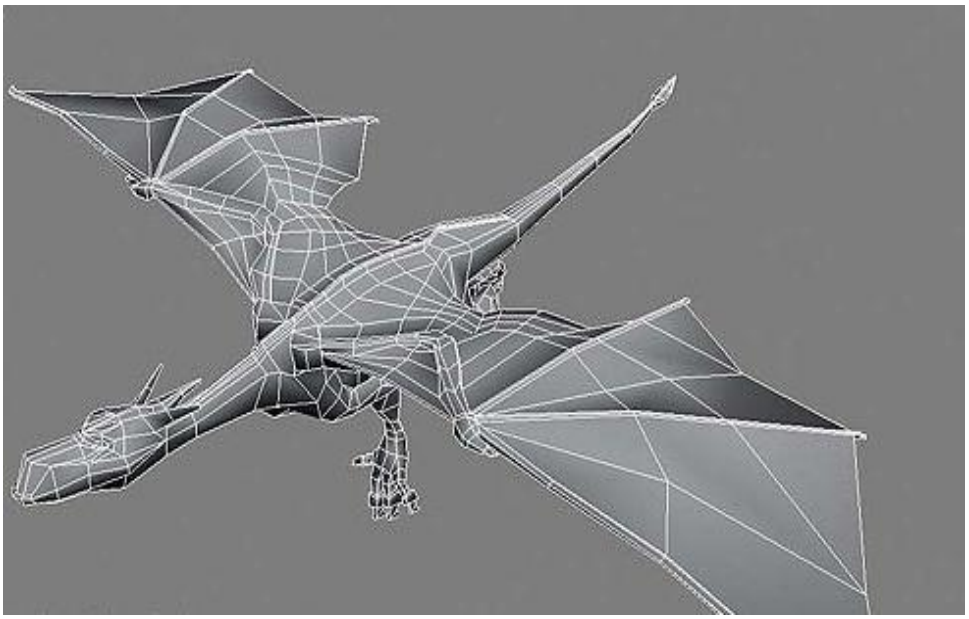
ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ
СЕВЕРО-КАВКАЗСКИЙ ФИЛИАЛ
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО ОБРАЗОВАТЕЛЬНОГО БЮДЖЕТНОГО
УЧРЕЖДЕНИЯ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
МОСКОВСКОГО ТЕХНИЧЕСКОГО УНИВЕРСИТЕТА СВЯЗИ И ИНФОРМАТИКИ

Дизайн графических и пользовательских интерфейсов

Методическое пособие по практическим занятиям



Ростов-на-Дону
2019

УДК 004.925

Дизайн графических и пользовательских интерфейсов . Методическое пособие по выполнению лабораторных работ . / Моск. техн. ун-т связи и информатики, Сев.-Кавк. филиал. – Ростов н/Д, 2019, 61 с.

В пособии даются организационно-методические указания на практические занятия, приводятся достаточно подробная методика выполнения заданий и порядок выполнения и оформления отчёта.

Предназначено для студентов обеих форм обучения, изучающих дисциплину «Дизайн графических и пользовательских интерфейсов », а также может быть полезно всем остальным студентам, желающим самостоятельно освоить программирование графики.

Обсуждено и утверждено на заседании кафедры ИВТ (протокол заседания кафедры №1 от 26.08.2019).

Содержание

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №1. ИЗУЧЕНИЕ ВИДА, СТРУКТУРЫ РАБОЧЕГО СТОЛА ЛАЗАРУС И НАЗНАЧЕНИЕ ОСНОВНЫХ УПРАВЛЯЮЩИХ ЭЛЕМЕНТОВ

Цель работы: освоить методы создания графического оконного интерфейса при помощи IDE Lazarus.

Lazarus - это **IDE** (*Integrated Development Environment*) - Интегрированная Среда Разработки программ, использующая компилятор **FPC** (*Free Pascal Compiler*), редакторы кода, форм, Инспектор Объектов, отладчик и многие другие инструменты.

Еще говорят, что среда **Lazarus** - это **RAD** (*Rapid Application Development*) - среда Быстрой Разработки Приложений.

До сих пор среды разработки программ, подобные **Lazarus**, были исключительно платными. **Lazarus** же стал первой (и пока единственной) IDE, доступной образовательным и государственным учреждениям совершенно бесплатно. Более того, **Lazarus** является проектом **Open Source** - проектом с открытым исходным кодом. Многие программисты по всему миру принимают участие в его развитии, исходный код **Lazarus** доступен для изучения и модификации. **Lazarus** имеет поддержку множества языков, в том числе и русского, что выгодно отличает его от других IDE.

Lazarus, как уже говорилось, - бесплатный и свободно распространяемый продукт. Благодаря этому, **Lazarus** все чаще используют для изучения программирования в школах и ВУЗах, а также на многих предприятиях. Но где его взять? На официальном сайте производителя: http://lazarus.freepascal.org

В правой верхней части сайта вы увидите следующую картинку:

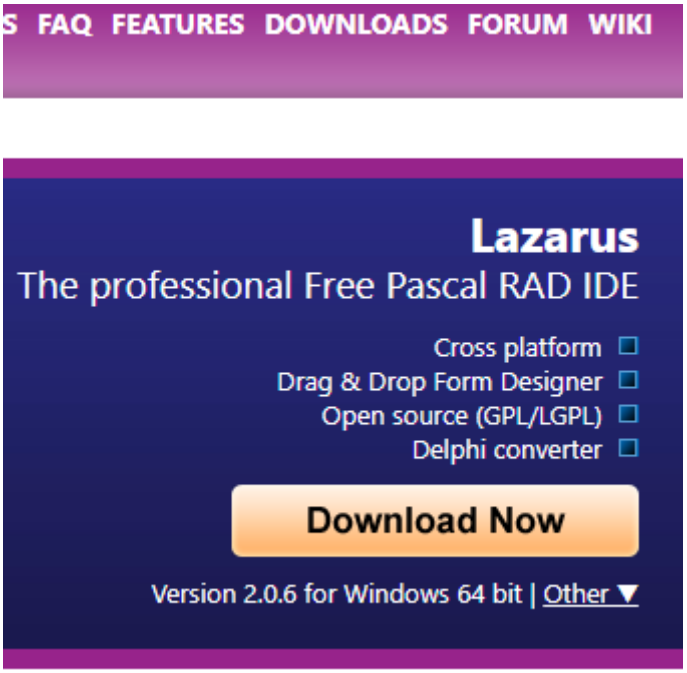


Рис. 1.1. Выбор и закачка необходимой реализации

Здесь вы сможете выбрать реализацию именно под вашу платформу, от Windows до Mac OS X, как 32-х так и 64-х разрядную. При написании курса использовался 32-х разрядный **Lazarus** для платформы Windows.

Нажав кнопку "**Download Now**" вы скачаете последнюю версию **Lazarus**. Кроме того, выбрать последнюю необходимую реализацию и скачать ее вы можете по адресу: http://sourceforge.net/projects/lazarus/files/

В этом случае, перейдя по ссылкам, вы получите доступ к закачке нескольких файлов, например,

- *lazarus-1.0.10-fpc-2.6.2-win32.exe*
- *lazarus-1.0.10-fpc-2.6.2-cross-arm-wince-win32.exe*
- *README.txt*

Нам нужен только первый файл из этого списка. Второй файл является расширением для разработки программ под **Windows CE** (она же **WinCE**) - это вариант операционной системы Microsoft Windows для наладонных

компьютеров, смартфонов и встраиваемых систем. На данном курсе эту возможность мы рассматривать не будем. Последний файл - простой текстовый файл с информацией о версии, он нам тоже не нужен.

Lazarus устанавливается достаточно просто. Собственно, ничего менять нам не придется, оставим все параметры, предложенные установщиком по умолчанию. Для начала выберем русский язык установки, затем все время будем нажимать кнопки "**Далее**". Лишь в предпоследнем окне установщика при желании можно поставить флажок "**Создать значок на Рабочем столе**". Когда укажем все параметры, начнется установка **Lazarus**. Придется подождать пару минут, пока распакуются и скопируются множество файлов. И, наконец, кнопка "**Завершить**" для закрытия окна установщика. Все, **Lazarus** у нас есть! Мы можем его загрузить.

В самом начале **Lazarus** выглядит несколько неопрятно:

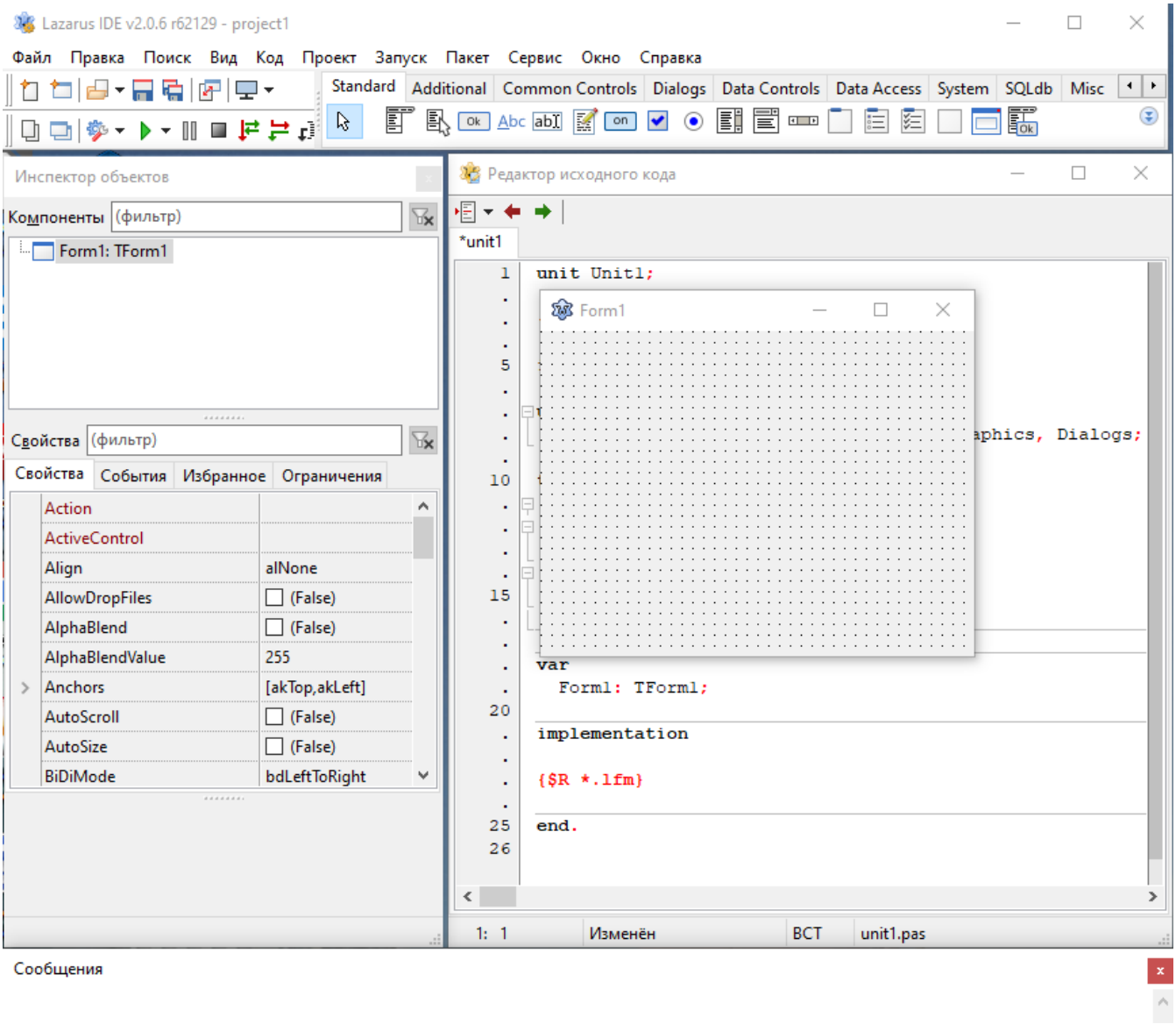


Рис. 1.2. Первый запуск Lazarus**Lazarus** состоит из нескольких окон (которые стоит подравнять, чтобы они занимали весь рабочий стол и не мешали друг другу):

1. **Главное окно**
2. **Инспектор объектов**
3. **Редактор форм**
4. **Редактор кода**
5. **Окно сообщений**

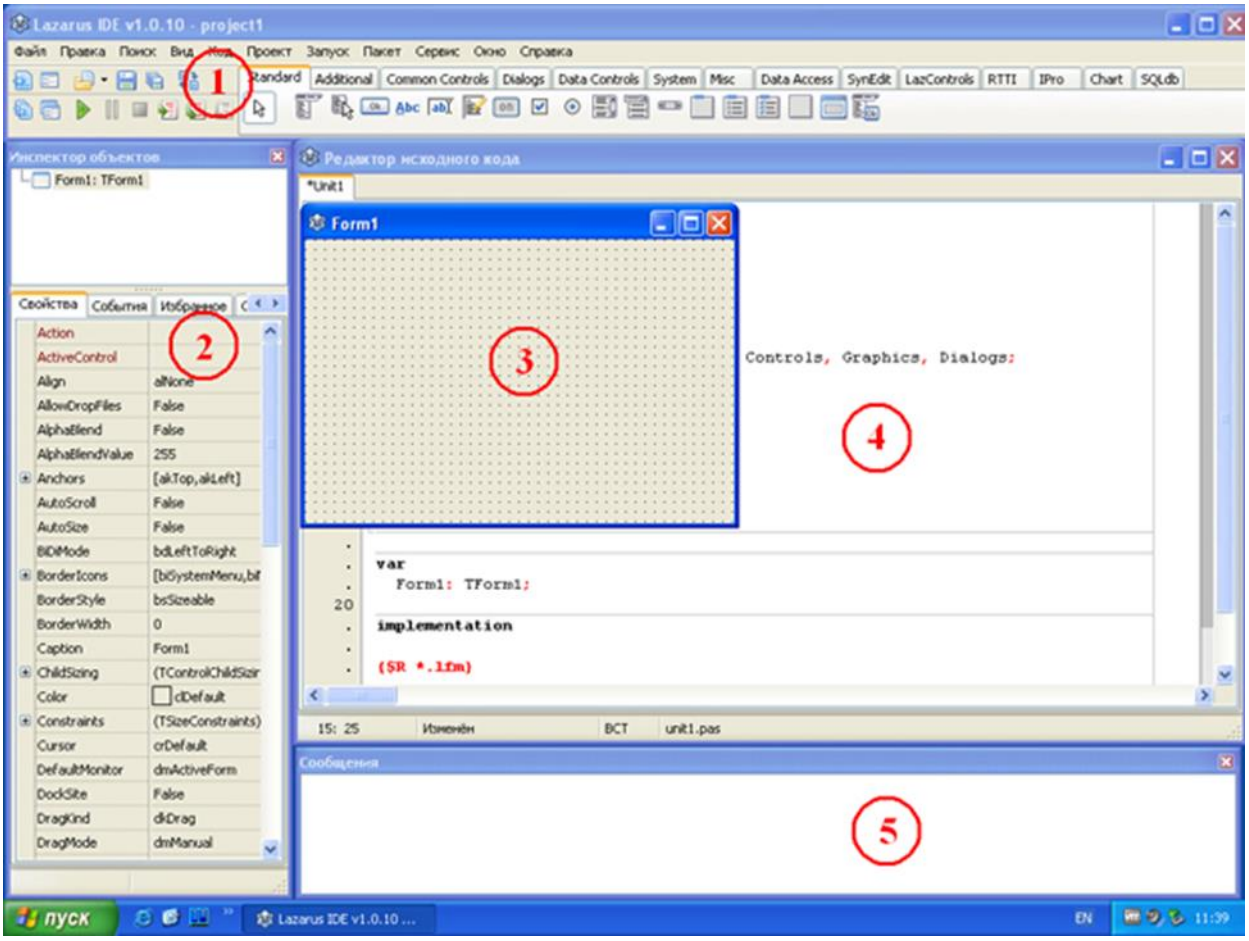


Рис. 1.3. Окна Lazarus

Главное окно

Главное окно состоит из следующих элементов:

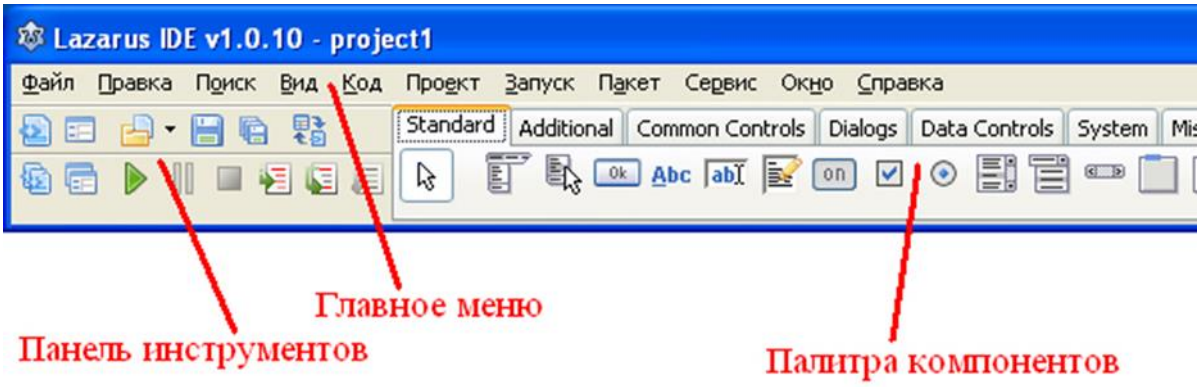


Рис. 1.4. Главное окно Lazarus

1. **Главное меню** содержит все команды, необходимые для правки,

компиляции, отладки программы, для запуска различных вспомогательных утилит.

2. **Панель инструментов** содержит кнопки чаще всего применяемых команд (эти же команды можно выполнить и с помощью **Главного меню**).

