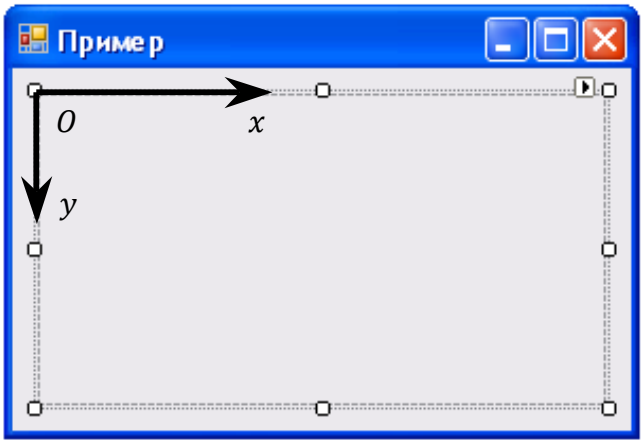


Рис. 8. Ориентация системы координат, связанной с областью **PictureBox**.

При выполнении программы можно получить размер области `pictureBox` через указатель на объект этого типа. Например, для компонента с именем `pictureBox1` ширина и высота внутренней области определяется инструкциями:

```
int w = this->pictureBox1->Width;        //Ширина
int h = this->pictureBox1->Height;       //Высота
```

Для отображения содержимого **PictureBox** нужно вызвать функцию обработки события **Paint**. Создание события **Paint** для компонента с именем **pictureBox1** приведет к добавлению в код функции вида:

```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e)
{
}
```

В теле функции прописываются строки кода, выполняемые при активации этого события, например, вызов функций рисования графических примитивов или отображения текста.

Часто, бывает, нужно вызвать функцию `pictureBox1_Paint` в теле другой функции, действующей в классе `Form1`, например в функции `button1_Click`. Чтобы при нажатии кнопки `button1` отображалось содержимое `pictureBox1`, необходимо в тело функции `button1_Click` включить несколько вспомогательных строк кода:

```
private: System::Void button1_Click(System::Object^  sender,
System::EventArgs^  e)
{
     System::Drawing::Graphics^ graph = this->pictureBox1-
>CreateGraphics();
     System::Drawing::Rectangle rect;
     System::Windows::Forms::PaintEventArgs^ e1 = gcnew
System::Windows::Forms::PaintEventArgs(graph, rect);
     this->pictureBox1->Paint(sender, e1);
}
```

2. Создание графических примитивов

Вывод графики обеспечивают функции объекта **Graphics**. Доступ к функциям этого объекта осуществляется через аргумент `e` функции обработки события `Paint (pictureBox1_Paint)` с помощью инструкции вида:
`e->Graphics->Имя_Функции(Аргументы_Функции).`

Функции объекта `Graphics` позволяют изобразить простейшие геометрические элементы (примитивы) в области компонента `pictureBox`. Приведем основные функции создания графических примитивов.

```
void DrawLine(System::Drawing::Pen^ MyPen, int x1, int y1, int
x2, int y2);
%Проводит прямую линию карандашом MyPen из точки с координатами
(x1,y1) в точку (x2,y2).%

void DrawRectangle(System::Drawing::Pen^ MyPen, int x, int y,
int w, int h);
%Рисует периметр прямоугольника карандашом MyPen с левым верхним
углом в точке (x,y), шириной w и высотой h.%

void FillRectangle(System::Drawing::Brushes^ MyBrush, int x, int
y, int w, int h);
%Рисует прямоугольник, закрашенный кистью MyBrush, с левым
верхним углом в точке (x,y), шириной w и высотой h.%

void DrawEllipse(System::Drawing::Pen^ MyPen, int x, int y, int
w, int h);
%Рисует эллипс карандашом MyPen в прямоугольной области с левым
верхним углом в точке (x,y), шириной w и высотой h.%

void FillEllipse(System::Drawing::Brushes^ MyBrush, int x, int
y, int w, int h);
%Рисует эллипс, закрашенный кистью MyBrush, в прямоугольной
области с левым верхним углом в точке (x,y), шириной w и высотой
h.%
```

При рисовании линий и контуров фигур в функциях указывают карандаш. Он задает вид линии (цвет, толщину, стиль). Для закраски внутренних областей фигур используются кисти. Они определяют способ заполнения области (заливка, штриховка, замощение рисунком).

Карандаши и кисти можно создать самостоятельно, указав необходимые параметры стиля. Однако в некоторых случаях удобно использовать стандартные карандаши и кисти.

Стандартные карандаши изображают непрерывную линию, имеют толщину 1 пиксель и различаются только цветом. Так, например, инструкция `Pens::Red` означает, что выбран красный стандартный карандаш, а `Pens::Blue` - синий.

Стандартные кисти закрашивают область одним цветом: `Brushes::Green` – зеленым, `Brushes::Yellow` – желтым и т.д.

Полный список цветов можно увидеть в списке, открывающимся после печати оператора `::` или найти в литературе.

Пример 2. Использование графических примитивов и генератора случайных чисел

Создадим приложение, строящее диаграмму дискретного случайного процесса. В главное окно приложения поместим объект `pictureBox1`. Его размеры установим кратными некоторому числу n. В нашем случае $n = 10$.

Создадим функцию `pictureBox1_Paint` и напишем ее реализацию.

```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e)
{
     //Блок 1
     /*Получим размеры области pictureBox1*/
     int w = this->pictureBox1->Width;
     int h = this->pictureBox1->Height;
     /*Отступим от левого верхнего угла pictureBox1 и отметим
точку ltp (left top point) – левый верхний угол области
построения диаграммы. */
     Point ltp(20, 20);
     /*Отступим от правого нижнего угла pictureBox1 по 20
пикселей с каждого края и отметим точку rbp (right bottom point)
– правый нижний угол области построения диаграммы.*/
     Point rbp(w - 20,h - 20);
     /*Введем обозначения.*/
     int n = 10; //Число возможных исходов и количество опытов
     int hx = (rbp.X - ltp.X)/n; //Шаг по оси Ox
     int hy = (rbp.Y - ltp.Y)/n; //Шаг по оси Oy
     int rad = 5; //Радиус маркера диаграммы

     //Блок 2
     /*Закрасим область построения диаграммы белым цветом.*/
     e->Graphics->FillRectangle(Brushes::White, ltp.X, ltp.Y,
rbp.X - ltp.X, rbp.Y - ltp.Y);
     /*Нарисуем границу области.*/
     e->Graphics->DrawRectangle(Pens::Black, ltp.X, ltp.Y, rbp.X
- ltp.X, rbp.Y - ltp.Y);

     //Блок 3
     /*Передача некоторой величины генератору случайных чисел.*/
     srand(time(NULL));
     for(int i = 0; i <= n; i++)
     {
          /*Получим случайное число yi.*/
          double yi = rand() % n*hy;
          /*Построим линию высотой yi, начиная от нижнего края
области.*/
          e->Graphics->DrawLine(Pens::Blue, ltp.X+i*hx+hx/2,
rbp.Y, ltp.X+i*hx+hx/2, rbp.Y-yi);
          /*На конце линии изобразим круглый маркер.*/
          e->Graphics->FillEllipse(Brushes::Blue,
ltp.X+i*hx+hx/2-rad, rbp.Y-yi-rad, 2*rad, 2*rad);
     }
}
```

Для хранения координат левого верхнего (`ltp`) и правого нижнего (`rbp`) углов области построения диаграммы используется структура `Point`. При создании объекта типа `Point` вызывается конструктор с параметрами, в который передаются координаты точки. Доступ к координатам можно осуществить через имя объекта с помощью оператора `(.)`, т.е. `ltp.X`, `ltp.Y` или `rbp.X`, `rbp.Y`.