3. **Палитра компонентов** содержит множество вкладок, на которых содержится богатый выбор компонентов из собственной библиотеки компонентов **Lazarus - LCL** (Lazarus Component Library).

Инспектор объектов

Окно **Инспектора объектов** состоит из двух частей:

- **Дерево объектов**, в котором в древовидной форме располагаются все объекты, используемые в текущей форме.

- **Окно с вкладками**, в котором можно настраивать различные свойства текущего объекта. Несмотря на то, что имеется 4 вкладки (**Свойства**, **События**, **Избранное**, **Ограничения**), чаще всего используются только первые две. О свойствах и событиях мы поговорим подробней в следующих лекциях.

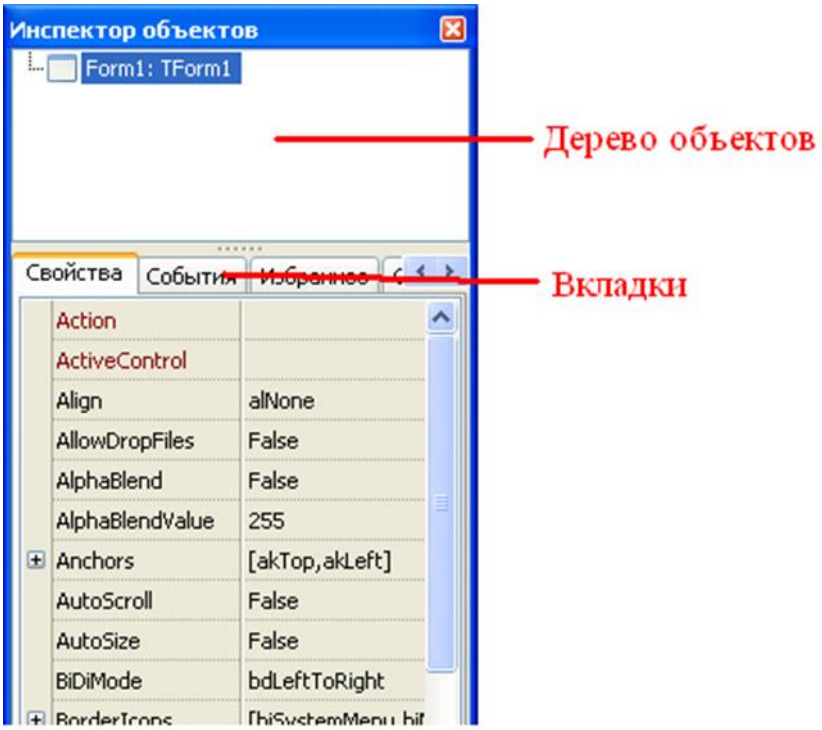


Рис. 1.5. Инспектор объектов

Редактор форм, Редактор кода и Окно сообщений

Последние три окна проще. **Редактор форм** предназначен, соответственно, для редактирования формы - положения и размеров компонентов, размещенных на этой форме.

Несмотря на явную схожесть, форма и окно приложения - не одно и то же. **Форма** - это то, что видит программист в процессе разработки проекта, а окно - это то, что увидит пользователь, когда загрузит нашу программу.

Редактор кода содержит исходный код, который нам придется вводить и модифицировать. Редактор обладает рядом полезных умений: подсвечивает синтаксис команд, делает авто-отступ и авто-завершение команд, выводит необходимые подсказки, в общем, сильно облегчает жизнь программисту.

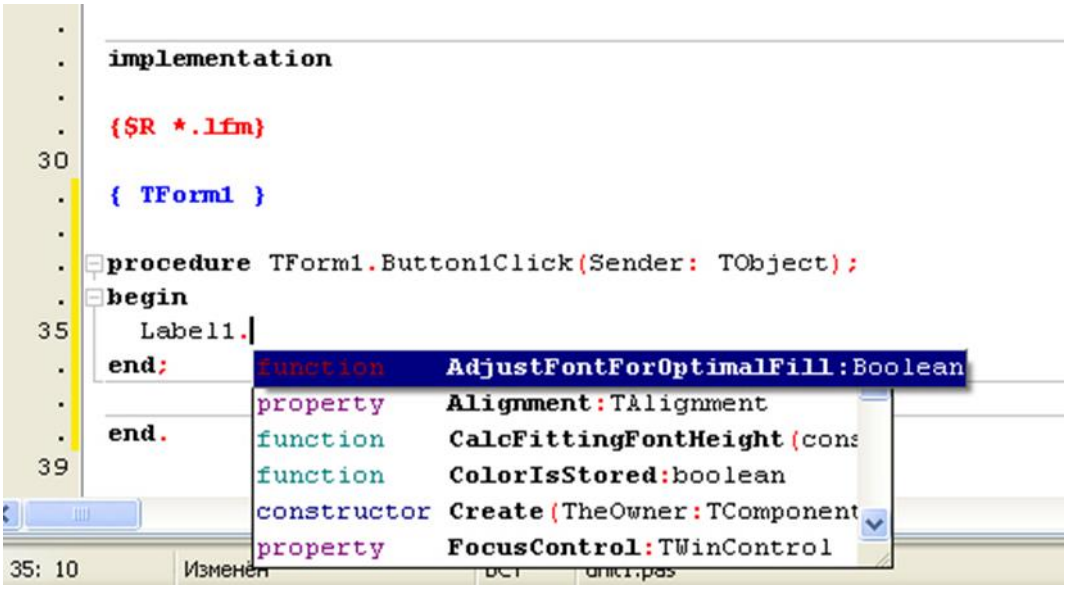


Рис. 1.6. Редактор кода

Нам придется часто переключаться между **Редактором форм** и **Редактором кода**. Проще всего это делать кнопкой **<F12>**.

И, наконец, **Окно сообщений** выводит различные сообщения: о найденных ошибках, о завершении компиляции, о наличии объявленных, но неиспользуемых переменных и т.п.

Lazarus очень похож на Delphi, однако есть и отличия.

- Lazarus, в отличие от Delphi, бесплатен, и может свободно и легально применяться в любом учебном, государственном или производственном учреждении, или дома.
- Lazarus имеет собственную библиотеку компонентов LCL (Lazarus Component Library), а Delphi - VCL (Visual Component Library). Однако VCL и LCL во многом так похожи, что программист при работе с компонентами почти не ощущает разницы. Часто (но не всегда) проекты, написанные на Delphi можно без потерь компилировать на **Lazarus**.
- Lazarus кросс-платформенная IDE, то есть, поддерживает различные

операционные системы. Существует, правда, Kilyx - реализация Delphi для Linux, однако Lazarus имеет реализации для гораздо большего списка операционных систем, причем как 32-х, так и 64-х разрядных версий.

- Lazarus, в силу того, что моложе Delphi, пока имеет меньшую поддержку: дополнительные компоненты сторонних разработчиков, книги на русском языке, сайты, посвященные языку и т.п.
- Lazarus имеет менее развитые средства для работы с Базами Данных. Будем надеяться, это временный недостаток.
- Некоторые компоненты LCL в Lazarus еще "сырые" - иногда попадаются свойства, которые не работают. Чаще всего, это второстепенные свойства, так что можно смело использовать компоненты и без них.

Несмотря на все отличия, эти IDE так похожи, что можно смело утверждать - Delphi-программист почти без усилий сможет пользоваться **Lazarus**, и наоборот. А бурное развитие молодого Lazarus гарантирует, что в будущем его немногочисленные недостатки будут исправлены.

Первая программа

Прежде всего, давайте создадим где-нибудь общую папку, в которую затем будем складывать все наши проекты. Пусть это будет F**:\Education**

Каждый проект по правилам, должен сохраняться в отдельную папку. Чтобы не запутаться, будем давать этим папкам имена, соответствующие номеру лекции, и номеру проекта в ней (занятие - проект). То есть, в первом занятии первый проект будет сохранен в папку

F:\Education\01-01

Вы можете выработать и собственные правила наименования папок - суть от этого не изменится, лишь бы вы сами потом в них не запутались. Мы

несколько раз упомянули слово **проект**.

Проект - *это то, что разрабатывает программист. Когда проект готов и* **скомпилирован**, *получается* **программа**, *с которой может работать пользователь. Проект - это набор связанных файлов различного типа, а программа - это полученный в результате компиляции* **исполняемый файл**. *Подробней о проектах мы поговорим в следующей лекции.*

По традиции, первая созданная программа должна просто выводить сообщение "Hello, world!" ("Привет, мир!"). Не будем отступать от традиций, и сделаем это тремя разными способами. Итак, создадим на диске указанные выше папки, куда сохраним наш первый проект. Загружаем **Lazarus**, если он еще не загружен, и выделяем редактор форм. В левой части, если вы не забыли, находится **Инспектор объектов**, и в нем выделена вкладка "**Свойства**" - это нам и нужно. Среди свойств найдите Caption, и вместо текста

Form1

который там находится по умолчанию, впишите

Hello, world!

По окончании этой процедуры нажмите **<Enter>**. Как только вы это сделаете, текст в заголовке формы изменится:

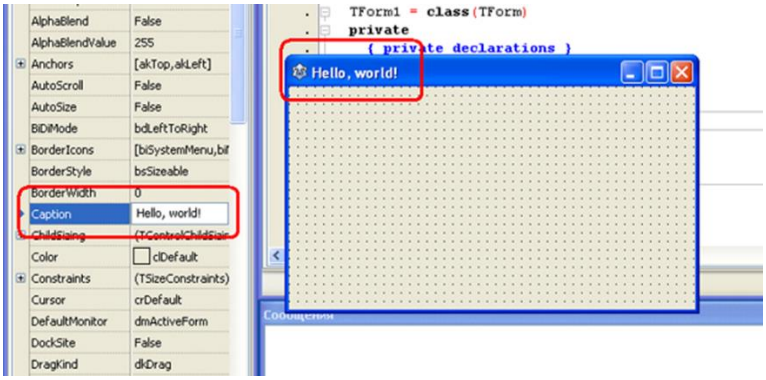


Рис. 1.7. Первые шаги

Теперь обратите внимание на **Палитру компонентов**. На вкладке **Standard**, четвертый значок изображает кнопку с надписью "**Ok**" на ней. Эта кнопка нам и нужна.

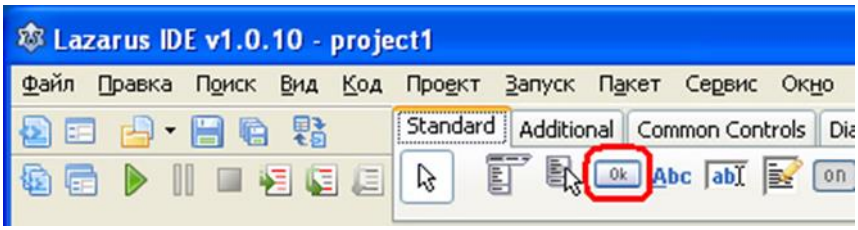
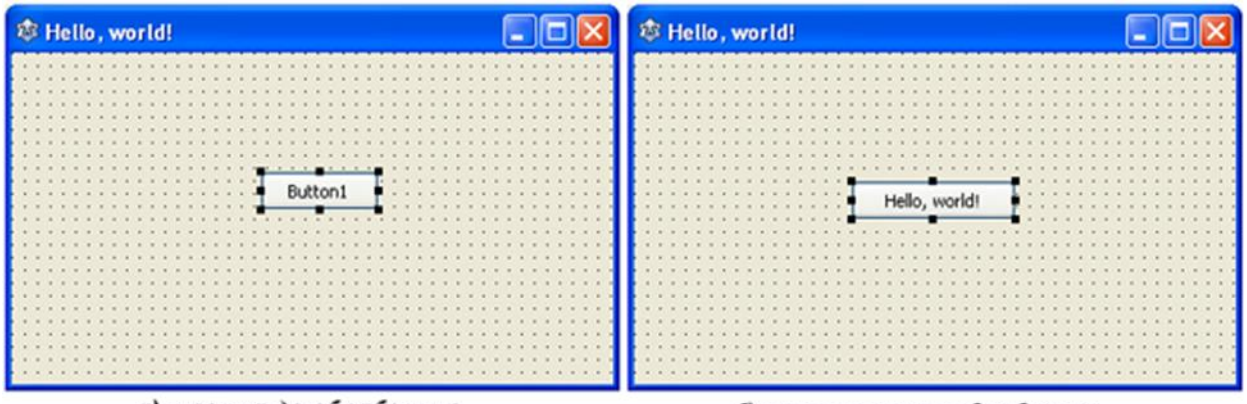


Рис. 1.8. Кнопка TButton в Палитре компонентов

Щелкните по ней мышью, а затем щелкните уже по форме, примерно по центру. На форме тут же появится кнопка, обрамленная рамочкой, с надписью *Button1* - такое имя **Lazarus** дал кнопке по умолчанию. Рамочка вокруг кнопки говорит о том, что ухватившись мышью за одну из ее сторон или углов, мы сможем менять размеры кнопки. Ухватившись за саму кнопку, мы сможем перемещать ее по форме.

Слева, в **Инспекторе объектов**, список свойств также изменился - некоторые остались прежними, другие добавились. Поступим, как и в прошлый раз - в свойстве Caption вместо *Button1* впишем *Hello, world!*. Затем мышью изменим размеры и расположение кнопки, чтобы у нас получилось примерно следующее:



а) кнопка до обработки *б) кнопка после обработки*

Рис. 1.9. Кнопка до и после обработки

Обратите внимание: перемещать кнопку по форме можно не только мышью, но и клавишами **<Ctrl> + <Кнопки управления курсором>** (стрелки вверх, вниз, влево, вправо). А изменять ее размеры - клавишами **<Shift> + <Кнопки управления курсором>**. Этот способ удобней для более точной настройки положения и размеров любых компонентов, не только кнопки.

Мы попробовали два способа вывести нужный текст. Теперь поработаем с исходным кодом. Щелкните дважды по кнопке, и **Lazarus** откроет **Редактор кода**, создав обработчик для этой кнопки, и установив курсор внутрь него. Здесь нам пока ничего понимать не нужно, просто впишите текст, прямо туда, где находится курсор:

```
ShowMessage('Hello, world!');
```

Чтобы получилось так:

Рис. 1.10. Код обработчика кнопки

На этом наш проект закончен. Осталось сохранить его и скомпилировать в программу. Выберите команду меню **Файл -> Сохранить все**, или (что проще) нажмите соответствующую кнопку на **Панели инструментов**:

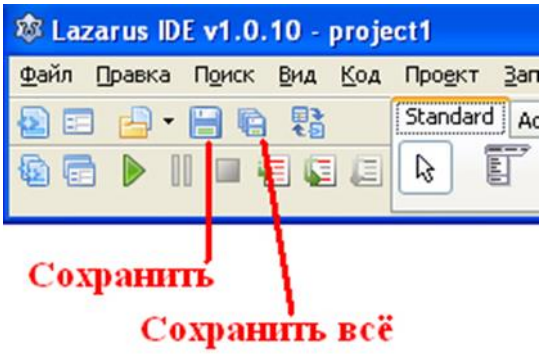


Рис. 1.11. Кнопки сохранения

Кнопка "**Сохранить**" сохраняет изменения текущей формы, кнопка "**Сохранить всё**" - изменения всех форм и модулей проекта. Поскольку форма у нас одна, то можно нажать любую из них. Как только вы это сделаете, выйдет окно сохранения проекта. Вы помните, в какую папку мы будем сохранять первый проект первой лекции? Туда и сохраняйте. Название проекта оставьте без изменений, подробней об этом мы поговорим в следующей лекции.

Как только вы сохраните проект, выйдет еще один запрос - на сохранение файла модуля **Unit1**. Здесь мы тоже оставляем название по умолчанию, нажмем лишь кнопку "**Сохранить**". Обратите внимание - кнопки "**Сохранить**" и "**Сохранить всё**" стали неактивными - это означает, что ни в форме, ни в проекте в целом у нас изменений нет.