При создании объекта `pictureBox1` его размеры были выбраны кратными числу `n`. Область построения диаграммы также кратна этому числу. В таком случае операции

целочисленного деления (rbp.X - ltp.X)/n и (rbp.Y - ltp.Y)/n являются правомерными. Вся область диаграммы разбивается на ячейки одинакового размера. Если же имеется остаток от деления, то в построении могут возникать неточности.

В целях упрощения для указания числа возможных исходов и количества опытов используется одна постоянная n = 10

Генерация случайных чисел осуществляется с помощью функции rand(), хранящейся в файле stdlib.h, который должен быть подключен к коду посредством директивы #include. Эта функция позволяет получить случайное число в диапазоне от 0 до некоторой величины, указанной в файле stdlib.h. Для изменения диапазона служит инструкция вида: rand() % n, где n – количество возможных исходов.

Чтобы функция rand() при каждом запуске программы выдавала разные результаты, нужно передать генератору какое-нибудь число – отправную точку. Для этого используется функция srand. В качестве аргумента можно передать системное время с помощью функции time, прописанной в файле time.h. Инструкция time(NULL) означает, что возвращаемое время не сохраняется.

Вид диаграммы случайного процесса показан на рис. 9.

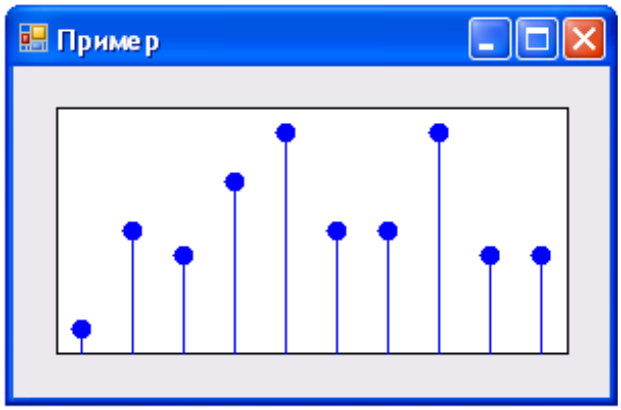


Рис. 9. Диаграмма случайного процесса, генерируемого компьютером

3. Создание карандашей

Для прорисовки линий, имеющих разнообразный вид, требуется создание карандаша. Карандаш является объектом типа System::Drawing::Pen. В функцию рисования передается указатель на объект такого типа.

При создании карандаша может использоваться один из конструкторов с параметрами. Самый простой конструктор требует указания цвета:

```
System::Drawing::Pen^ MyPen = gcnew
System::Drawing::Pen(Color::Blue);
```

В этом случае остальные параметры (свойства) карандаша можно задать, осуществив к ним доступ через указатель MyPen с помощью оператора (->).

Ширину изображаемой карандашом линии хранит свойство Width. Ширина линии измеряется в пикселях:

```
MyPen->Width = 2; /*Для карандаша MyPen установлена ширина линии
2 пикселя*/
```

Стиль линии содержит свойство DashStyle. Это свойство может принимать одно из значений, хранящихся в пространстве имен System::Drawing::Drawing2D::DashStyle. Список возможных значений приведен в табл. №8. Так, например, для прорисовки пунктирных линий карандашом MyPen требуется строка вида:

```
MyPen->DashStyle = System::Drawing::Drawing2D::DashStyle::Dash;
```

Табл. №8. Стили карандашей

Стиль	Описание стиля	Тип линии
`Solid`	Сплошная линия	
`Dash`	Пунктирная линия типа «тире»	
`Dot`	Пунктирная линия типа «точка»	
`DashDot`	Пунктирная линия типа «тире-точка»	
`DashDotDot`	Пунктирная линия типа «тире-точка-точка»	
`Custom`	Создание пунктирной линии с указанием длин штрихов и отступов между ними, которые передаются в свойство `DashPattern (MyPen->DashPattern)` в виде массива.	

Пример 3. Использование различных стилей карандаша

Внесем некоторые дополнения в пример 2. Разобьем область построения диаграммы по вертикали на уровни, соответствующие случайным значениям `yi` (см. пример 2), и изменим толщину границы области (рис. 10).

```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e)
{
     //Блок 1 - без изменений
     ...
     //Блок 2
     /*Создадим карандаш MyPen черного цвета толщиной 2 пикселя*/
     System::Drawing::Pen^ MyPen = gcnew
     System::Drawing::Pen(Color::Black, 2);
     /*Закрасим область построения диаграммы белым цветом.*/
     e->Graphics->FillRectangle(Brushes::White, ltp.X, ltp.Y,
rbp.X - ltp.X, rbp.Y - ltp.Y);
     /*Нарисуем карандашом MyPen границу области построения
диаграммы.*/
     e->Graphics->DrawRectangle(MyPen, ltp.X, ltp.Y, rbp.X -
ltp.X, rbp.Y - ltp.Y);
     /*Изменим параметры карандаша.*/
     /*Устанавливаем свойство рисования пунктирной линии.*/
     MyPen->DashStyle =
System::Drawing::Drawing2D::DashStyle::Dash;
     /*Устанавливаем толщину линии 1 пиксель.*/
     MyPen->Width = 1;
     /*Изображаем уровни диаграммы*/
     for(int i=1; i<n; i++)
          e->Graphics->DrawLine(MyPen, ltp.X, ltp.Y+i*hy, rbp.X,
ltp.Y+i*hy);
     //Блок 3 - без изменений
     ...
}
```

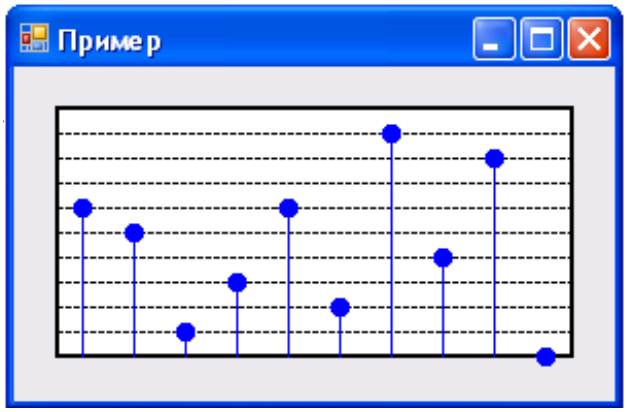


Рис. 10. Диаграмма случайного процесса с горизонтальными уровнями

```
/*Создадим карандаш MyPen черного цвета толщиной 2 пикселя*/
System::Drawing::Pen^ MyPen = gcnew
System::Drawing::Pen(Color::Black, 2);
```

Для создания карандаша `MyPen` в примере использовался конструктор с двумя параметрами (цвет и толщина линии). После прорисовки границ области диаграммы были изменены некоторые свойства объекта `MyPen`, и этим же карандашом были изображены уровни. В случаях, когда необходимо часто изменять тип линий, лучше использовать несколько различных карандашей.

4. Вывод текста

Вывод текста в область компонента `PictureBox` обеспечивает функция:

```
void DrawString(System::String^ str, System::Drawing::Font^
MyFont, System::Drawing::Brush^ MyBrush, int x, int y);
```

`str` – содержит строку типа `System::String^`,
`MyFont` – определяет шрифт выводимого текста,
`MyBrush` – задает цвет и способ заливки текста,
`x, y` – координаты левого верхнего угла области, в которую будет выведен текст.

Здесь появляется новый объект типа `System::Drawing::Font^`. В качестве аргумента `MyFont` в функцию можно передавать шрифт, который уже используется некоторым обектом интерфейса, например, главной формой:

```
e->Graphics->DrawString("Строка", this->Font, Brushes::Blue, 20,
20);
```

Создать свой шрифт позволяет, например, конструктор:

```
System::Drawing::Font(System::Drawing::FontFamily^ fstr, float
emSize, System::Drawing::FontStyle newStyle);
```

где `fstr` – строка, содержащая название стандартного семейства шрифтов,
`emSize` – ширина самой широкой буквы нового шрифта в пунктах (пт),
`newStyle` – задает стиль нового шрифта. Последний аргумент может принимать одно из значений, содержащихся в контейнере `FontStyle`. Список значений, отвечающих различным стилям шрифта, представлен в табл. №9.

Табл. №9. Стили шрифта

Название стиля	Описание
Regular	Стандартный текст
Bold	Текст, выделенный жирным шрифтом
Italic	Курсив
Underline	Текст с подчеркиванием снизу
Strikeout	Зачеркнутый текст

Так будет выглядеть создание жирного шрифта размером 20 пт семейства Tahoma:

```
System::Drawing::Font^ myFont = gcnew
System::Drawing::Font("Tahoma", 20, FontStyle::Bold);
```

Пример 4. Вывод текста

Дополним диаграмму из примера 2-3 числовыми подписями.

```
private: System::Void pictureBox1_Paint(System::Object^  sender,
System::Windows::Forms::PaintEventArgs^  e)
{
      //Блок 1 – без изменений
      ...
      //Блок 2 - без изменений
      ...

      for(int i=1; i<=n; i++)
      {
            /*Подпись по вертикали*/
            e->Graphics->DrawString(Convert::ToString(n-i),
Form::Font, Brushes::Black, ltp.X-15, ltp.Y+i*hy-hy/2);
            /*Подпись по горизонтали*/
            e->Graphics->DrawString(Convert::ToString(i),
Form::Font, Brushes::Black, ltp.X+i*hx-17, rbp.Y+5);
      }

      //Блок 3 – без изменений
      ...
}
```

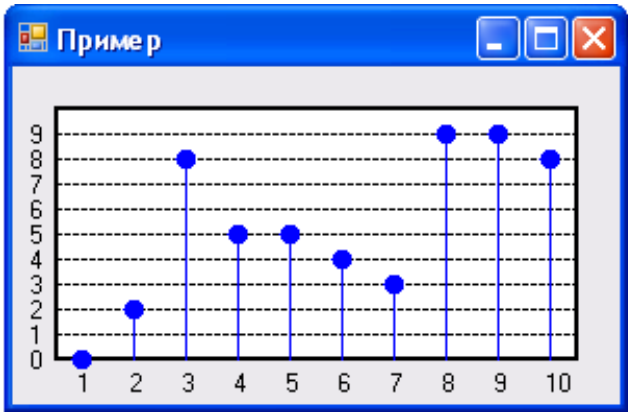


Рис. 11. Диаграмма случайного процесса с подписями по осям

В случаях, когда выводимые на экран значения типа `double` хранят большое число знаков после запятой, следует применять процедуру округления. В частности, иногда можно использовать простую схему:

```
double x = 4.528475;
double roundx = int(x*100+0.5)/100.0;/*Округление до сотых*/
```

Инструкция вида `int(x*100+0.5)` осуществляет явное преобразование типа в тип `int`. Прибавление к числу `x*100` десятичной дроби `0.5` позволяет округлить число в большую сторону, если в третьем разряде после запятой будет стоять число `a`, большее или равное `5` (`5,6,7,8,9`). Заметим, что с точки зрения правил округления результат будет не всегда верным (случай `a = 5`).

После явного приведения типа полученное число делим на `100.0`. Результатом деления будет число `roundx` типа `double`, равное `4.53`.

ГЛАВА 3. Практические задания

1. Построение графиков функций

Задание 1

1. Выбрать функцию для построения графика и реализовать контроль ее области определения. Инициализировать глобальные переменные: `double a, b` – левый и правый концы промежутка построения графика, `int n` – количество точек, по которым строится график.
2. Расположить на форме поля ввода чисел `a,b,n` и компонент `PictureBox`. Создать кнопку, после нажатия которой происходит считывание данных и построение графика. В функции обработки события `Click` осуществить конвертацию введенных данных, контроль правильности ввода и при отсутствии ошибок вызвать функцию обработки события `Paint` компонента `PictureBox`.
3. В функции обработки события `Paint` реализовать прорисовку координатной сетки, графика функции и числовых значений координатных осей.

Пояснения. Линейное преобразование координат

Последний пункт нуждается в уточнении.

График функции должен быть построен на отрезке $[a,b]$ и иметь множество значений

$$[c,d] = \left[\min_{[a,b]} f(x)\ , \max_{[a,b]} f(x)\right].$$

С помощью алгоритмов поиска минимума и максимума определите границы множества значений функции на отрезке $[a,b]$. Таким образом, график функции будет построен в прямоугольнике $[a,b]\times[c,d]$ «изображаемой» системы координат.

Для хранения координат точек графика введем два массива `x, y`. Изменяя значение абсциссы `x[i]`, можно определить значение ординаты `y[i]`:

```
for(int i = 0; i <= n; i++)
{
     x[i] = a + i*(b-a)/n;
     y[i] = f(x[i]);
}
```

После того, как массивы `x, y` будут заполнены, нужно осуществить переход от «изображаемой» системы координат к «фактической», связанной с компонентом PictureBox. Этот переход выполняется с помощью линейного преобразования.

В соответствии с ранее введенными обозначениями, область построения графика ограничивалась прямоугольником с левой верхней `(Point ltp)` и правой нижней `(Point rbp)` точками плоскости PictureBox. Для удобства дальнейшего изложения введем некоторые обозначения:

```
int apb = ltp.X;
int bpb = rbp.X;
int cpb = rbp.Y;
int dpb = ltp.Y;
```

Установим взаимно однозначное соответствие между отрезками $[a,b]$ и $[apb, bpb]$. А именно, преобразуем отрезок $[a,b]$ в $[apb, bpb]$ таким образом, чтобы точка a перешла в apb, а точка b в bpb (рис. 12). Тогда внутренняя точка x перейдет в некоторую точку xpb. Будем искать закон соответствия в виде линейной функции:

$$xpb = k_1 x + k_2, \qquad (1)$$

где k_1 и k_2 - неизвестные коэффициенты.