Хорошо, проект мы сохранили, однако работающей программы у нас пока нет. Чтобы ее получить, нужно **скомпилировать** проект. Причем сделать это можно тремя командами **Главного меню**:

- **Запуск -> Компилировать**
- **Запуск -> Собрать**
- **Запуск -> Запустить**

Последняя команда не только компилирует проект и создает загрузочный файл программы, но и сразу запускает его на выполнение, так что в большинстве случаев предпочтительней именно эта команда. Удобнее всего воспользоваться соответствующей кнопкой на **Панели инструментов**:

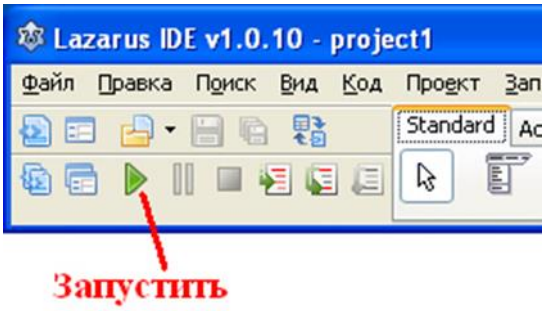


Рис. 1.12. Кнопка запуска

Нажмите эту кнопку, проект скомпилируется и сразу же запуститься. Как только мы нажмем кнопку "**Hello, world!**" в окне программы, выскочит записанное в **Редакторе кода** сообщение:

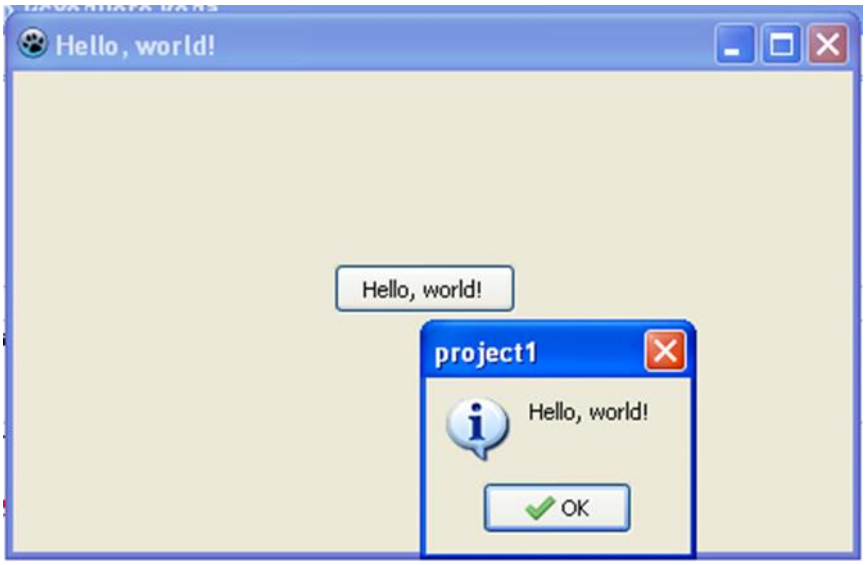


Рис. 1.13. Работа нашей программы

Нажмите кнопку "**Ок**", закрыв сообщение, после чего закройте саму программу (не проект!). Если выйдет подобное сообщение:

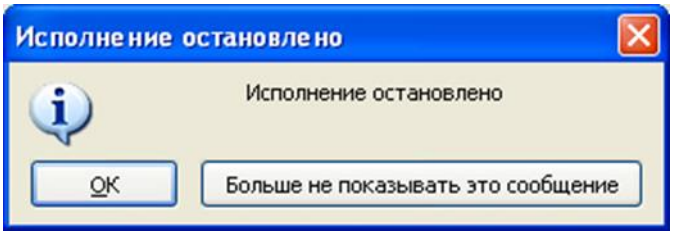


Рис. 1.14. Сообщение об остановке выполнения

то советую нажать кнопку "**Больше не показывать это сообщение**", чтобы в дальнейшем оно вам не надоедало. После этого можете закрыть и **Lazarus**. Исполняемый файл **project1.exe** с нашей программой вы найдете в папке, куда сохраняли проект. Это уже вполне работоспособная программа, её можно запустить прямо из этой папки или переслать другу, чтобы он позавидовал вашим новым знаниям (для выполнения программы требуется только один файл **project1.exe**, остальные файлы не нужны - это файлы проекта, которые нужны только программисту).

Полезные ссылки

В заключение лекции дам вам несколько ссылок, которые могут оказаться полезными при изучении материалов курса.

- http://lazarus.freepascal.org/ - Официальный сайт Lazarus. Здесь вы сможете и новые версии найти, и новости почитать.
- http://wiki.freepascal.org/Main_Page/ru - Русскоязычная документация по Lazarus и Free Pascal. Если постараться, тут можно найти ответы почти на все вопросы, которые непременно у вас будут возникать.
- http://www.cyberforum.ru/lazarus/ - Большой форум по Lazarus. Форумы вообще очень полезны и информативны, а форумы о программировании особенно. Даже если вы и не смогли найти в Интернете решение вашей проблемы, на форуме непременно найдется какой-нибудь "гуру", который вам поможет советом.

- http://www.freepascal.ru/ - Русскоязычный информационный портал о Free Pascal и Lazarus. Тоже весьма полезный сайт.

- http://lazarus.16mb.com/ - А это уже мой сайт, который задумывался для поддержки данного курса. В разделе "**Скачать**" вы сможете найти исходные коды всех примеров из всех лекций курса на случай, если у вас что-то не будет получаться, или будет просто лень набивать код самостоятельно. Там же вы найдете и сможете скачать все инструменты программиста, которые рассматриваются на данном курсе, за исключением самого Lazarus - программа очень часто обновляется, а на сайте разработчика всегда самая свежая версия, поэтому скачивать Lazarus разумней там. Так же, на сайте я постараюсь почаще выкладывать дополнительный материал, не вошедший в данный курс.

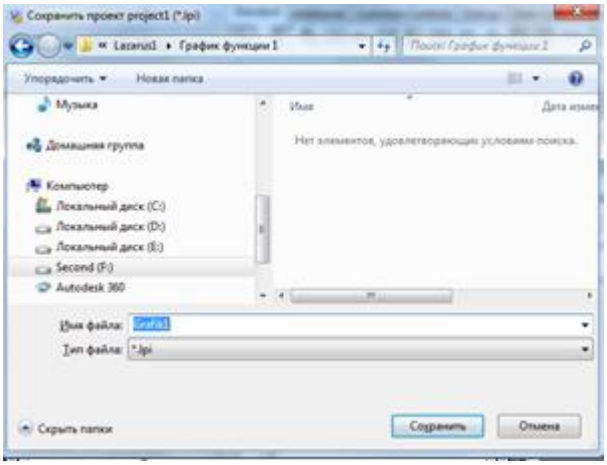
ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №2. ИЗУЧЕНИЕ ПРИНЦИПОВ ВЫДАЧИ ИНФОРМАЦИИ В ВИДЕ ДИАГРАММ И ГРАФИКОВ

Цель работы: освоить методы выдачи информации в виде диаграмм и графиков.

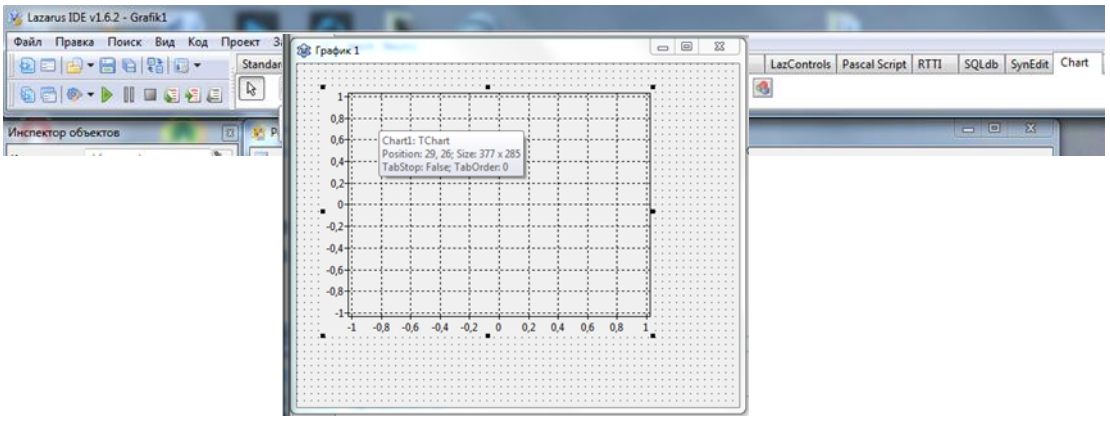
Построение графика случайной функции

Задача построения графика по заданным данным или известной функции в Lazarus решается при помощи компонента TChart. Нарисуем график по заданной функции, имеющей в аргументе случайную величину и потому обновляющей значения по нажатию кнопки.

Создадим новый проект и запишем его в отдельную папку под именем Grafik1.lpi.

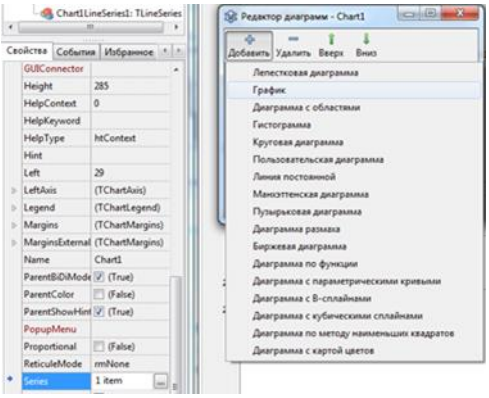


Выберем на вкладке Chart (справа на панели компонентов) компонент TChart и размещаем его на существующей форме.



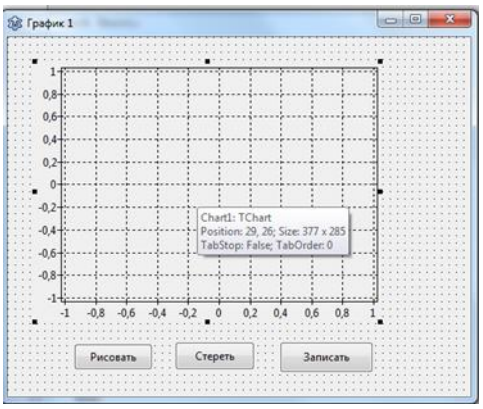
Добавляем к его коллекции данных (Series) (см. свойства TChart) объект типа

TLineSeries. Для этого в свойстве нашего графика TChart находим свойство Series, нажатием на многоточие вызываем редактор диаграмм, нажатием кнопки + добавляем «График».



Добавляем три кнопки, для рисования, очистки графика и сохранения графика в файл BMP (код диалога сохранения файла с выбором места и имени в данном случае мы не используем, поэтому имя файла включаем в код).

В редакторе форма выглядит примерно так:



Далее записываем код нажатия кнопок.

Кнопка Рисовать:

```
procedure TForm1.Button1Click(Sender: TObject);

begin
    randomize;
    Chart1LineSeries1.Clear();
    for i:=1 to 100 do
    begin
```

```
        Chart1LineSeries1.AddXY(i*0.1,sin(random(4)*i*0.1));
    end;
end;
```

Кнопка Стереть:

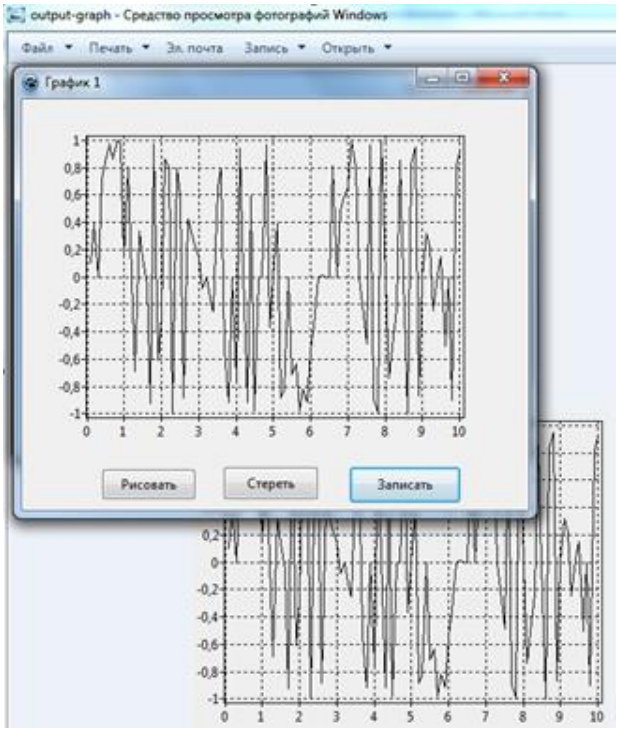
```
procedure TForm1.Button2Click(Sender: TObject);
begin
    Chart1LineSeries1.Clear();
end;
```

Кнопка Запистаь:

```
procedure TForm1.Button3Click(Sender: TObject);
begin
    Chart1.SaveToBitmapFile('output-graph.bmp');
end;
```

На рисунке показан результат работы программы и фрагмент созданного рисунка.

Рисунок сохраняется в папке проекта под именем *'output-graph.bmp'*.



Построение графика функций с использованием функций и процедур

Задача 1. Выполните табулирование функции **y=sin(x)**, если известны начальное значение интервала, на котором изменяется функция, конечное значение интервала и шаг ее изменения. Требования к программе:

1. Вычисление функции y=sin(x) оформите в виде пользовательской функции.
2. Задачу табулирования функции выполните в виде пользовательской процедуры.
3. Создайте событийную процедуру, из которой вызовите пользовательскую процедуру.
4. Постройте график функции.

Комментарий: Вновь рассмотрим решение задачи табулирования функции, однако теперь дополним графический интерфейс программы построением графика. Для этого, прежде всего, разместим на форме компонент **Chart1** из палитры **Chart**, на который будем выводить график функции. Найдем свойство **Titles** компонента и введем в подсвойстве **Text** заголовок **График функции**. Теперь нужно сделать заголовок видимым, выставив свойство **Visible** в **True**. Сделайте двой-

ной щелчок на объекте **Chart1** и в появившемся окне, нажав кнопку **Add**, выберите тип графика **BarSeries**. Отключим легенду графика, выполнив команду **Legend/Visible/False**).

С точки зрения алгоритмизации суть пользовательских функции и процедуры, представленных в листинге 104, ничем не отличается от кода листинга 98.

Листинг 104

```
implementation

{$R *.lfm}

{ TForm1 }

Function func(x : Single) : Single;
      Var y : Single;
Begin
   y := Sin(x);
  func:=y;
End;

Procedure tab(Var a, b, h: Single);
 Var i,n : Integer;
     y, x : Single;
     MySerie: TLineSeries;
Begin
 Form1.Chart1.ClearSeries; // Очищаем компонент Chart1
 MySerie:=TLineSeries.Create(Form1.Chart1); // Создаем свою серию в компоненте
                                            //Chart1
 MySerie.LinePen.Color:=clRed; // Определяем ее цвет (красный)
 Form1.Chart1.AddSeries(MySerie);
 Form1.StringGrid1.Cells[0,0]:=' X';
 Form1.StringGrid1.Cells[1,0]:=' f(X)';
 Form1.StringGrid1.GridLineWidth:=2;
 Form1.StringGrid1.RowCount:=2;
 x := a;
 n := Round((b - a) / h) + 1;
 For i := 1 To n Do
     Begin
       y := func(x);
Form1.StringGrid1.Cells[0,Form1.StringGrid1.RowCount-1]:=FloatToStrF(X,ffGeneral,6,6);
Form1.StringGrid1.Cells[1,Form1.StringGrid1.RowCount-1]:=FloatToStrF(Y,ffGeneral,6,6);
    Form1.StringGrid1.RowCount:=Form1.StringGrid1.RowCount + 1;
     MySerie.AddXY(X,Y);
     x := x + h;
     End;
   Form1.StringGrid1.RowCount:=Form1.StringGrid1.RowCount-1;
End;
```

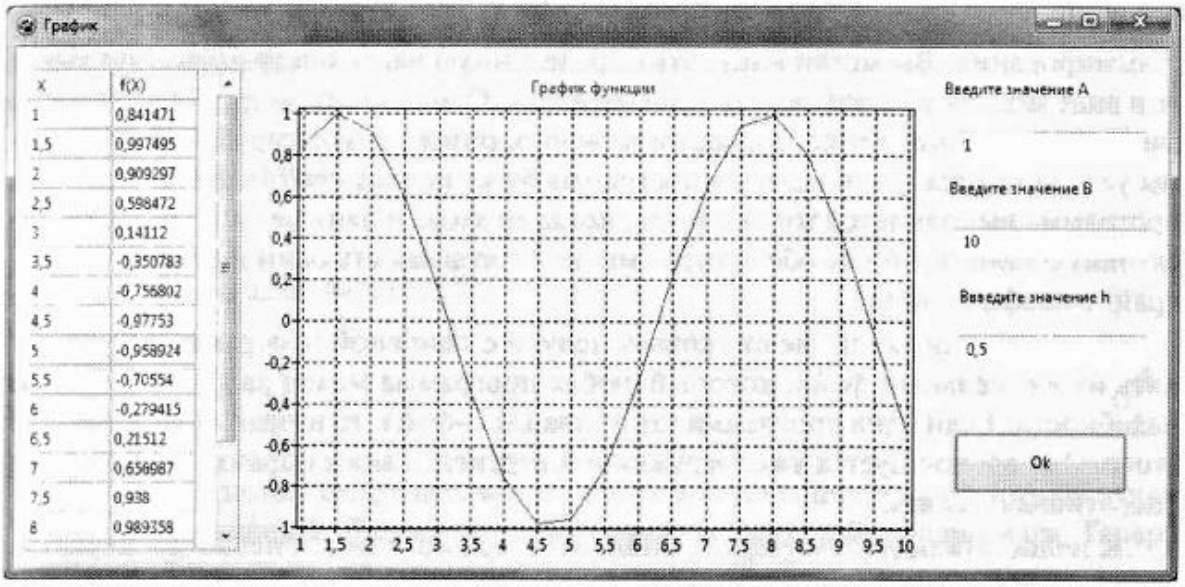


Рис. 135. Вывод графика на элемент управления Chart

```
Procedure TForm1.Button1Click(Sender: TObject);
    Var  a, b, h : Single;
Begin
  a := StrToFloat (Edit1.Text);
  b := StrToFloat (Edit2.Text);
  h := StrToFloat (Edit3.Text);
  tab(a, b, h);
End;
```

Построенный график функции представлен на рис. 135.

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №3. ФРАКТАЛЬНАЯ ГРАФИКА

Цель работы: научиться выполнять программирование фрактальной графики.

Постановка задачи

Понятия фрактал и фрактальная геометрия, появившиеся в конце 70-х, с середины 80-х гг. прочно вошли в обиход математиков и программистов. Слово фрактал образовано от латинского fractus и в переводе означает состоящий из фрагментов. Оно было предложено Бенуа Мандельбротом в 1975 г. для обозначения нерегулярных, но самоподобных структур, которыми он занимался.

Самыми известными фрактальными объектами являются деревья: от каждой ветки ответвляются меньшие, похожие на нее, от тех — еще меньшие и так далее. По отдельной ветке математическими методами можно проследить свойства всего дерева. Фрактальными свойствами обладают многие природные объекты: снежинка при увеличении оказывается фракталом; по фрактальным алгоритмам растут кристаллы и растения. Если посмотреть на береговую линию моря на картах все более крупного масштаба, то становятся видны все новые изгибы и изломы, похожие на более крупные.

Роль фракталов в машинной графике сегодня достаточно велика. Они приходят на помощь, например, когда требуется, с помощью нескольких коэффициентов, задать линии и поверхности очень сложной формы. С точки зрения машинной графики, фрактальная геометрия незаменима при генерации искусственных облаков, гор, поверхности моря. Фактически найден способ легкого представления сложных неевклидовых объектов, образы которых весьма похожи на природные.