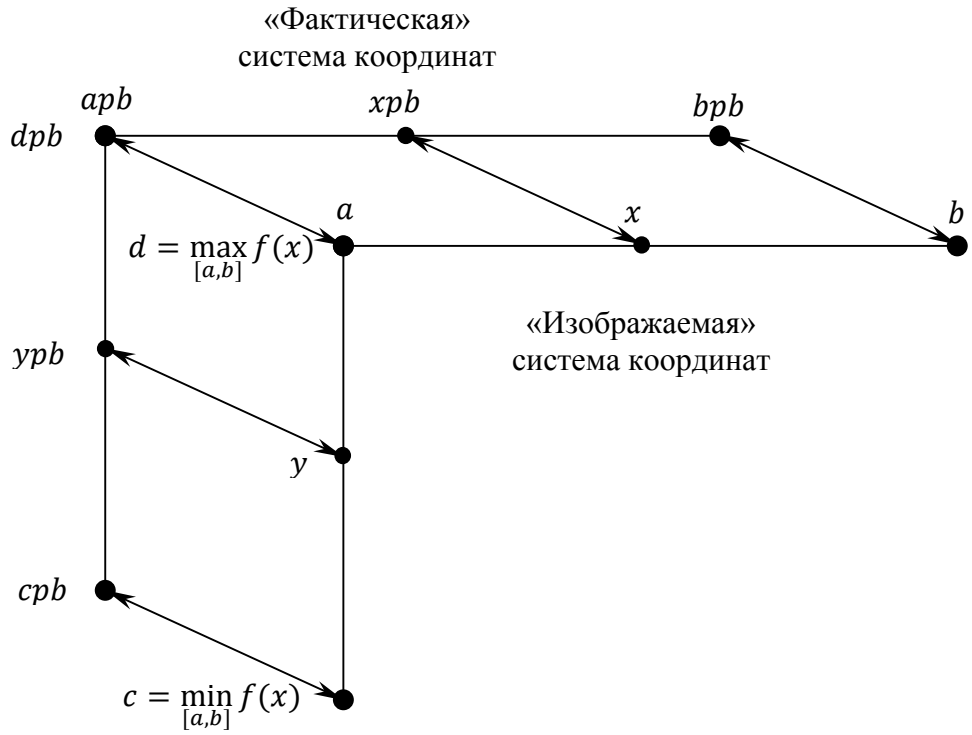


Рис. 12. Взаимно однозначное соответствие систем координат

Закон (1) должен удовлетворять условиям соответствия концов отрезков:

$$apb = k_1 a + k_2,$$
$$bpb = k_1 b + k_2.$$

Из этих условий можно определить значения неизвестных коэффициентов k_1 и k_2:

$$k_1 = \frac{bpb - apb}{b - a}, \qquad k_2 = apb - a\frac{bpb - apb}{b - a}.$$

Аналогично производится преобразование отрезка $[c, d]$ в $[cpb, dpb]$. Закон соответствия будет иметь вид:

$$ypb = k_3 y + k_4,$$

где

$$k_3 = \frac{dpb - cpb}{d - c}, \qquad k_4 = cpb - c\frac{dpb - cpb}{d - c}.$$

По «изображаемым» координатам вычисляются координаты точек графика в области компонента PictureBox:

```
/*Инициализация переменных k1, k2, k3, k4*/
for(int i = 0; i <= n; i++)
{
     xpb[i] = k1*x[i]+k2;
     ypb[i] = k3*y[i]+k4;
}
```

Пример 5. Выполнение лабораторной работы по теме «Построение графиков функций»

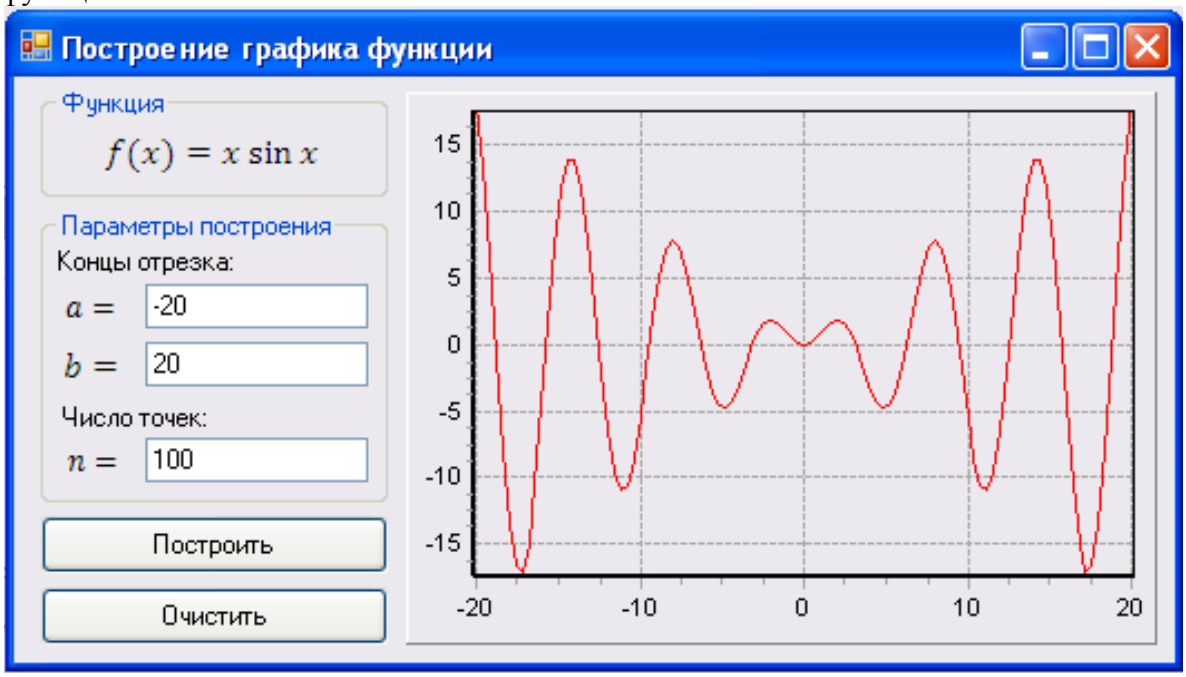


Рис. 13. Интерфейс программы

2. Аффинные преобразования координат

Теоретическая справка

Определение. Аффинным преобразованием координат называется линейное преобразование $f: \mathbb{R}^n \to \mathbb{R}^n$ вида

$$f(x) = Ax + b,$$

где $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{n}} \times \mathbb{R}^{\boldsymbol{n}}$ – невырожденная матрица, $\boldsymbol{b} \in \mathbb{R}^{\boldsymbol{n}}$ – вектор переноса начала координат.

Свойства аффинных преобразований в пространстве:

1) Образом прямой является прямая, образом плоскости – плоскость,
2) Сохраняет параллельность прямых и плоскостей,
3) Сохраняет пропорции параллельных объектов,
4) Обладают свойством взаимной однозначности.

Различают четыре вида простейших (элементарных) аффинных преобразований координат:

1) Поворот,
2) Растяжение/сжатие,
3) Отражение (частный случай поворота),
4) Перенос.

Теорема. Любое аффинное преобразование является суперпозицией элементарных преобразований.

Рассмотрим аффинное преобразование точек плоскости. Из определения следует, что координаты точки при изменении системы координат вычисляются по формулам:

$$\begin{cases} x^* = \alpha x + \beta y + \lambda, \\ y^* = \gamma x + \delta y + \mu, \end{cases} \qquad (2)$$

где $\begin{vmatrix} \alpha & \beta \\ \gamma & \delta \end{vmatrix} \neq 0, \alpha, \beta, \gamma, \delta, \lambda, \mu$ – некоторые числа.

Поясним это на примере. Выберем систему координат Oxy и зафиксируем в ней точку $M(x,y)$. Продублируем оси системы Oxy и вмороженную в нее точку. Для дублированных объектов введем новые обозначения: $O^*x^*y^*$, M^*. Выполним поворот $O^*x^*y^*$ относительно точки O на некоторый угол, затем перенос системы $O^*x^*y^*$ и растяжение вдоль осей Ox, Oy (рис. 14). При этом координаты точки M^* в системе $O^*x^*y^*$ всегда остаются равными (x,y), а координаты точки M^* в исходной системе Oxy $-(x^*,y^*)$ могут быть вычислены по формуле (1), в которой числа $\alpha,\beta,\gamma,\delta,\lambda,\mu$ имеют смысл, показанный на рис. 15. Отметим, что ортогональность системы $O^*x^*y^*$ **в общем случае не сохраняется**.

Формулы пересчета координат точки плоскости при элементарных преобразованиях, представлены в табл. №10.

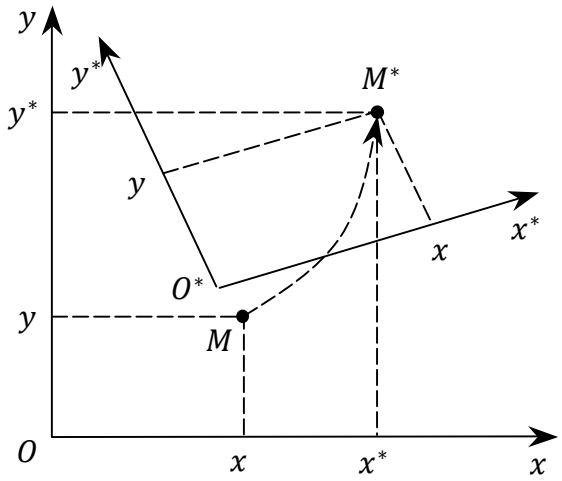


Рис. 14. Изменение координат точки после аффинного преобразования

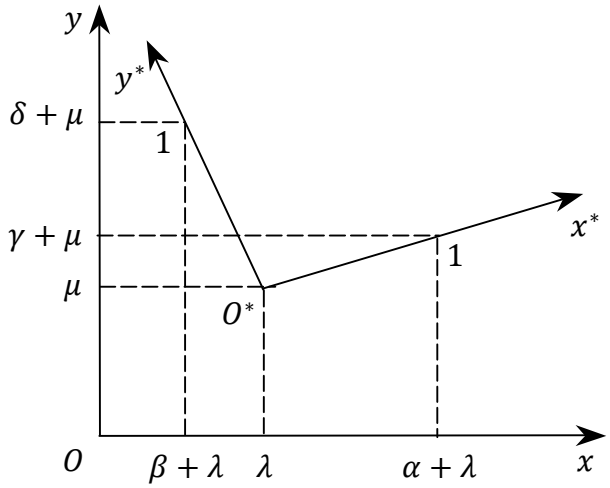
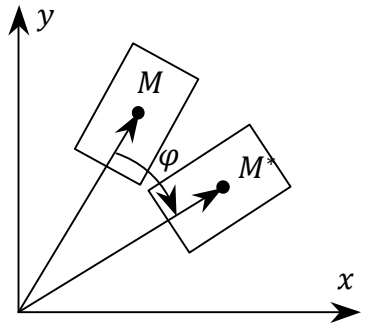
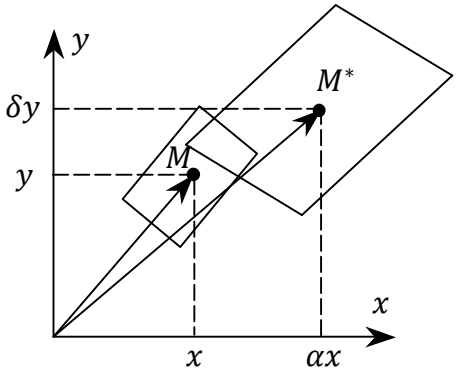
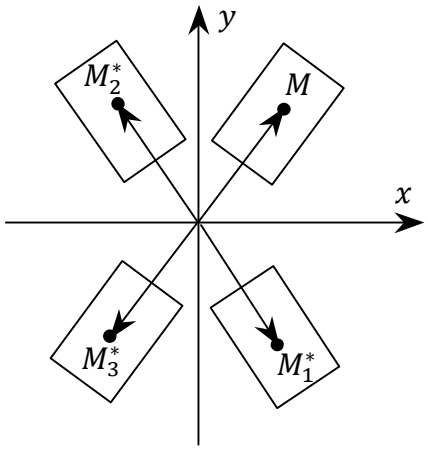
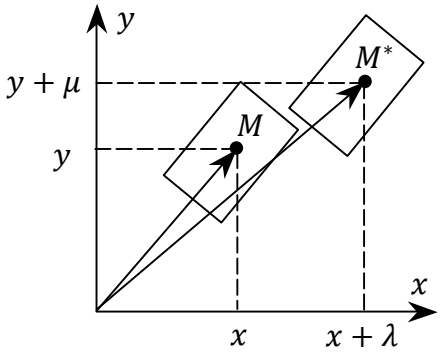


Рис. 15. Геометрический смысл коэффициентов системы (2)

Табл. №10. Элементарные аффинные преобразования

№	Преобразование	Формулы пересчета	Иллюстрация
1	Поворот	$\begin{cases} x^* = x \cdot cos\varphi - y \cdot sin\varphi, \\ y^* = x \cdot sin\varphi + y \cdot cos\varphi, \end{cases}$ $(x^*, y^*) = (x, y) \cdot \begin{bmatrix} cos\varphi & sin\varphi \\ -sin\varphi & cos\varphi \end{bmatrix}.$	
2	Растяжение/ сжатие	$\begin{cases} x^* = \alpha \cdot x, \\ y^* = \delta \cdot y, \end{cases}$ $(x^*, y^*) = (x, y) \cdot \begin{bmatrix} \alpha & 0 \\ 0 & \delta \end{bmatrix}.$ **Растяжение:** вдоль оси Ox при $\alpha > 1$, вдоль оси Oy при $\delta > 1$. **Сжатие:** вдоль оси Ox при $0 < \alpha < 1$, вдоль оси Oy при $0 < \delta < 1$.	
3	Отражение	1) Относительно оси Ox: $\begin{cases} x^* = x, \\ y^* = -y, \end{cases}$ $(x^*, y^*) = (x, y) \cdot \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$ 2) Относительно оси Oy: $\begin{cases} x^* = -x, \\ y^* = y, \end{cases}$ $(x^*, y^*) = (x, y) \cdot \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}.$ 3) Относительно начала координат: $\begin{cases} x^* = -x, \\ y^* = -y, \end{cases}$ $(x^*, y^*) = (x, y) \cdot \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}.$	
4	Перенос (сдвиг)	$\begin{cases} x^* = x + \lambda, \\ y^* = y + \mu, \end{cases}$ $(x^*, y^*) = (x, y) + (\lambda, \mu).$	

Из табл. №9 видно, что каждое преобразование кроме переноса можно осуществить, умножив справа вектор-строку (x, y) на матрицу перехода, соответствующую этому преобразованию. Более универсальным является подход, при котором любому аффинному преобразованию соответствует матрица перехода одной размерности, а новые координаты получаются путем однообразных действий с матрицами перехода, какое бы преобразование ни требовалось. Такой подход требует введения некоторых понятий.

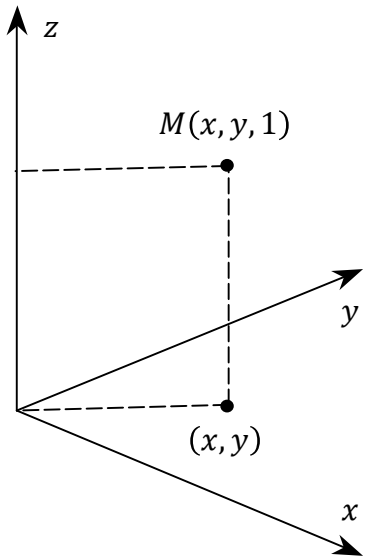


Рис. 16. Геометрический смысл однородных координат

Определение. Однородными координатами точки (x, y) называется любая тройка одновременно не равных нулю чисел x_1, x_2, x_3, которая связана с исходными координатами следующим образом:

$$\frac{x_1}{x_3} = x, \qquad \frac{x_2}{x_3} = y.$$

Получаем переход от двумерного пространства к трехмерному:

$$(x, y) \to (x_1, x_2, x_3) = (x_3 x,\ x_3 y,\ x_3).$$

Не нарушая общности рассуждений, выбираем $x_3 = 1$ (рис. 16), т.е.