Одним из основных свойств фракталов является самоподобие. В самом простом случае небольшая часть фрактала содержит информацию о всем фрактале.

Классификация фракталов

Геометрические фракталы

Фракталы этого класса самые наглядные. В двухмерном случае их получают с помощью некоторой ломаной (или поверхности в трехмерном случае), называемой генератором. За один шаг алгоритма каждый из отрезков, составляющих ломаную, заменяется на ломаную-генератор, в соответствующем масштабе. В результате бесконечного повторения этой процедуры, получается геометрический фрактал.

Примером такого фрактального объекта является триадная кривая Кох

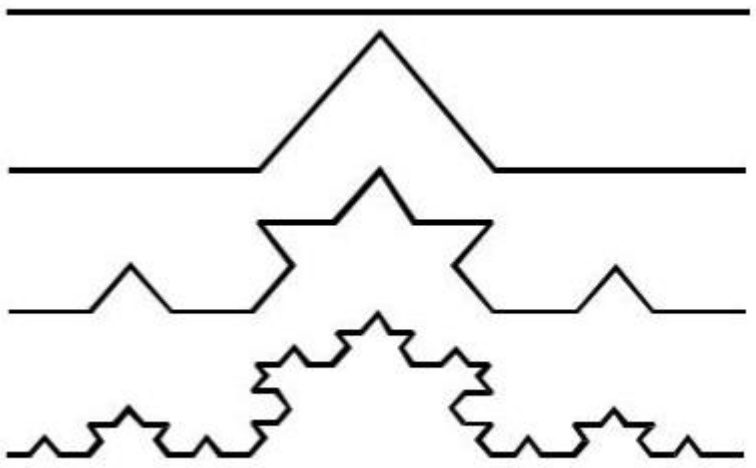


Рисунок 3.1 Построение триадной кривой Кох

Построение кривой начинается с отрезка единичной длины – это нулевое поколение кривой Кох. Далее каждое звено (в нулевом поколении один отрезок) заменяется на образующий элемент. В результате такой замены получается

следующее поколение кривой Кох. В первом поколении - это кривая из четырех прямолинейных звеньев, каждое длиной по 1/3. Для получения третьего поколения проделываются те же действия - каждое звено заменяется на уменьшенный образующий элемент. Итак, для получения каждого последующего поколения, все звенья предыдущего поколения необходимо заменить уменьшенным образующим элементом. Кривая n-го поколения при любом конечном n называется предфракталом.

На рис. 3.1 представлены три поколения кривой. При n стремящемся к бесконечности кривая Кох становится фрактальным объектом.

В машинной графике использование геометрических фракталов необходимо при получении изображений деревьев, кустов, береговой линии. Двумерные геометрические фракталы используются для создания объемных текстур.

Алгебраические фракталы

Это самая крупная группа фракталов. Свое название они получили за то, что их строят на основе алгебраических формул, иногда весьма простых.

Методов получения алгебраических фракталов несколько. Один из методов представляет собой многократный (итерационный) расчет функции $Zn+1= f(zn)$, где Z – комплексное число, а f – некая функция. Расчет данной функции продолжается до выполнения определенного условия. И когда это условие выполнится - на экран выводится точка. При этом значение функции для разных точек комплексной плоскости может иметь разное поведение:

- с течением времени стремится к бесконечности;
- стремится к 0;
- принимает несколько фиксированных значений и не выходит за их пределы;
- поведение хаотично, без каких-либо тенденций.

Примером этого вида фракталов является множество Мандельброта (рис. 3.2).

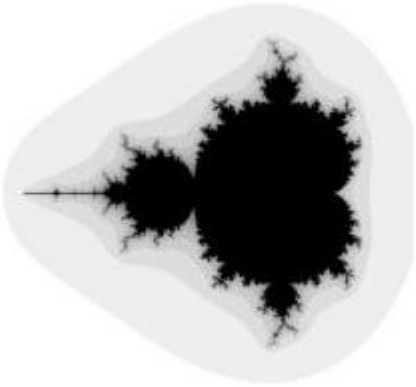


Рис. 3.2. Множество Мандельброта

Стохастические фракталы

Еще одним известным классом фракталов являются стохастические фракталы, которые получаются в том случае, если в итерационном процессе случайным образом менять какие-либо его параметры. При этом получаются объекты очень похожие на природные - несимметричные деревья, изрезанные береговые линии и т.д. (рис. 9.3). Двумерные стохастические фракталы используются при моделировании рельефа местности и поверхности моря.

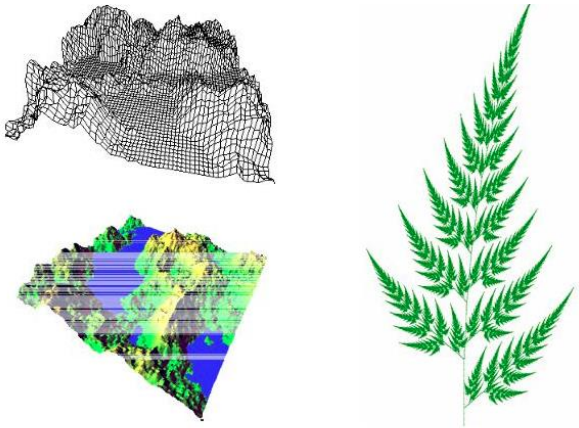


Рис. 3.3. Стохастические фракталы

Способность фрактальной графики моделировать образы живой природы вычислительным путем часто используют для автоматической генерации необычных иллюстраций.

Одним из фракталов множества Мандельброта является «паук». Рассматривается множество таких a, которые стремятся к бесконечности при итерировании вида:

z0=c0=a;

z'=z2+c;

c'=c/2+z';

Для построения фрактала будем использовать следующие формулы:

z.X=(z1.X)2-(z1.Y)2+c.X;

z.Y=2*z1.X*z1.Y+c.Y;

c.X=c1.X/2+z.X;

c.Y=c1.Y/2+z.Y;

Для прорисовки фрактала следует нажать кнопку "Fractal"

Код программы:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, ExtCtrls,
  StdCtrls;

type

  { TForm1 }

  TForm1 =class(TForm)
    Fractal: TButton;
    PaintBox1: TPaintBox;
    procedure FractalClick(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation
```

```
{$R *.lfm}

{ TForm1 }

procedure TForm1.FractalClick(Sender: TObject);
   type
        TComplex = record
                X : Real;
                Y : Real;
        end;
       const
        iter = 50;
        max  = 16;
       var
        z1, z2, c1, c2 : TComplex;
        x, y, n : Integer;
        Mx, My  : Integer;
         col:TColor;
begin
    PaintBox1.Canvas.Clear; //очищаем canvas
    Mx := PaintBox1.Canvas.Width div 2;  //вычисляем центр (ширина)
    My := PaintBox1.Canvas.Height div 2; //вычисляем центр (высота)
    for y:=-My to My   do
     for x:=-Mx  to Mx   do
     begin
           //устанавливаем начальные значения параметров
           n:=0;
           z1.X:=x*0.01;
           z1.Y:=y*0.01;
           c1.X:=z1.X;
           c1.Y:=z1.Y;
            while ((z1.X*z1.X+z1.Y*z1.Y)<max) and (n<iter) do
            begin
                 z2:=z1;
                 c2:=c1;
                 z1.X:=(z2.X*z2.X)-(z2.Y*z2.Y)+c1.X;
                 z1.Y:=2*z2.X*z2.Y+c1.Y;
                 c1.X:=c2.X/2+z1.X;
                 c1.Y:=c2.Y/2+z1.Y;
                 n:=n+1;
            end;

             if (n<iter)    //цвет выбираем по числу итераций
             then
             begin
             col :=30*n mod 255;
             PaintBox1.Canvas.Pixels[(Mx  div  2)+x,  (My  div  2)+y]:=RGBToColor(0,
col, col);
             end;
     end;
end;

end.
```

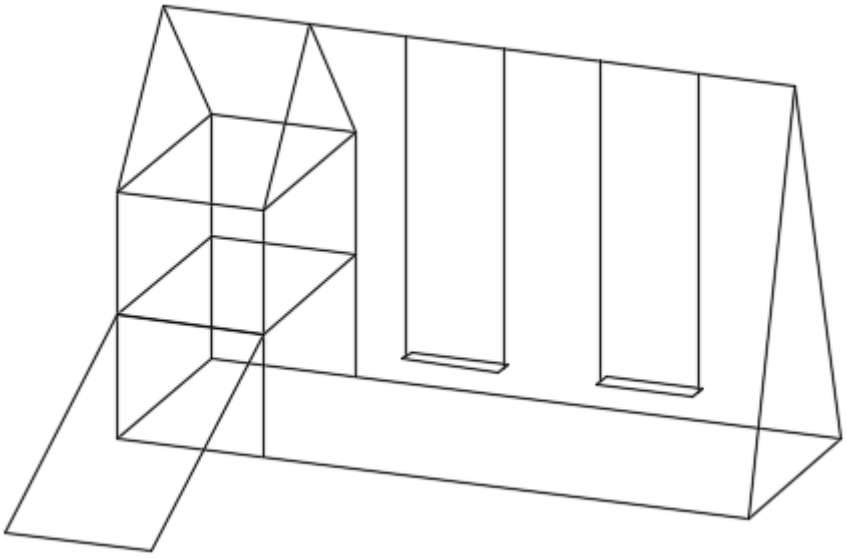
ПРАКТИЧЕСКОЕ ЗАНЯТИЕ №4. МОДЕЛИ ТРЁХМЕРНЫХ ОБЪЕКТОВ.

Цель работы: освоить теоретические принципы и инструментальные средства моделирования каркасных моделей трёхмерных объектов.

Создание каркасных 3D-моделей

Каркасные 3D-модели представляет собой представление реального объекта, созданное с использование ребер или скелета.

Каркасные 3D-модели состоят из точек, линий, дуг, окружностей и других кривых, определяющих ребра или центральные линии объектов.



Каркасная 3D-модель используется для следующих целей.

Создание базовых 3D-проектов для проверки и быстрого внесения корректировок

Обзор модели со всех сторон

Анализ пространственных взаимосвязей, включая расстояния между углами и ребрами, а также визуальная проверка на предмет возможных пересечений

Создание видов в перспективе (недоступно в AutoCAD LT)

Автоматическая генерация ортогональных и дополнительных видов

Ссылочная геометрия для моделирования 3D-тела, поверхности или сети

Рекомендуется сохранить каркасную 3D-геометрию на отдельном ссылочном слое, чтобы иметь к ней доступ при необходимости проверки целостности 3D-модели или частичного ее воссоздания.

Способы построения каркасных 3D-моделей

Для создания каркасных моделей требуется наличие определенного опыта. Для освоения каркасного моделирования лучше начинать с построения простых моделей с последующим переходом на более сложные.

Имеется возможность создавать каркасные модели путем размещения плоских 2D объектов в любом месте 3D пространства. Для этого предлагаются следующие способы:

Задайте 3D-координаты, отражающие местоположение определяющих точек объекта на осях *X*, *Y* и *Z*.

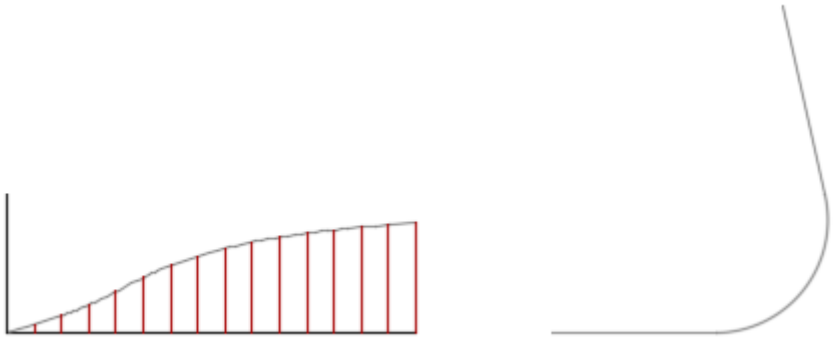
Задайте рабочую плоскость по умолчанию, которая является плоскостью *XY* ПСК, на которой создаются плоские объекты (дуги и окружности).

Создайте объект и перенесите, скопируйте или поверните его для окончательного размещения в 3D-пространстве.

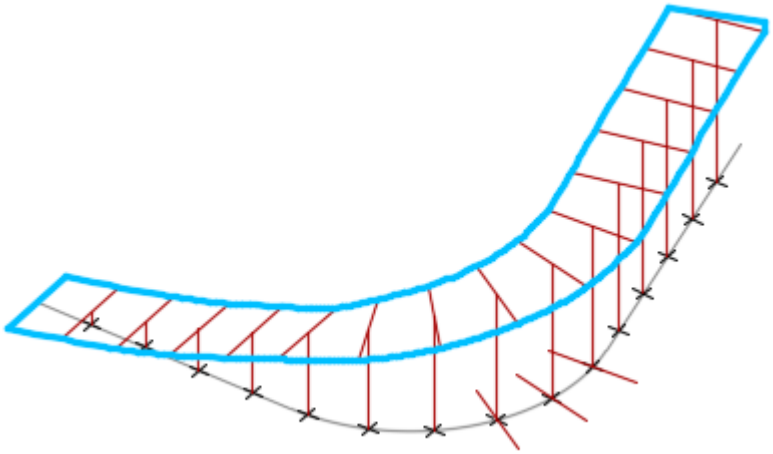
Прим.: Для создания каркасной геометрии на основе областей, 3D-тел, поверхностей и сетей можно использовать команду ИЗВЛРЕБРА. Извлеченные ребра формируют каркас, составленный из 2D объектов (отрезков, окружностей) и 3D полилиний. (Эта функция недоступна в AutoCAD LT.)

Пример каркасной 3D-модели

Иногда работу над проектом проще всего начать с каркасной модели. Такую модель также можно использовать как ссылочную геометрию при моделировании тел и поверхностей. В качестве примера можно привести дорожное полотно, уровень которого повышается при повороте. Изменение отметки носит нелинейный характер, а радиус кривой является постоянным значением (см. рисунок ниже).



Вспомогательные линии Т-образной формы с равным шагом обеспечивают точки для внутренней и внешней кромки В-сплайна дорожного полотна. Верхние края каждой Т-линии можно окантовать или скруглить.



На этом этапе можно использовать сплайны, чтобы создать 3D-тело для визуализации, расчета разрезов и заливки или для расчетов инженерных конструкций.

1. Разработка полигональной модели объекта

1.1 Составляющие элементы объекта

Полигональная модель в компьютерной графике — это образ объекта, составленный из множества многоугольников. Мы будем разрабатывать полигональную модель объекта «самолёт». Трехмерное изображение объекта представлено на рис. 1.1.

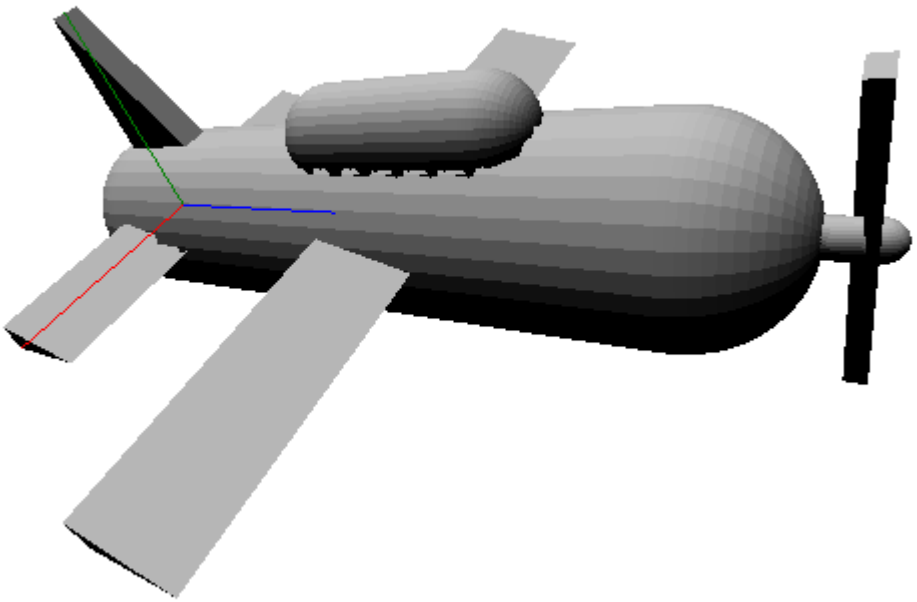


Рисунок 1.1 – Трехмерное изображение проектируемого объекта

Объект состоит из нескольких видов поверхностей, которые в свою очередь разбиваются на полигоны для удобства хранения и обработки. Объект состоит из следующих видов фигур:

- капсула;
- параллелепипед;
- призма, в основании которой прямоугольный треугольник.

Капсула задаётся базовой точкой, радиусом, высотой цилиндрической части капсулы. Параллелепипед задаётся базовой точкой, шириной, длиной и высотой, на основе которых происходит расчет координат восьми точек вершин параллелепипеда. Призма задаётся базовой точкой, 2-мя катетами треугольника, который лежит в основании призмы, и высотой призмы.

1.2 Триангуляция поверхности объекта

Триангуляция поверхностей – это процесс разбиения сложных объектов на треугольные полигоны. Триангуляция удобна при программировании графики, т.к.:

- треугольник является простейшим полигоном, вершины которого однозначно задают грань;
- любую область можно гарантированно разбить на треугольники;
- вычислительная сложность алгоритмов разбиения на треугольники существенно меньше, чем при использовании других полигонов;
- реализация процедур визуализации более проста для области, ограниченной треугольником;
- для треугольника легко определить три его ближайших соседа, имеющих с ним общие грани.

Над объектом производится триангуляция следующим образом. Все примитивы, из которых состоит объект разбиты на треугольники. Все прямоугольные грани разбиваются диагональю на два равных треугольника. Таким образом, параллелепипеды состоят из 12 треугольных полигонов. Боковая поверхность цилиндра (часть нашей капсулы) разбита на части по высоте и каждая часть разбивается на равные прямоугольники, которые после разбиваются на треугольники.
Для получения координат вершин капсулы используются формулы перехода из цилиндрической системы координат в декартову систему координат (рис. 1.2). В формулах цилиндр имеет радиус R и высоту ZC. Полной окружности соответствует диапазон изменения параметра угла от 0 до 2Пи. Если рассматривать некоторое фиксированное число, равномерно распределенных, точек по окружности, то приращение параметра между точками можно считать константой (рис. 1.3). На концах капсулы находятся полусферы, которые получаются путём разбиения конуса на части по высоте и вдоль основания, после этого нормализуем координаты всех вершин так что бы они лежали на нашей сфере и полученные прямоугольники разбиваем на полигоны.

$$X = R * \sin \alpha$$

$$Y = R * \cos \alpha$$

$$Z = ZC$$

Рисунок 1.2 – Формулы перехода из цилиндрической в декартову систему координат

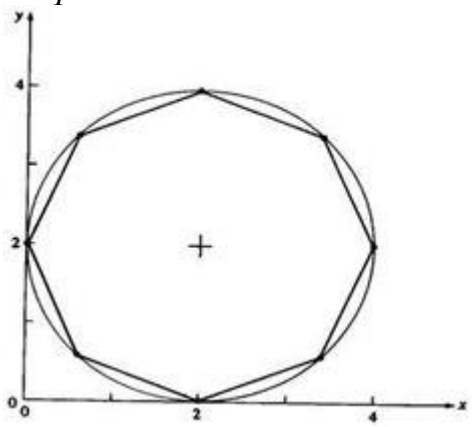


Рисунок 1.3 – Параметрическое представление окружности

Призма, в основании которой прямоугольный прямоугольник состоит из 2 треугольников и 3 прямоугольников, которые легко разбиваются на полигоны. На рисунке 1.4 представлено каркасное изображение объекта.

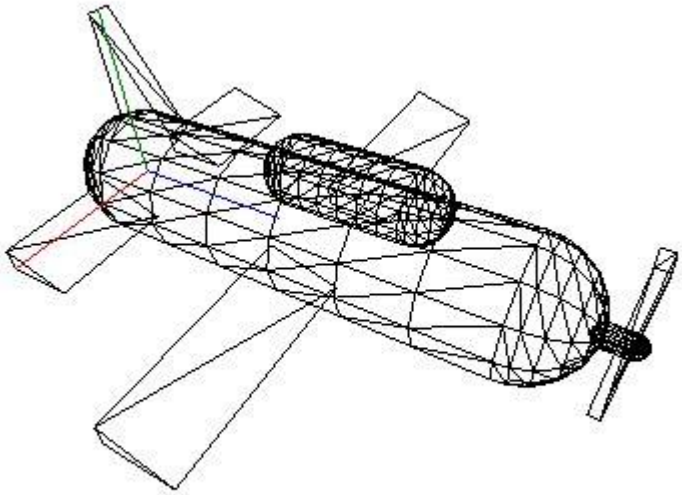


Рисунок 1.4 – Каркасное изображение проектируемого объекта

2. Описание алгоритмов аффинных преобразований

Любое изображение, выводимое на экран монитора, состоит из точек. Каждая точка в трехмерном пространстве, в свою очередь содержит три координаты – X (абсцисса), Y (ордината), Z (аппликата). Координаты точки однозначно определяют ее положение в системе координат. Мы будем использовать правую декартовую прямоугольную систему координат, общий вид которой показан на рисунке 2.1.

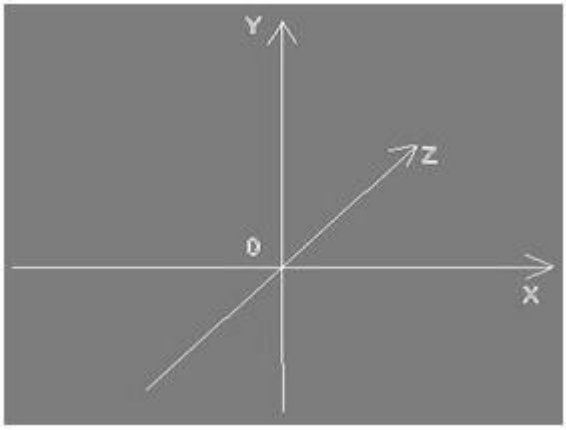


Рисунок 2.1 – Правая декартовая прямоугольная система координат

Здесь буквами x, y, z обозначены положительные направления осей Ox, Oy и Oz соответственно. Для вывода сцены на экран используются 3 системы координат:

- СК сцены;
- экранная СК;
- СК камеры.

Начальной СК является СК сцены (либо камеры, при переходе в вид из камеры), конечной – экранная СК. Для перехода из одной системы координат в другую используются формулы поворота, смещения и масштабирования.

После выполнения геометрических преобразований, координаты (x, y, z) точки переходят в новые координаты (x*, y*, z*). Общие формулы преобразования

координат показаны на рисунке 2.2. Но более удобной является матричная форма записи трехмерного преобразования (рис. 2.3). Здесь к координатам (x ,y, z) точки добавилась четвертая координата, равная единице и необходимая для выполнения преобразований в матричной форме. Такие координаты называются однородными.

$$x^* = \alpha_1 x + \alpha_2 y + \alpha_3 z + \lambda$$
$$y^* = \beta_1 x + \beta_2 y + \beta_3 z + \mu$$
$$z^* = \gamma_1 x + \gamma_2 y + \gamma_3 z + \nu$$

Рисунок 2.2 – Формулы преобразования координат точки

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} * A \qquad A = \begin{bmatrix} \alpha_1 & \beta_1 & \gamma_1 & 0 \\ \alpha_2 & \beta_2 & \gamma_2 & 0 \\ \alpha_3 & \beta_3 & \gamma_3 & 0 \\ \lambda & \mu & \nu & 1 \end{bmatrix}$$

Рисунок 2.3 – Матричная форма записи трехмерного преобразования

2.1 Перемещение объекта

Одним из простейших и часто применяемых аффинных преобразований является перемещение (сдвиг, перенос). Матрица A перемещения на вектор (λ, μ, ν) показана на рисунке 2.4. Алгоритм представлен на рисунке 2.5.

$$|A] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \lambda & \mu & \nu & 1 \end{bmatrix}$$

Рисунок 2.4 – Матрица перемещения