$$(x, y) \to (x, y, 1).$$

Теперь любое преобразование, включая сдвиг, можно осуществить, умножив справа вектор-строку $(x, y, 1)$ на одну из матриц перехода:

$$(x^*, y^*, 1) = (x, y, 1) \cdot [\dots];$$

Запишем матрицы перехода, соответствующие элементарным преобразованиям:

1. Матрица поворота $[R]$:

$$[R] = \begin{bmatrix} cos\varphi & sin\varphi & 0 \\ -sin\varphi & cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix};$$

2. Матрица растяжения $[D]$:

$$[D] = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{bmatrix};$$

3. Матрицы отражения относительно координатных осей $[M_x], [M_y], [M_{x,y}]$:

$$[M_x] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad [M_y] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad [M_{x,y}] = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix};$$

4. Матрица переноса $[T]$:

$$[T] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda & \mu & 1 \end{bmatrix}.$$

Пример 6. Комбинация элементарных преобразований

Построим матрицу поворота вокруг точки A(a, b) на угол φ. Для эттого осуществим преобразование координат в виде последовательности элементарных преобразований и запишем соответствующие матрицы перехода.

1) Сдвиг начала координат в точку A:

$$[T_{-A}] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 0 \end{bmatrix}.$$

При этом начало старой системы переходит в точку $A^*(-a,-b)$ новой системы координат.

2) Поворот на угол φ относительно начала сдвинутой системы координат (точки A):

$$[R_\varphi] = \begin{bmatrix} cos\varphi & sin\varphi & 0 \\ -sin\varphi & cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix};$$

3) Сдвиг начала координат в точку $A^*(-a,-b)$:

$$[T_A] = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix}.$$

Матрица поворота на угол φ относительно точки A(a, b) получается в результате последовательного перемножения матриц элементарных преобразований:

$$[T_{-A}] \cdot [R_\varphi] \cdot [T_A] = \begin{bmatrix} cos\varphi & sin\varphi & 0 \\ -sin\varphi & cos\varphi & 0 \\ -acos\varphi + bsin\varphi & -asin\varphi + bcos\varphi & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{bmatrix} =$$

$$= \begin{bmatrix} cos\varphi & sin\varphi & 0 \\ -sin\varphi & cos\varphi & 0 \\ -acos\varphi + bsin\varphi + a & -asin\varphi + bcos\varphi + b & 1 \end{bmatrix};$$

Аффинные преобразования в пространстве выполняются аналогично. Запишем матрицы перехода, соответствующие элементарным преобразованиям:

1. Матрицы вращения
   1) В плоскости Oyz:

$$[R_x] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\varphi & sin\varphi & 0 \\ 0 & -sin\varphi & cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

   2) В плоскости Oxz:

$$[R_y] = \begin{bmatrix} cos\psi & 0 & -sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ sin\psi & 0 & cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

   3) В плоскости Oxy:

$$[R_z] = \begin{bmatrix} cos\chi & sin\chi & 0 & 0 \\ -sin\chi & cos\chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

2. Матрица растяжения:

$$[D] = \begin{bmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

3. Матрицы отражения
   1) Относительно плоскости Oyz

$$[M_x] = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

2) Относительно плоскости Oxz

$$[M_y] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

3) Относительно плоскости Oxy

$$[M_z] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix};$$

4. Матрица переноса:

$$[T] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \lambda & \mu & \nu & 1 \end{bmatrix}.$$

Задание 2

Написать программу, позволяющую реализовать аффинное преобразование плоской фигуры в виде последовательности элементарных преобразований: поворота, растяжения/сжатия, отражения, сдвига. В области PictureBox изобразить координатные оси и произвольный рисунок. Расположить на форме поля ввода параметров элементарных преобразований и командные кнопки (рис. 17). После нажатия кнопки, отвечающей определенному преобразованию, рисунок должен перестраиваться (рис. 18-22).

Пример 7. Выполнение лабораторной работы по теме «Аффинные преобразования»

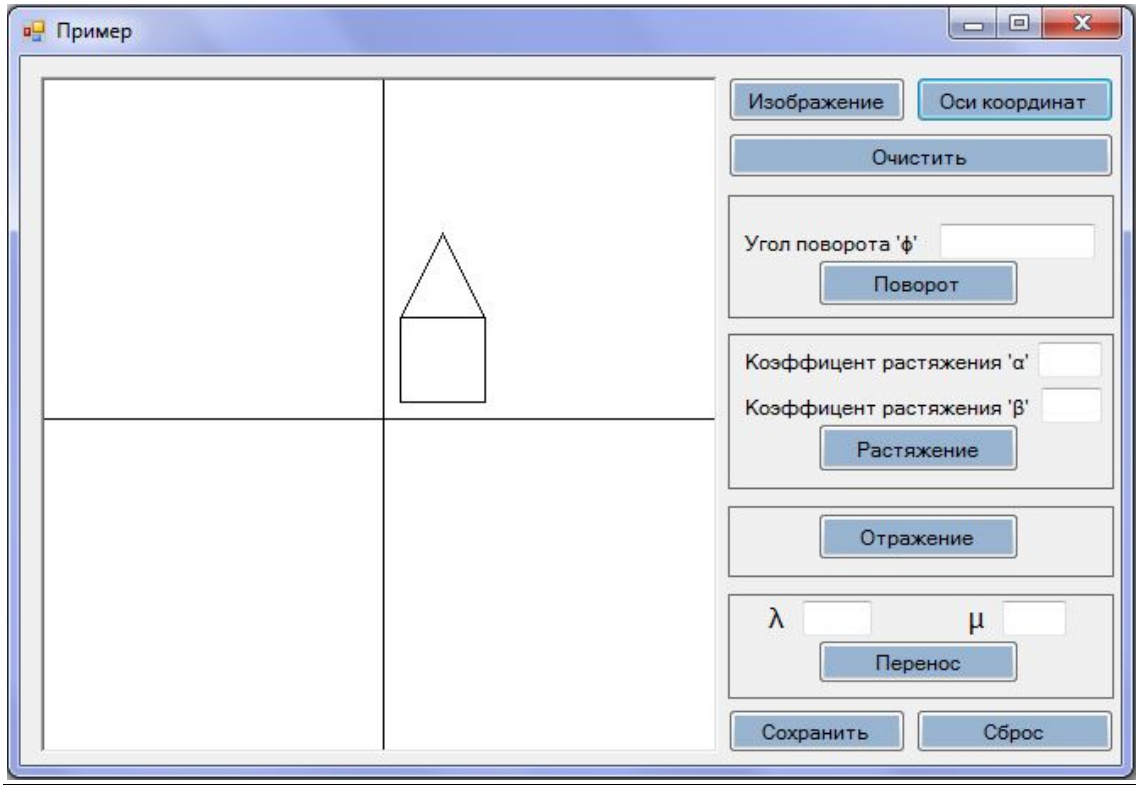


Рис. 17. Интерфейс программы

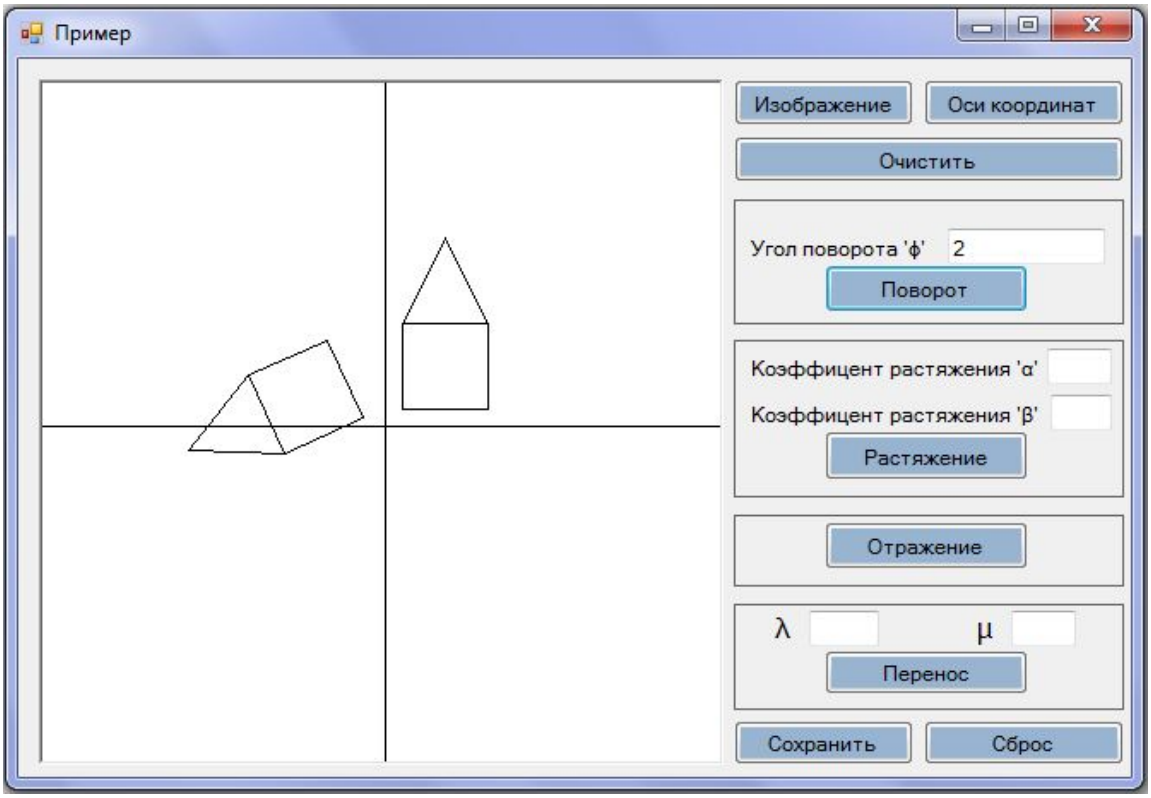


Рис. 18. Поворот на угол $\varphi = 2$ рад

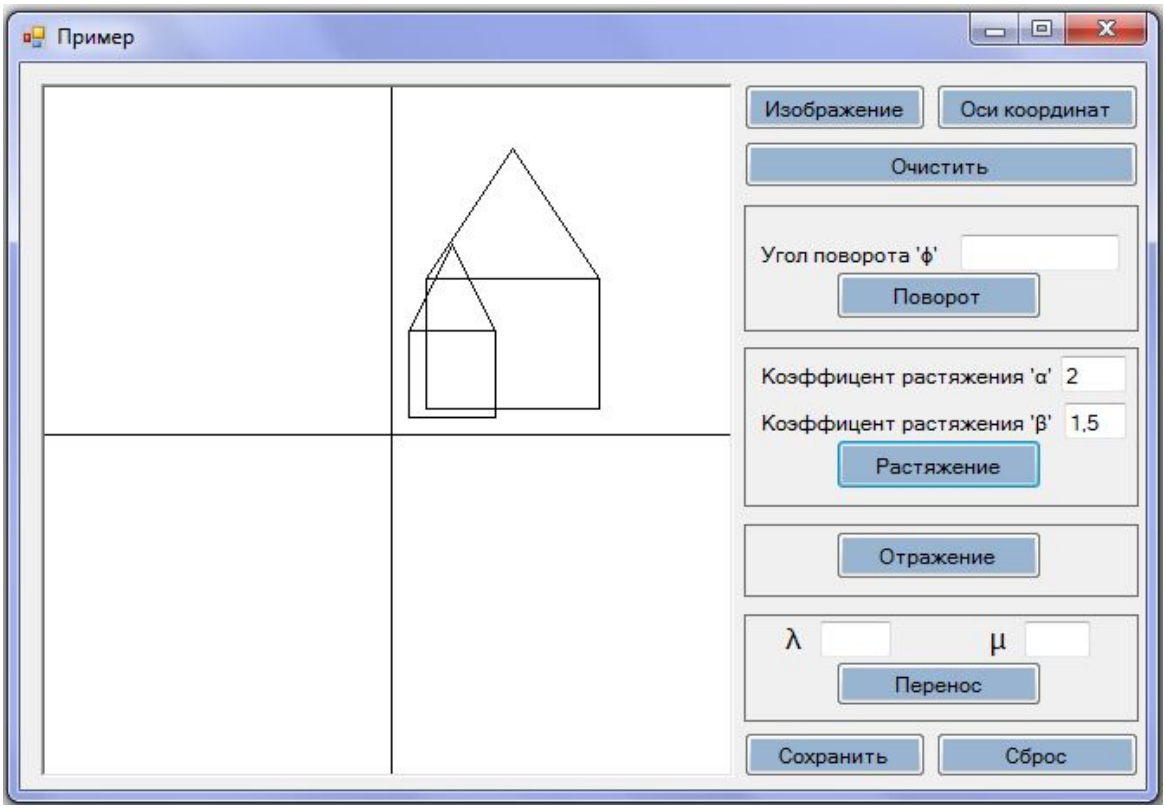


Рис. 19. Растяжение вдоль координатных осей с коэффициентами $\alpha = 2, \beta = 1{,}5$