```
//формируем матрицу переноса
Matrix MatrixMove = new Matrix(new double[,]{
            {1, 0, 0, 0},
            {0, 1, 0, 0},
            {0, 0, 1, 0},
            {dx, dy, dz, 1}
        });

 // Умножаем её на матрицу трансформации объекта
MatrixTransform *= MatrixMove;

// И применяем к каждой точки
polygon[] = plane.GetPolygon();
for(int i = 0; i < polygon.count; i++)
{
  polygon[i].p1 *= MatrixTransform;
  polygon[i].p2 *= MatrixTransform;
  polygon[i].p3 *= MatrixTransform;
}
```

Рисунок 2.5 – Алгоритм переноса объекта

2.2 Вращение объекта

Аффинным преобразованием, которое позволяет смотреть на объект под разными углами и с разных сторон, является поворот объекта относительно координатных осей. Матрицы поворотов вокруг координатных осей показаны на рисунках 2.6, 2.7, 2.8 и вокруг произвольной оси на рисунке 2.9.

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

Рисунок 2.6 – Матрица поворота вокруг оси OX

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

Рисунок 2.7 – Матрица поворота вокруг оси OY

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

Рисунок 2.8 – Матрица поворота вокруг оси OZ

$$M(\hat{\mathbf{v}},\theta) = \begin{pmatrix} \cos\theta + (1-\cos\theta)x^2 & (1-\cos\theta)xy - (\sin\theta)z & (1-\cos\theta)xz + (\sin\theta)y \\ (1-\cos\theta)yx + (\sin\theta)z & \cos\theta + (1-\cos\theta)y^2 & (1-\cos\theta)yz - (\sin\theta)x \\ (1-\cos\theta)zx - (\sin\theta)y & (1-\cos\theta)zy + (\sin\theta)x & \cos\theta + (1-\cos\theta)z^2 \end{pmatrix}$$

Рисунок 2.9 – Матрица поворота вокруг произвольной оси

Алгоритм поворота вокруг произвольной оси представлен на рисунке 2.10.

```
//формируем матрицу поворота;
Matrix MatrixRotate = new Matrix(new double[,]{
        {Math.Cos(angle) + (1 - Math.Cos(angle)) * axis.x * axis.x, (1 - Math.Cos(angle)) * axis.y * axis.x - Math.Sin(angle) * axis.z, (1 -
Math.Cos(angle)) * axis.x * axis.z + Math.Sin(angle) * axis.y, 0},

        {(1 - Math.Cos(angle)) * axis.x * axis.y + Math.Sin(angle) * axis.z, Math.Cos(angle) + (1 - Math.Cos(angle)) * axis.y * axis.y, (1 -
Math.Cos(angle)) * axis.y * axis.z - Math.Sin(angle) * axis.x, 0},

        {(1 - Math.Cos(angle)) * axis.z * axis.x - Math.Sin(angle) * axis.y, (1 - Math.Cos(angle)) * axis.z * axis.y + Math.Sin(angle) * axis.x,
Math.Cos(angle) + (1 - Math.Cos(angle)) * axis.z * axis.z, 0},

        {0, 0, 0, 1}
    });

//формируем матрицу переноса  объекта
Matrix MatrixMove = new Matrix(new double[,]{
        {1, 0, 0, 0},
        {0, 1, 0, 0},
        {0, 0, 1, 0},
        {-dx, -dy, -dz, 1}
    });
//формируем матрицу обратного переноса
Matrix MatrixUMove = new Matrix(new double[,]{
        {1, 0, 0, 0},
        {0, 1, 0, 0},
        {0, 0, 1, 0},
        {dx, dy, dz, 1}
    });

//умножаем матрицы
MatrixTransform *= MatrixMove * MatrixRotate * MatrixUMove

// И применяем к каждой точке
polygon[] = plane.GetPolygon();
for(int i = 0; i < polygon.count; i++)
{
  polygon[i].p1 *= MatrixTransform;
  polygon[i].p2 *= MatrixTransform;
  polygon[i].p3 *= MatrixTransform;
}
```

Рисунок 2.10 – Алгоритм поворота вокруг произвольной оси

2.3 Масштабирование объекта

Другим важнейшим аффинным преобразованием является масштабирование. Масштабирование — изменение размера изображения с сохранением пропорций. Под масштабированием подразумевается как увеличение изображения, так и его уменьшение. Матрица масштабирования представлена на рисунке 2.11, а алгоритм масштабирования на рисунке 2.12.

$$\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Рисунок 2.11 – Матрица масштабирования

```
Matrix MatrixScale = new Matrix(new double[,]{
                        {sx, 0, 0, 0},
                        {0, sy, 0, 0},
                        {0, 0, sz, 0},
                        {0, 0, 0, 1}
                       });

  // Умножаем её на матрицу трансформации объекта

  MatrixTransform *= MatrixScale;

  // И применяем к каждой точки

  polygon[] = plane.GetPolygon();
  for(int i = 0; i < polygon.count; i++)
  {
    polygon[i].p1 *= MatrixTransform;
    polygon[i].p2 *= MatrixTransform;
    polygon[i].p3 *= MatrixTransform;
  }
```

Рисунок 2.12 – Алгоритм масштабирования

3. Описание алгоритмов проекционных преобразований

Проецирование - это процесс полу¬чения изображения предмета на какой-либо поверх¬ности, получившиеся при этом изображение, называют проекцией предмета.

Элементами, с помощью которых осуществляется проецирование, являются (рис. 3.1):

- центр проецирования - точка, из которой производится проецирование;
- объект проецирования - изображаемый предмет;
- плоскость проекции - плоскость, на которую производится проецирование;
- проецирующие лучи - воображаемые прямые, с помощью которых производится проецирование.

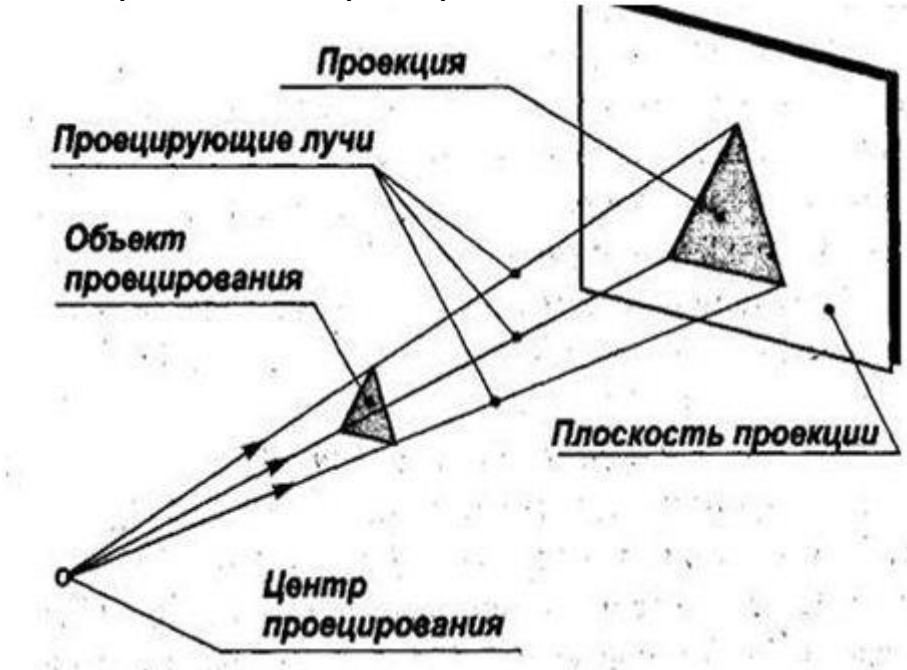


Рисунок 3.1 – Центральное проецирование [2]

Результатом проецирования является изображение или проекция объекта. Различают центральное и параллельное проецирование. При центральном проецировании все проецирующие лучи исходят из одной точки - центра проецирования, находящегося на определённом расстоянии от плоскости проекций (рисунок 3.1). При параллельном проецировании все проецирующие лучи параллельны между собой. На рисунке 3.2 показано, как получается параллельная прямоугольная проекция. Центр проецирования предполагается условно удалённым в бесконечность. Тогда параллельные лучи отбросят на плоскость проекций тень, которую можно принять за параллельную проекцию изображаемого предмета.

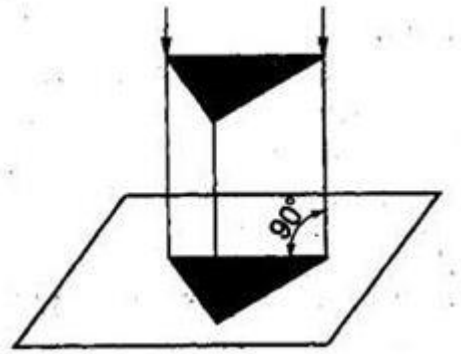


Рисунок 3.2 – Параллельное прямоугольное проецирование [2]

В проекте реализовано переключение между центральным и параллельным проецированием. Отличия при реализации системы проецирования появляются при вычислении координат проекций на аксонометрическую плоскость проекций камеры. Эта плоскость имеет размеры (–1,+1) по ширине и (–1/ar,+1/ar) по высоте, где ar – форматное отношение. Если при центральном проецировании имеет место формула, показанная на рисунке 3.3, то для параллельного проецирования справедлива формула, представленная на рисунке 3.4.

$$x_{per} = d \cdot x / z,$$
$$y_{per} = d \cdot y \cdot ar / z.$$

Рисунок 3.3 – Формула вычисления координат проекций при центральном проецировании

$$x_{per} = d \cdot x / a$$
$$y_{per} = d \cdot y \cdot ar / b$$

Рисунок 3.4 – Формула вычисления координат проекций при параллельном проецировании

В этой формуле a - половина ширины окна вывода, а b - половина его высоты.

4. Описание алгоритмов произвольных преобразований камеры

Для построения изображения на экранной плоскости используется модель представления при помощи камеры. Она отвечает за предоставление пользователю мощнейшего механизма движения среди объектов сцены.

Камера задается следующими параметрами:

- углом обзора;
- фокусным расстоянием (от точки нахождения камеры до плоскости, на которую мы проецируем);

- ближними и дальними плоскостями, которые участвуют в отсечении по пирамиде видимости.

Для камеры были реализованы: поворот, смещение и панорамирование.
Камера расположена в определенной точке пространства, а также имеет целевую точку. На основании этих двух точек рассчитываются три вектора: вверх, вправо и вперед, которые однозначно определяют направление камеры и образуют оси системы координат камеры (рисунок 4.1).

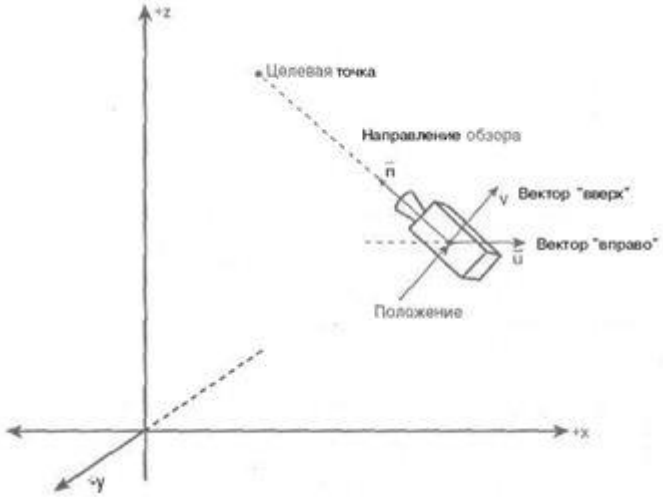


Рисунок 4.1 – Камера [4]

Получаем из МСК координаты точек объекта в координатной системе камеры. Схема представлена на рисунке 4.2.

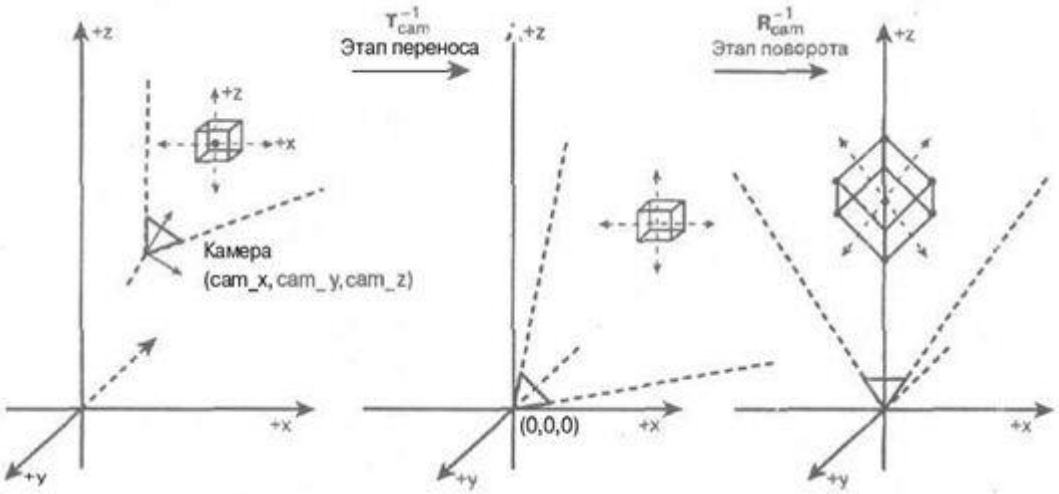


Рисунок 4.2 – Получение координат точек объекта в СК камеры [4]

Основные функции для работы с камерой представленны на рисунках 4.3, 4.4, 4.5.