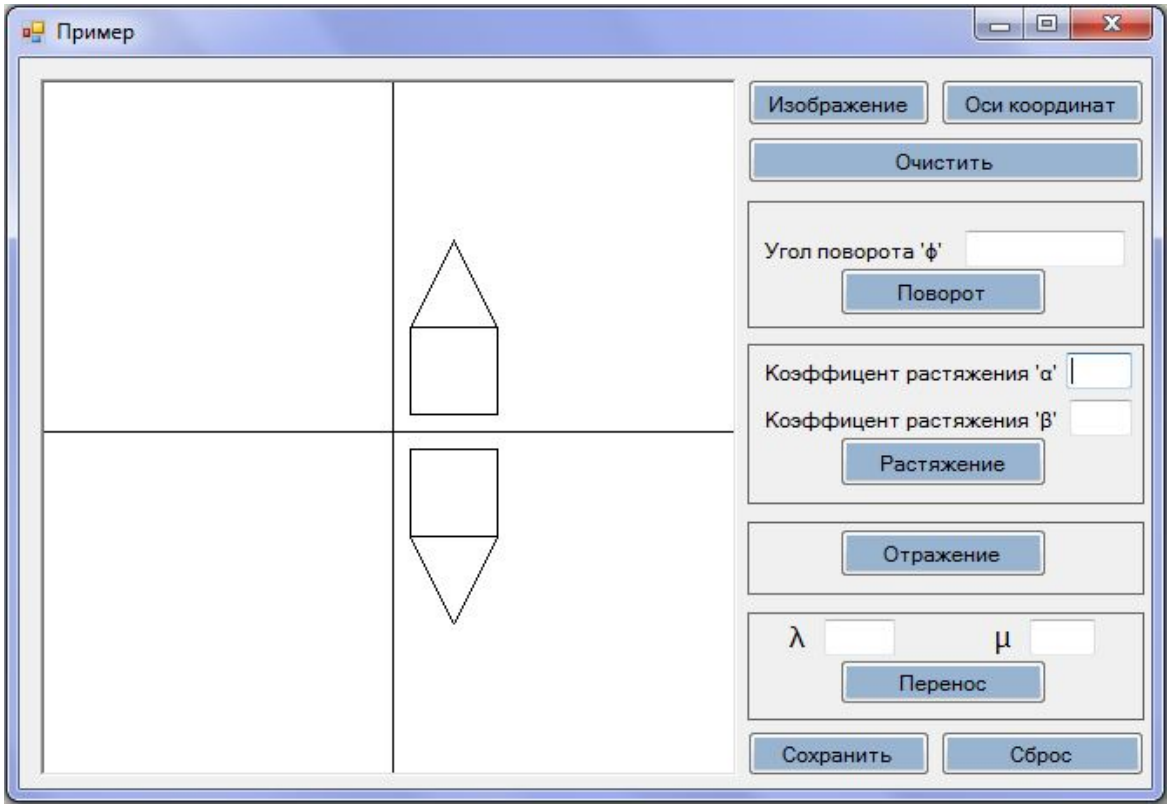


Рис. 20. Отражение относительно оси абсцисс

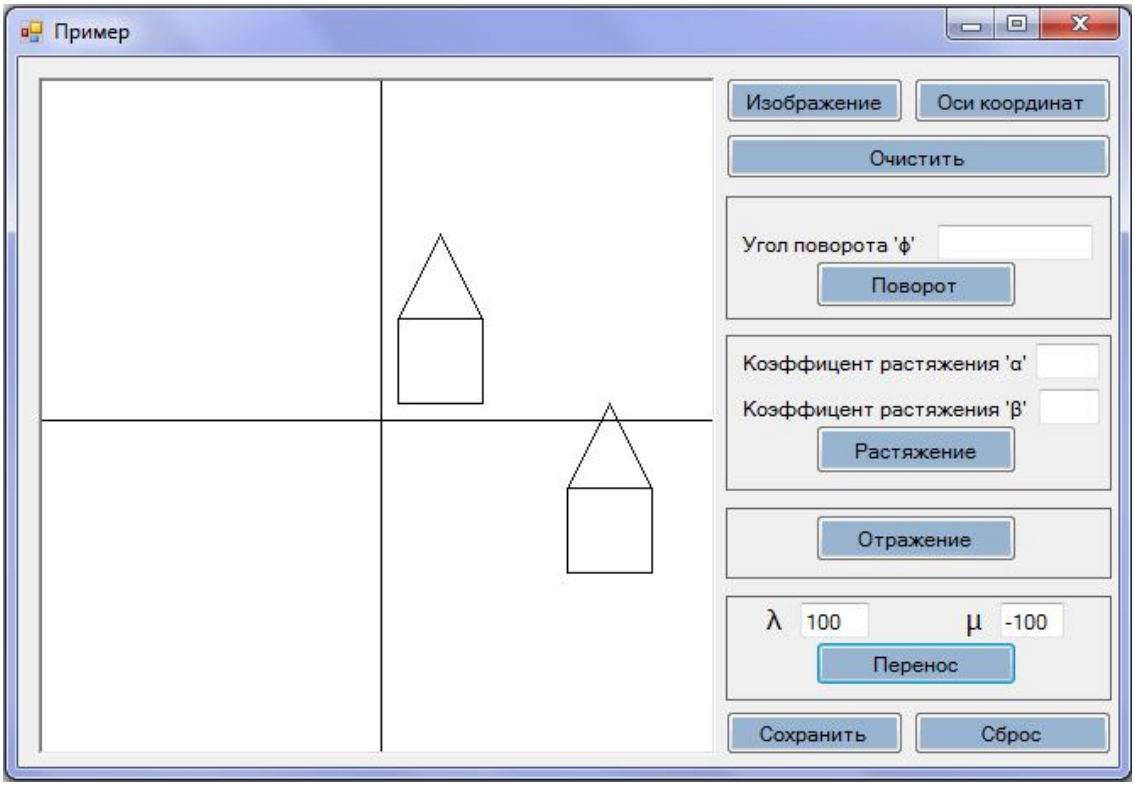


Рис. 21. Перенос с коэффициентами $\lambda = 100$, $\mu = -100$

Выполним поворот на угол $\varphi = 3$ рад, далее нажмем «Сохранить» и, задав параметры $\alpha = 2$ и $\beta = 2$, выполним растяжение вдоль координатных осей.

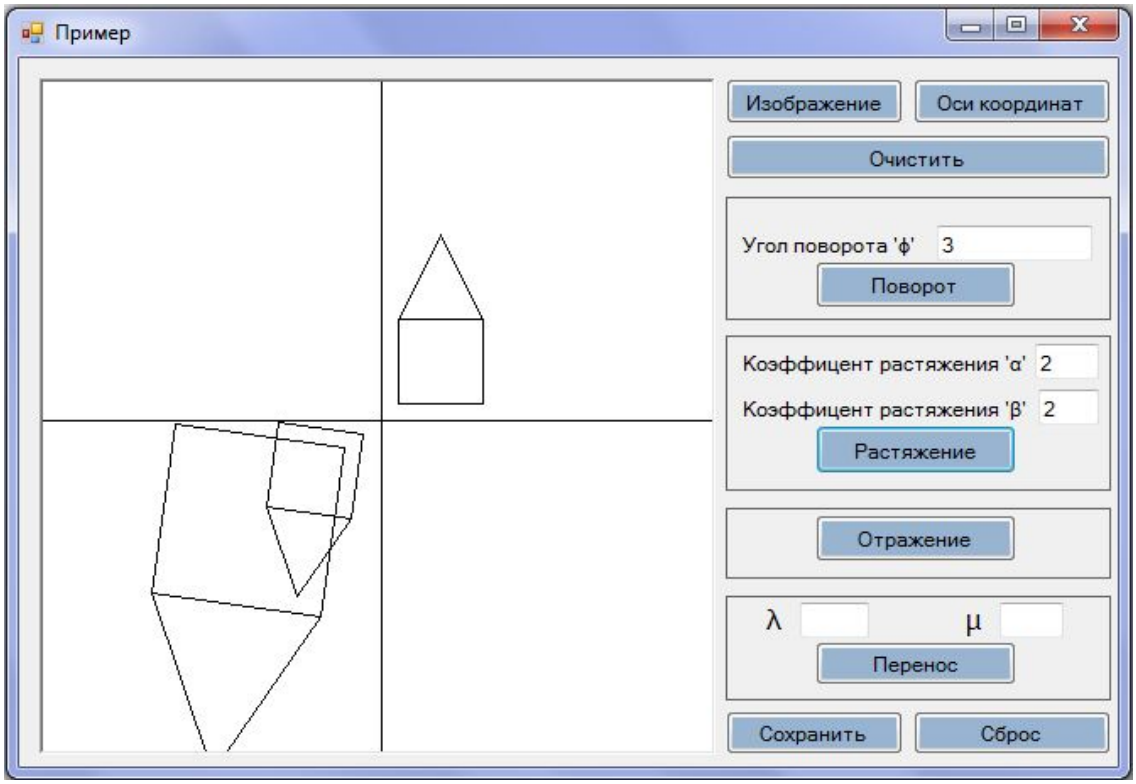


Рис. 22. Аффинное преобразование плоской фигуры как последовательность элементарных преобразований

Литература

1. Шикин Е.В., Боресков А.В. Компьютерная графика. М. «Диалог-МИФИ». 1995.
2. Культин Н.Б. Основы программирования в Microsoft Visual C++ 2010. — СПб.: БХВ-Петербург, 2010. — 384 с.
3. Интернет ресурс http://msdn.microsoft.com/ru-ru/library/