```
public Matrix GetViewMatrix()
{
    //Преобразовываем вектора так что б они были единичными и перпендикулярными
    Look.Normalize(); //Нормализация вектора

    Up = Vector.Cross(Look, Right); //Векторное произведение
    Up.Normalize();

    Right = Vector.Cross(Up, Look);
    Right.Normalize();

    double x = -Vector.Dot(Right, Pos);  //Скалярное произведение
    double y = -Vector.Dot(Up, Pos);
    double z = -Vector.Dot(Look, Pos);

    return new Matrix(new double[,] { {Right.x, Up.x, Look.x, 0},
                                      {Right.y, Up.y, Look.y, 0},
                                      {Right.z, Up.z, Look.z, 0},
                                      {x, y, z, 1}
                                    });
}
```

Рисунок 4.3 – Функция нахождения матрицы вида

```
public void RotationUpDownSphere(double angle)  // Вращение вокруг вектора Up исходящий из Target
{
    Matrix MatRotation = Matrix.MatrixRotationAxis(Up, angle);
    Right = Right * MatRotation;
    Look = Look * MatRotation;
    Pos = Pos * Matrix.MatrixMove(-Target.x, -Target.y, -Target.z) * MatRotation * Matrix.MatrixMove(Target.x, Target.y, Target.z);
}
public void RotationRightLeftSphere(double angle)  // Вращение вокруг вектора Right исходящий из Target
{
    Matrix MatRotation = Matrix.MatrixRotationAxis(Right, angle);
    Up = Up * MatRotation;
    Look = Look * MatRotation;
    Pos = Pos * Matrix.MatrixMove(-Target.x, -Target.y, -Target.z) * MatRotation * Matrix.MatrixMove(Target.x, Target.y, Target.z);
}
```

Рисунок 4.4 – Функции вращения вокруг точки цели

```
public void pitch(double angle)  //Вращение вокруг правого вектора
{
    Matrix MatRotation = Matrix.MatrixRotationAxis(Right, angle);
    Up = Up * MatRotation;
    Look = Look * MatRotation;
}

public void yaw(double angle) //Вращение вокруг вектора вверх
{
    Matrix MatRotation = Matrix.MatrixRotationAxis(Up, angle);
    Right = Right * MatRotation;
    Look = Look * MatRotation;
}

public void roll(double angle) //Вращение вокруг вектора направления
{
    Matrix MatRotation = Matrix.MatrixRotationAxis(Look, angle);
    Right = Right * MatRotation;
    Up = Up * MatRotation;
}
```

Рисунок 4.5 – Функции вращения камеры вокруг своих осей

Для того чтобы получить координаты точки, представленной в мировых

координатах, в координатах камеры необходимо эту точку умножить на матрицу камеры трансформации из МСК. Точку следует представить в однородных координатах. Матрица трансформации камеры приведена на рисунке 4.6. В этой матрице U – вектор «вправо», V – вектор «вверх», N – вектор «вперед», camPos – позиция камеры в МСК.

$$\begin{pmatrix} Ux & Vx & Nx & 0 \\ Uy & Vy & Ny & 0 \\ Uz & Vz & Nz & 0 \\ camPos\,U & camPos\,V & camPos\,N & 1 \end{pmatrix}$$

Рисунок 4.6 – Матрица камеры трансформации из МСК в СК камеры

После получения координат объекта в системе координат камеры получаем аксонометрическую проекцию объекта на плоскости проекций камеры. На рисунке 4.7 приведена упрощенная двумерная проекция плоскости обзора на плоскость xz. Примем угол обзора FOV равным 90 градусов, а расстояние от камеры до плоскости проекций d равным 1.

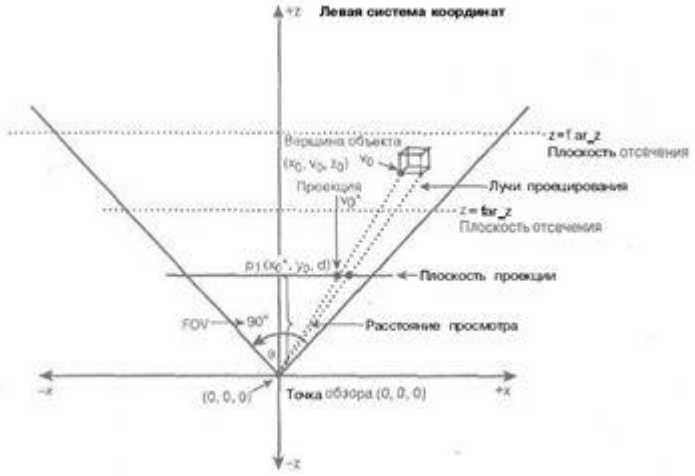


Рисунок 4.7 – Упрощенная двумерная проекция плоскости обзора на плоскость xz [4]

На рисунке 4.8 показан пример построения аксонометрического преобразования.

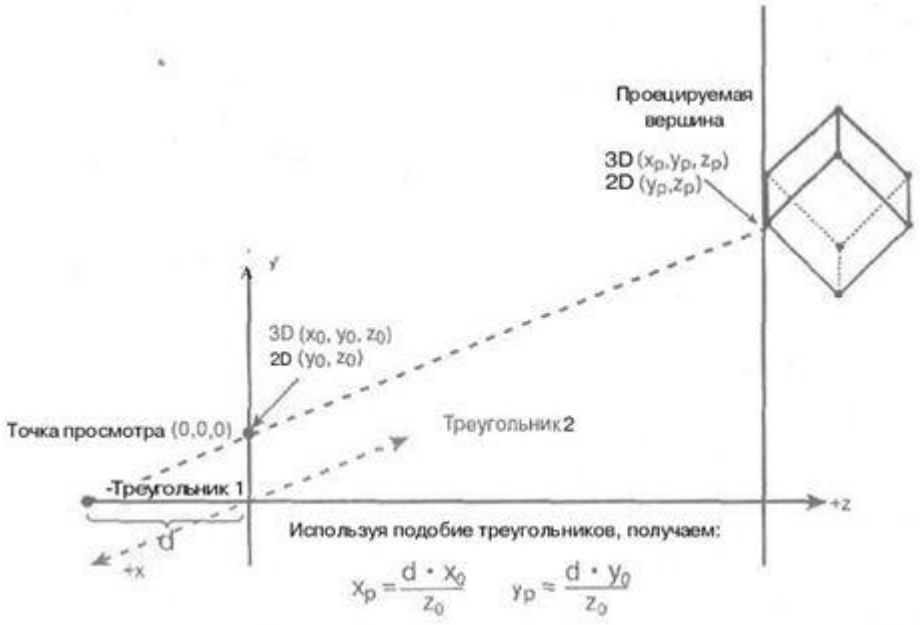


Рисунок 4.8 – Пример построения аксонометрического преобразования [4]

Перед нахождением перспективной проекции, нужно отсечь полигоны по пирамиде видимости (рис. 4.9).

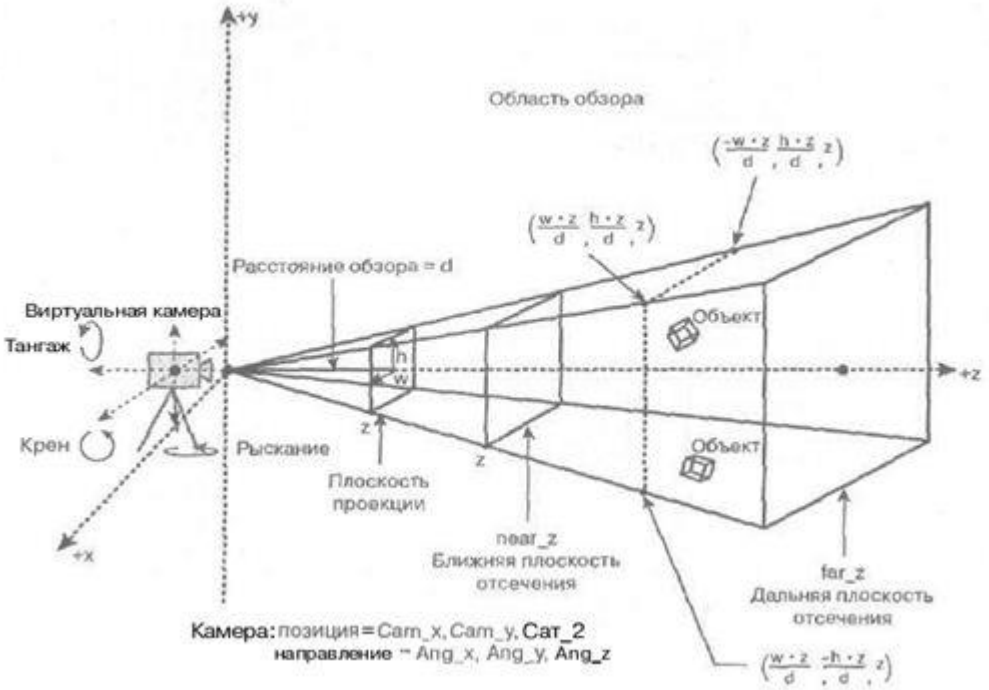


Рисунок 4.9 – Пирамида видимости камеры [4]

Алгоритм перспективного проецирования объекта и отсечение по пирамиде видимости представлен на рисунке 4.10.

```
//В TListPol будут полигоны попавшие в пирамиду видимости
List<Polygon> TListPol = new List<Polygon>();
foreach(Polygon pol in ListPol)
{
    //Попадает ли полигон в область между передней и задней плоскостью пирамиды видимости
     if (pol.p1.z < FarPlane && pol.p2.z < FarPlane && pol.p3.z < FarPlane && pol.p1.z > NearPlane && pol.p2.z > NearPlane && pol.p3.z >
NearPlane)
    {
        double Max1X = pol.p1.z * (Width / 2) / Focus;
        double Max1Y = pol.p1.z * (Height / 2) / Focus;

        double Max2X = pol.p2.z * (Width / 2) / Focus;
        double Max2Y = pol.p2.z * (Height / 2) / Focus;

        double Max3X = pol.p3.z * (Width / 2) / Focus;
        double Max3Y = pol.p3.z * (Height / 2) / Focus;

        //Попадает ли полигон в область между боковыми, верхней и нижней плоскостями пирамиды видимости
        if (pol.p1.x < Max1X && pol.p1.x > -Max1X && pol.p1.y < Max1Y && pol.p1.y > -Max1Y &&
            pol.p2.x < Max2X && pol.p2.x > -Max2X && pol.p2.y < Max2Y && pol.p2.y > -Max2Y &&
            pol.p3.x < Max3X && pol.p3.x > -Max3X && pol.p3.y < Max3Y && pol.p3.y > -Max3Y)
        {
            TListPol.Add(pol);
         }
    }
}
//Находим проекции каждой точки всех полигонов и записываем в ListPoints
List<PointF[]> ListPoints;
Foreach(Polygon pol in TListPolygon)
{
    PointF[] points = new PointF[3];
    points[0].x = Width / 2 + Focus * pol.p1.x / p.z; //Width – ширина окна, Height – высота окна, Focus – фокусное растояние
    points[0].y = Height / 2 - Focus * pol.p1.y / p.z;

    points[1].x = Width / 2 + Focus * pol.p2.x / p.z;
    points[1].y = Height / 2 - Focus * pol.p2.y / p.z;

    points[2].x = Width / 2 + Focus * pol.p3.x / p.z;
    points[2].y = Height / 2 - Focus * pol.p3.y / p.z;
    ListPoints.Add(points);
}
```

Рисунок 4.10 – Алгоритм перспективного проецирования

5. Программная реализация

Пример ввода параметров объекта представлен на рисунке 5.1

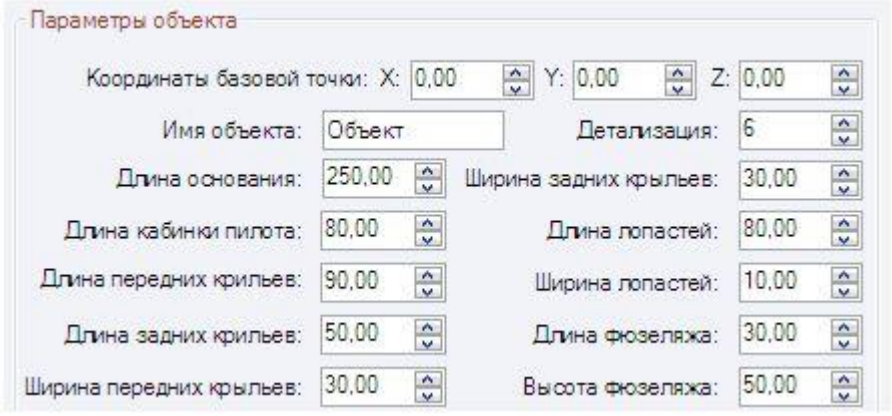


Рисунок 5.1 – Ввод параметров объекта

Примеры работы программы представлены на рисунках 5.2, 5.3, 5.4, 5.5.

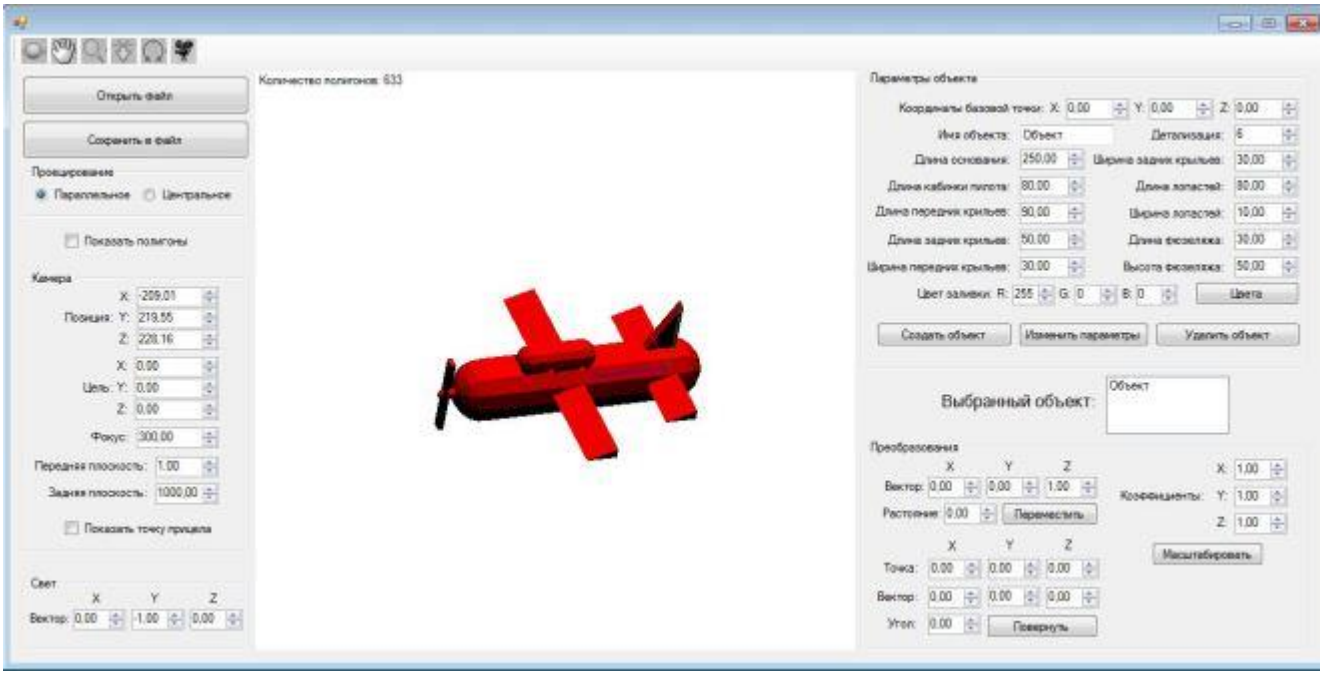


Рисунок 5.2 – Работа программы, пример 1

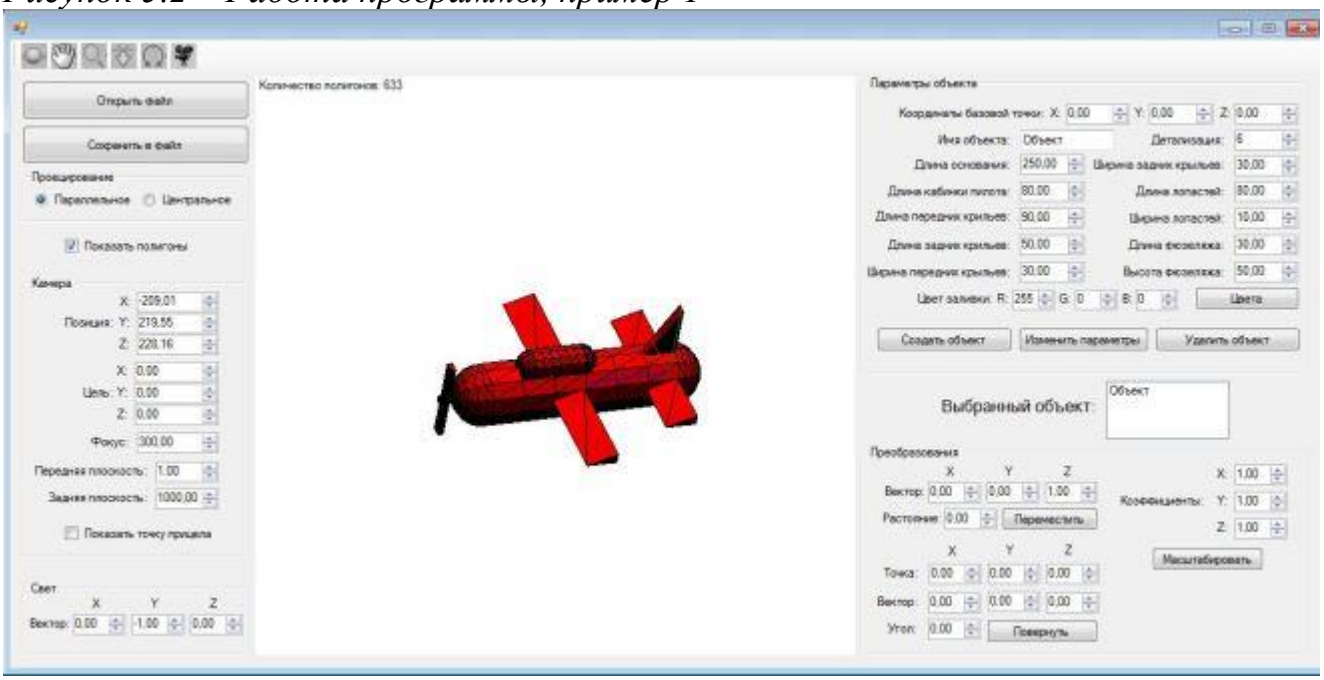


Рисунок 5.3 – Работа программы, пример 2

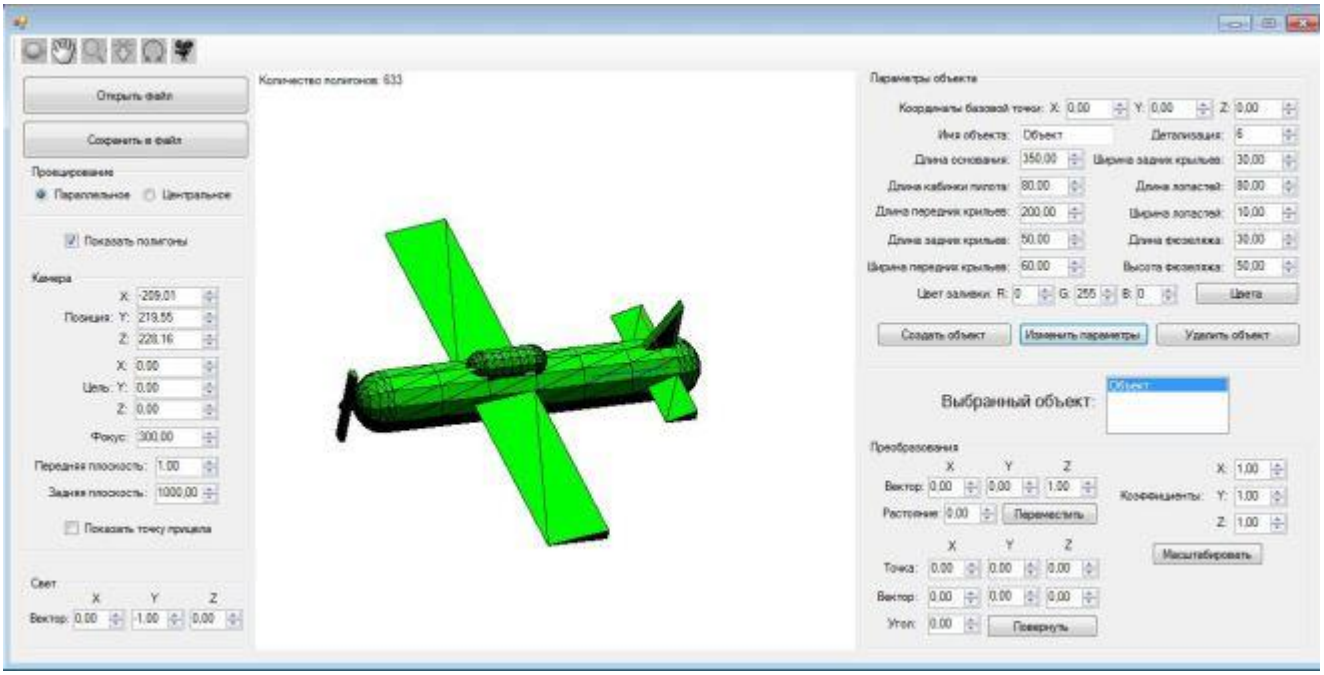


Рисунок 5.4 – Работа программы, пример 3

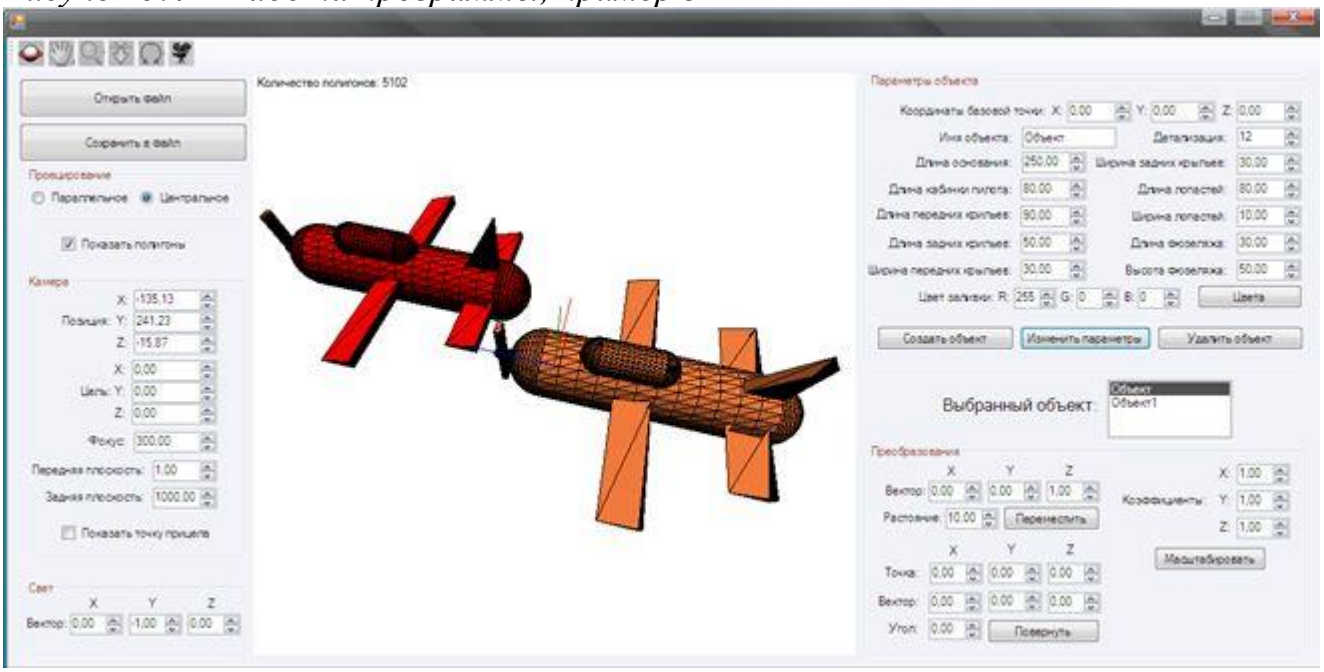


Рисунок 5.5 – Работа программы, пример 4

Выводы

В ходе разработки проекта была создана программная система, которую можно использовать как наглядный пример проектирования простейших трехмерных систем по визуализации некоторых объектов. Также система позволяет продемонстрировать на примере работу некоторых основных алгоритмов компьютерной графики, запрограммированных на языке высокого уровня.

В разработанной системе обеспечено наличие настраиваемого аппарата проецирования, реализовано параллельное и центральное проецирование. Реализована возможность управления камерой, наличие средств сохранения и загрузки параметров объекта.

Достоинствами данной системы является ее простота, наглядность, достаточно удобный интерфейс и простота представления сходных и выходных данных

программы. Недостаткам программы является ее низкая защищенность внешним воздействиям и большой размер кода.

Полезные ресурсы

1. Роджерс Д. «Алгоритмические основы машинной графики»: Пер. с англ.- М.:Мир, 1989.-512с.
2. Рождерс Д., Адамс Дж. «Математические основы машинной графики»: Пер. с англ.- М.: Мир, 2001.-604с.
3. Вангер Ф.С. «Справочник по трехмерному моделированию», СПб, 2002г.
4. Ламот Андре. Программирование трехмерных игр для Windows. Советы профессионала по трехмерной графике и растеризации.: Пер. с англ.–М.: Издательский дом «Вильяме», 2004.-1424с.: ил.- Парал. тит. англ.
5. Френк Д. Луна «Введение в программирование трехмерных игр с DirectX 9.0», Wordware Publishing, 2003.
6. Alan Watt «3D Computer Graphics» 3-е изд. Addison-Wesley, 2000
7. Форум для разработчиков игр и программистов 3D графики [Электронный ресурс]: – Режим доступа: http://www.gamedev.ru


```
unit UTiGr;{Абдулов Тимур Рифович 2015 год Email hostingurifa@gmail.com   .
;INFO
;Site    https://sites.google.com/site/timpascallib/
;Youtube https://www.youtube.com/watch?v=N_PZVx062bo
;Google+
https://plus.google.com/u/0/+%D0%A2%D0%B8%D0%BC%D1%83%D1%80%D0%90%D0%B1%D0%B4%D1%83%D0%BB%D0%BE%D0%B2/posts
;GIST    https://gist.github.com/MisterTimur/30a217abcefc812be744
;-------------------------------------------------------------------------------}
{$mode objfpc}{$H+}
interface
uses  Windows,Classes, SysUtils,Forms,Graphics;
Const TiMaxKOlPix=4000000; // Максимальное количкество  пикселей
Type  TiImg=Class  // Класс для работы с графикой
  BitMap:TBitMap;
  info:TBitMapInfo;
  Pixels:Array[0..TiMaxKOlPix] of LongWord;
  Function    RePiX(X,Y:LongWord):LongWord;
  Procedure   WrPix(X,Y,C:LongWord);
  Procedure   ReCan(C:TCanvas);
  Procedure   WrCan(C:TCanvas);
  Procedure   ReEcr;
  Constructor Create;
end;
implementation
Procedure   TiImg.ReCan(C:TCanvas);// Читает изображение
begin
      BitMap.Height:=c.Height;
      BitMap.Width:=c.Width;
```

```
      BitMap.PixelFormat:=pf24bit;

      with info.bmiHeader do begin
      biWidth:=c.Width;
      biHeight:=c.Height;
      biSize:=SizeOf(TBITMAPINFOHEADER);
      biCompression:=BI_RGB;
      biBitCount:=32;
      biPlanes:=1;
      biSizeImage:=0;
      end;

      BitBlt(BitMap.canvas.Handle,0,0,C.Width, C.Height,c.Handle,0,0,SRCCOPY);
      GetDIBits(BitMap.Canvas.Handle,BitMap.Handle,0,BitMap.Height-
1,addr(Pixels),info,DIB_RGB_COLORS);

end;
Procedure   TiImg.WrCan(C:TCanvas);// Записывает изображение
var
ORect,IREct:TRect;
begin

ORect.Left:=0;
ORect.Top:=0;
ORect.Right:=C.Width;
ORect.Bottom:=c.Height;

IRect.Left:=0;
IRect.Top:=0;
IRect.Right:=BitMap.Width;
IRect.Bottom:=BitMap.Height;


SetDIBits(BitMap.Canvas.Handle,BitMap.Handle,0,BitMap.Height-
1,addr(Pixels),info,DIB_RGB_COLORS);
c.CopyRect(orect,BitMap.Canvas,irect);
end;
Procedure   TiImg.ReEcr; // итает каритнку с экрана
begin
      BitMap.Height:=Screen.Height;
      BitMap.Width:=Screen.Width;
      BitMap.PixelFormat:=pf24bit;

      with info.bmiHeader do begin
      biWidth:=Screen.Width;
```

```
      biHeight:=Screen.Height;
      biSize:=SizeOf(TBITMAPINFOHEADER);
      biCompression:=BI_RGB;
      biBitCount:=32;
      biPlanes:=1;
      biSizeImage:=0;
      end;

      BitBlt(BitMap.canvas.Handle,0,0,Screen.Width,
Screen.Height,GetDC(0),0,0,SRCCOPY);
      GetDIBits(BitMap.Canvas.Handle,BitMap.Handle,0,BitMap.Height-
1,addr(Pixels),info,DIB_RGB_COLORS);


end;
Constructor TiImg.Create;
begin
BitMap:=TBitMap.Create;
end;
Function    TiImg.RePiX(X,Y:LongWord):LongWord;// Читает цвет пикселя
begin
  RePiX:=Pixels[(BitMap.Width*Y)+X];
end;
Procedure   TiImg.WrPiX(X,Y,C:LongWord);// Записывает цвет пикселя
begin
  Pixels[(BitMap.Width*Y)+X]:=C
end;
end.
```

ПРАКТИЧЕСКОЕ ЗАНЯТИЕ № 5. БИБЛИОТЕКА OPENGL.

Цель работы: освоить подключение и работу с OpenGL в Lazarus под Windows.

OpenGL (Open Graphics Library — открытая графическая библиотека) — спецификация, определяющая независимый от языка программирования кросс-платформенный программный интерфейс для написания приложений, использующих двумерную и трёхмерную компьютерную графику.
Включает более 250 функций для рисования сложных трёхмерных сцен из простых примитивов. Используется при создании компьютерных игр, САПР, виртуальной реальности, визуализации в научных исследованиях. На платформе Windows конкурирует с Direct3D.

Основные возможности OpenGL

Что предоставляет библиотека в распоряжение программиста? Основные возможности:

- Геометрические и растровые примитивы. На основе геометрических и растровых примитивов строятся все объекты. Из геометрических примитивов библиотека предоставляет: точки, линии, полигоны. Из растровых: битовый массив(bitmap) и образ(image)
- Использование B-сплайнов. B-сплайны используются для рисования кривых по опорным точкам.
- Видовые и модельные преобразования. С помощью этих преобразований можно располагать обьекты в пространстве, вращать их, изменять форму, а также изменять положение камеры из которой ведётся наблюдение.
- Работа с цветом. OpenGL предоставляет программисту возможность работы с цветом в режиме RGBA (красный-зелёный-синий-альфа) или используя индексный режим, где цвет выбирается из палитры.
- Удаление невидимых линий и поверхностей. Z-буферизация.
- Двойная буферизация. OpenGL предоставляет как одинарную так и двойную буферизацию. Двойная буферизация используется для того, чтобы устранить мерцание при мультипликации, т.е. изображение каждого кадра сначала рисуется во втором(невидимом) буфере, а потом, когда кадр полностью нарисован, весь буфер отображается на экране.
- Наложение текстуры. Позволяет придавать объектам реалистичность. На объект, например шар, накладывается текстура(просто какое-то изображение), в результате чего наш объект теперь выглядит не просто как шар, а как разноцветный мячик.

- Сглаживание. Сглаживание позволяет скрыть ступенчатость, свойственную растровым дисплеям. Сглаживание изменяет интенсивность и цвет пикселей около линии, при этом линия смотрится на экране без всяких зигзагов.
- Освещение. Позволяет задавать источники света, их расположение, интенсивность, и т.д.
- Атмосферные эффекты. Например туман, дым. Всё это также позволяет придать объектам или сцене реалистичность, а также "почувствовать" глубину сцены.
- Прозрачность объектов.
- Использование списков изображений.

Дополнительные библиотеки

Для OpenGL существуют так называемые вспомогательные библиотеки.
Первая из этих библиотек называется GLU. Эта библиотека уже стала стандартом и поставляется вместе с главной библиотекой OpenGL. В состав этой библиотеки вошли более сложные функции, например для того чтобы определить цилиндр или диск потребуется всего одна команда. Также в библиотеку вошли функции для работы со сплайнами, реализованы дополнительные операции над матрицами и дополнительные виды проекций.
Следующая библиотека, также широко используемая - это GLUT. Это также независимая от платформы библиотека. Она реализует не только дополнительные функции OpenGL, но и предоставляет функции для работы с окнами, клавиатурой и мышкой. Для того чтобы работать с OpenGL в конкретной операционной системе (например Windows или X Windows), надо провести некоторую предварительную настройку и эта предварительная настройка зависит от конкретной операционной системы. С библиотекой GLUT всё намного упрощается, буквально несколькими командами можно определить окно, в котором будет работать OpenGL, определить прерывание от клавиатуры или мышки и всё это не будет зависеть от операционной системы. Библиотека предоставляет также некоторые функции, с помощью которых можно определять некоторые сложные фигуры, такие как конусы, тетраэдры, и даже можно с помощью одной команды определить чайник!
Есть ещё одна библиотека похожая на GLUT, называется она GLAUX. Это библиотека разработана фирмой Microsoft для операционной системы Windows. Она во многом схожа с библиотекой GLUT, но немного отстаёт от неё по своим возможностям. И ещё один недостаток заключается в том, что библиотека GLAUX предназначена только для Windows, в то время как GLUT поддерживает очень много операционных систем.
Синтаксис команд
type glCommand_name[1 2 3 4][b s i f d ub us ui][v]

(type1 arg1,…,typeN argN)

Таким образом, имя состоит из нескольких частей: Gl это имя библиотеки, в которой описана эта функция: для базовых функций OpenGL, функций из библиотек GLU, GLUT, GLAUX это gl, glu, glut, glaux соответственно

Command_name - имя команды

[1 2 3 4] – число аргументов команды

[b s i f d ub us ui] – тип аргумента:

b - GLbyte байт,

s - GLshort короткое целое,

i - GLint целое,

f - GLfloat дробное,

d - GLdouble дробное с двойной точностью,

ub - GLubyte беззнаковый байт,

us - GLushort беззнаковое короткое целое,

ui - GLuint беззнаковое целое.

Полный список типов и их описание можно посмотреть в файле gl.h.

[v] – наличие этого символа показывает, что в качестве параметров функции используется указатель на массив значений.

Рисование геометрических объектов

Очистка окна.

glClearColor (clampf r, clampf g, clampf b, clampf a)

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)

Вершины и примитивы

Вершина является атомарным графическим примитивом OpenGL и определяет точку, конец отрезка, угол многоугольника и т.д. Все остальные примитивы формируются с помощью задания вершин, входящих в данный примитив. Например, отрезок определяется двумя вершинами, являющимися концами отрезка.

С каждой вершиной ассоциируются ее атрибуты. В число основных атрибутов входят положение вершины в пространстве, цвет вершины и вектор нормали.

Положение вершины в пространстве

Положение вершины определяются заданием ее координат в двух-, трех-, или четырехмерном пространстве (однородные координаты). Это реализуется с помощью нескольких вариантов команды **glVertex***:

void glVertex[2 3 4][s i f d] (type coords)

void glVertex[2 3 4][s i f d]v (type *coords)

Каждая команда задает четыре координаты вершины: x, y, z, w.

Команда **glVertex2*** получает значения x и y. Координата z в таком случае устанавливается по умолчанию равной 0, координата w – равной 1. Vertex3*

получает координаты x, y, z и заносит в координату w значение 1. Vertex4* позволяет задать все четыре координаты.
Для ассоциации с вершинами цветов, нормалей и текстурных координат используются текущие значения соответствующих данных, что отвечает организации OpenGL как конечного автомата. Эти значения могут быть изменены в любой момент с помощью вызова соответствующих команд.

Цвет вершины

Для задания текущего цвета вершины используются команды :
void glColor[3 4][b s i f] (GLtype components)
void glColor[3 4][b s i f]v (GLtype components)

Первые три параметра задают R, G, B компоненты цвета, а последний параметр определяет коэффициент непрозрачности (так называемая альфа-компонента). Если в названии команды указан тип ‘f’ (float), то значения всех параметров должны принадлежать отрезку [0,1], при этом по умолчанию значение альфа-компоненты устанавливается равным 1.0, что соответствует полной непрозрачности. Тип ‘ub’ (unsigned byte) подразумевает, что значения должны лежать в отрезке [0,255].
Вершинам можно назначать различные цвета, и, если включен соответствующий режим, то будет проводиться линейная интерполяция цветов по поверхности примитива.
Для управления режимом интерполяции используется команда
void glShadeModel (GLenum mode)
вызов которой с параметром GL_SMOOTH включает интерполяцию (установка по умолчанию), а с GL_FLAT – отключает.

Нормаль

Определить нормаль в вершине можно, используя команды
void glNormal3[b s i f d] (type coords)
void glNormal3[b s i f d]v (type coords)
Для правильного расчета освещения необходимо, чтобы вектор нормали имел единичную длину. Командой **glEnable(GL_NORMALIZE)** можно включить специальный режим, при котором задаваемые нормали будут нормироваться автоматически.
Режим автоматической нормализации должен быть включен, если приложение использует модельные преобразования растяжения/сжатия, так как в этом случае длина нормалей изменяется при умножении на модельно-видовую матрицу. Однако применение этого режима уменьшает скорость работы механизма визуализации OpenGL, так как нормализация векторов имеет заметную вычислительную сложность (взятие квадратного корня и т.п.). Поэтому лучше

сразу задавать единичные нормали.
Отметим, что команды
void glEnable (GLenum mode)
void glDisable (GLenum mode)
производят включение и отключение того или иного режима работы конвейера OpenGL. Эти команды применяются достаточно часто, и их возможные параметры будут рассматриваться в каждом конкретном случае.
Операторные скобки **glBegin / glEnd**
Мы рассмотрели задание атрибутов одной вершины. Однако, чтобы задать атрибуты графического примитива, одних координат вершин недостаточно. Эти вершины надо объединить в одно целое, определив необходимые свойства. Для этого в OpenGL используются так называемые операторные скобки, являющиеся вызовами специальных команд OpenGL Определение примитива или последовательности примитивов происходит между вызовами команд
void glBegin (GLenum mode);
void glEnd (void);
Параметр mode определяет тип примитива, который задается внутри и может принимать следующие значения: GL_POINTS каждая вершина задает координаты некоторой точки.

- **GL_LINES** каждая отдельная пара вершин определяет отрезок; если задано нечетное число вершин, то последняя вершина игнорируется.
- **GL_LINE_STRIP** каждая следующая вершина задает отрезок вместе с предыдущей.
- **GL_LINE_LOOP** отличие от предыдущего примитива только в том, что последний отрезок определяется последней и первой вершиной, образуя замкнутую ломаную.
- **GL_TRIANGLES** каждая отдельная тройка вершин определяет треугольник; если задано не кратное трем число вершин, то последние вершины игнорируются.
- **GL_TRIANGLE_STRI**P каждая следующая вершина задает треугольник вместе с двумя предыдущими.
- **GL_TRIANGLE_FAN** треугольники задаются первой и каждой следующей парой вершин (пары не пересекаются).
- **GL_QUADS** каждая отдельная четверка вершин определяет четырехугольник; если задано не кратное четырем число вершин, то последние вершины игнорируются.
- **GL_QUAD_STRIP** четырехугольник с номером n определяется вершинами с номерами 2n-1, 2n, 2n+2, 2n+1.
- **GL_POLYGON** последовательно задаются вершины выпуклого

многоугольника.

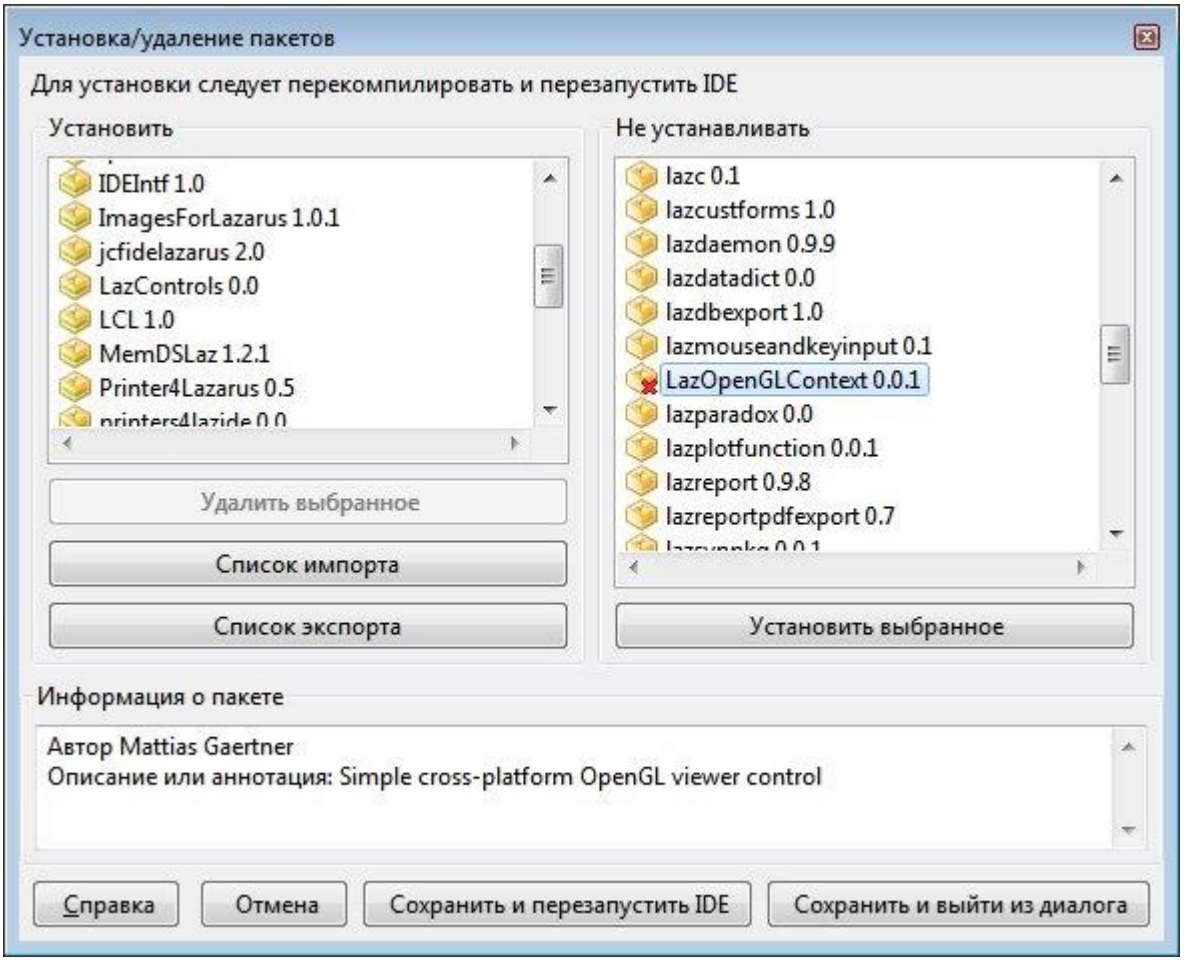
Например, чтобы нарисовать треугольник с разными цветами в вершинах, достаточно написать:

```
glBegin(GL_TRIANGLES);
glColor3f(1.0, 0.0, 0.0); glVertex3f(0.0, 0.0, 0.0);
glColor3ub(0,255,0); glVertex3f(1.0, 0.0, 0.0);
glColor3fv(BlueCol); glVertex3f(1.0, 1.0, 0.0);
glEnd();
```

Установка пакета получения контекста устройства в Lazarus

Как и в любой среде программирования, для того чтобы начать работать с графикой, необходимо получить контекст устройства и связать его с контекстом воспроизведения библиотеки OpenGL. В Lazarus этот процесс осуществляется довольно просто, за счет использования встроенной библиотеки **OpenGLContext** с готовым компонентом **TOpenGLControl**.

Однако компонент **TOpenGLControl**, по — умолчанию, не установлен в среде Lazarus, поэтому необходимо установить данный компонент. Выберите пункт меню "Пакет -> Установить/Удалить пакеты" (Package -> Install/Uninstall Packages), откроется окно установки новых пакетов в среду Lazarus.



В окне установки новых пакетов, в списке неустановленных пакетов, необходимо

найти пакет с именем lazopenglcontext 0.0.1, выбрать его и нажать кнопку "Установить выбранное" (Install selection). После этого необходимо нажать кнопку "Сохранить и перезапустить IDE" (Save and Rebuild IDE), для пересборки Lazarus уже с компонентом **TOpenGLControl**. В окне подтверждения установки нового пакета, нажмите кнопку "Продолжить" (Continue).
Если все проделано верно, то в панели инструментов появится вкладка OpenGl с компонентом TOpenGLControl.
Можно начинать работать с библиотекой.

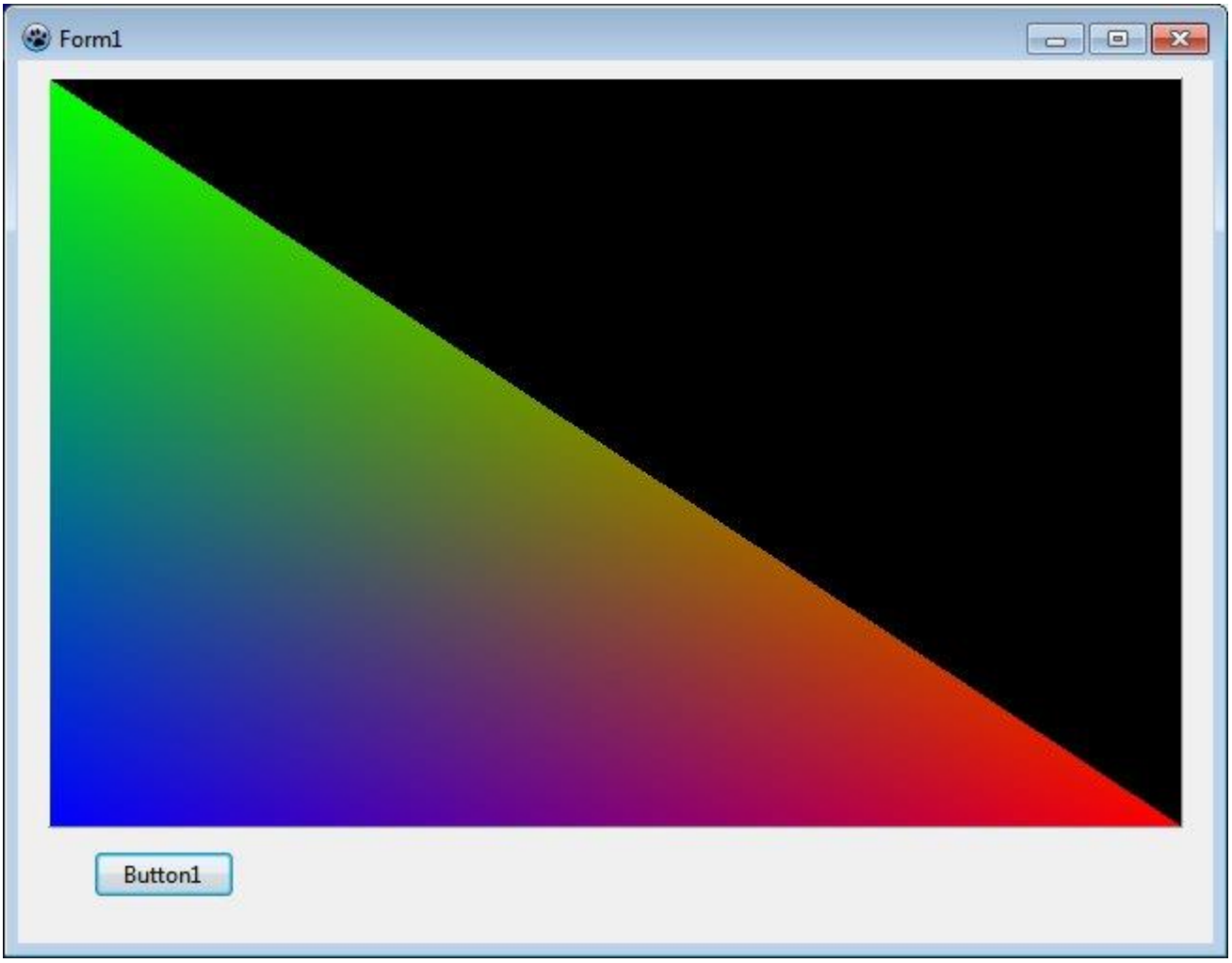
Простейшая программа на OpenGL в Lazarus

Расположите на форме два компонента
Panel1: TPanel
Button1: TButton
пропишите следующий код на события **OnCreate** и **ButtonClick**

```
unit Unit1;

{$mode objfpc}{$H+}

interface

// Подключение библиотек. Для работы С OpenGL необходимы OpenGLContext, GL, GLU
uses
  Classes, SysUtils, FileUtil, Forms, Controls, Graphics, Dialogs, StdCtrls,
  ExtCtrls, OpenGLContext, GL, GLU;

type

  { TForm1 }

  TForm1 = class(TForm)
    Button1: TButton;
    Panel1: TPanel;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
    OpenGLControl1: TOpenGLControl; // Контекст воспроизведения OpenGL
```

```
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}

{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);

begin
  // Создание контекста воспроизведения OpenGL и привязка его к панели на форме
  OpenGLControl1:=TOpenGLControl.Create(Self);
  with OpenGLControl1 do begin
    Name:='OpenGLControl1';
    Align:=alClient;
    Parent:=Panel1;
  end;

end;

procedure TForm1.Button1Click(Sender: TObject);
begin
 glClearColor (0, 0, 0, 0); // цвет фона
 glClear (GL_COLOR_BUFFER_BIT);      // очистка буфера цвета
 glMatrixMode(GL_MODELVIEW);  // Выбор видовой матрицы
 glLoadIdentity();   //  Установка в единичные значения
 glOrtho(0, 1, 0, 1, -1 ,1); // Установка проекции окна
 //glTranslatef(0.5,0.5,0);  // Сдвиг
 //glRotatef(10, 0, 0, 0);   // Вращение
 glBegin(GL_TRIANGLES);      // Рисование треугольника
      glColor3f(1.0,0.0,0.0);     // Красный
      glVertex3f(1,0,0 ); // Верх треугольника (Передняя)
      glColor3f(0.0,1,0.0);
      glVertex3f( 0,1,0);
```

```
      glColor3f(0,0,1);
      glVertex3f( 0,0,0);
 glEnd;
 OpenGLControl1.SwapBuffers; // Отрисовка из буффера
end;

end.
```



http://grafika.me/node/550