

Апрель 2010

№2

[ПРОграммист]

Программирование и алгоритмизация.
Для молодых и активных людей..

Что? Где? Когда?

Или с чего начать программировать

Пишем "Змейку"
средствами GDI на Delphi

Delphi 2010



Работа с
графикой на
канве в Delphi
Урок 3-4

Основные
изменения

И многое другое...

Издается с марта 2010. Выходит ежемесячно
№2, апрель 2010 г.

Редакция:
Utkin, JTG, Алексей Шульга, Сергей Бадло

Дизайн и верстка:
Егор Горохов, Indian, Сергей Бадло

Авторский состав:
Utkin, Ivan32, Владимир Дегтярь,
Денис Пискунов,
Вадим Буренков

Контакты:
Вопросы и авторские статьи направляйте на
maindatacentr@gmail.com
Вопросы и предложения для редакции
reddatacentr@gmail.com
Информационная поддержка
www.programmersforum.ru

Примечание:
Издание некоммерческое. Все материалы,
товарные знаки, торговые марки и логотипы,
упомянутые в журнале, принадлежат их
владельцам. Мнение авторов не всегда
совпадает с мнением редакции. Перепечатка
материалов журнала и использование их в
любой форме, в том числе в электронных СМИ,
возможно только с разрешения редакции.
Формат А4, 58 стр.

Обложка номера:
Использован логотип Embarcadero Technologies.

ТЕМА НОМЕРА

Ждем ваши статьи с.0x02

НЕВЕРОЯТНО, НО ФАКТ

Любопытные факты с.0x03

ТОЧКА ЗРЕНИЯ. ОБЩИЕ ВОПРОСЫ

Рабство программиста с.0x06

Что, где, когда... или с чего начать программировать с.0x10

НОВОСТИ ПРОГРАММИРОВАНИЯ

Основные изменения в языке Дельфи 2010 с.0x13

АЛГОРИТМЫ. ГРАФИКА В DELPHI

GameDEV на Delphi. Делаем змейку с.0x1D

Как работать с графикой на канве в среде Дельфи. Урок 3-4 с.0x28

ЛАБОРАТОРИЯ

Введение в SSE с.0x32

ЮМОР. ХОХМЫ ПРОГРАММИСТОВ

Фразеологизмы с.0x39

Слово редакции

Дорогие друзья! Добро пожаловать в очередной и апрельский выпуск журнала «ПРОграммист» от клуба программистов www.programmersforum.ru. В прошлом месяце дебютировал наш журнал. Не все было гладко, но постепенно все утряслось. Мы, честно говоря, ожидали шквала гневных писем, сообщающих о недочетах, но обошлось все благосклонным отношением со стороны нашего клуба. Спасибо Вам!

Как и обещали, в сегодняшнем выпуске мы продолжим уроки графики от Владимира Дегтяря, пофилософствуем вместе с **Utkin**-ым об эффективности имеющихся механизмов ООП и рассмотрим, что нового преподнесла разработчику Дельфи 2010. Денис Пискунов даст свои советы начинающим «с чего собственно начинать» и как отточить свое мастерство настоящего программиста. В рубрике «Графика» приветствуем новый урок по созданию с азов игры «змейка» от Вадима Буренкова, надеемся совсем не последний. **Ivan_32** поддержал рубрику «Лаборатория». На этот раз, мы рассмотрим базовые принципы работы с расширением SSE. И какой-же апрельский выпуск без курьезных фактов и шуток. Приветствуем новые циклы статей, будьте активнее. Наслаждайтесь выпуском и оставайтесь с нами!

Рубрики журнала (плавающие)

- Новости программирования (новые языки, концепции, среды, дополнительные утилиты)
- Отдел тестирования (оценка удобства использования той или иной средой)
- Общие вопросы (вопросы правового использования, личные мнения о тех или иных механизмах и языках и т.д.)
- Алгоритмы (описание некоторых методов работы без привязки к языку и платформе)
- Юмор (специфические хохмы программеров)
- Реализация (описание различных тонкостей программирования)
- Разработка (циклы статей по созданию программных проектов от этапа постановки задачи до получения работающей программы)
- Переводные материалы (перевод статей по программированию)
- Рубрика про железки / Лаборатория

Периодичность выхода – раз в месяц. Ящик для корреспонденции и вопросов: maindatacentr@gmail.com

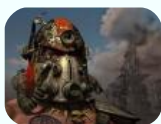
Общие требования к присылаемым материалам

У нас нет категоричных требований к оформлению материалов, но в связи с особенностями верстки (используется свободное ПО «SCRIBUS») и облегчения труда редакторов, есть некоторый желательный минимум:

- статья должна иметь выраженную структуру с разделами и содержать – название статьи, сведения об авторах, экскурс или введение, сведения об используемых средствах разработки, теоретическую и/или практическую часть, заключение (выводы, чего добились) и ресурсы к статье (ссылки на код, интернет-ресурсы, литературу)
- текст статьи в формате MS Word, VK WordPad **или обычным текстовым файлом**, шрифт Arial 10
- все рисунки, таблицы должны иметь упоминание в тексте и иметь подпись
- рисунки (скрины) к статье должны прилагаться в виде отдельных файлов в формате PNG, BMP или TIF
- разделы статьи отделять двумя <ENTER>
- не используйте табуляцию без необходимости, в конечном итоге присланный материал все-равно переводится в текстовый формат
- по присланным материалам автор получает рецензию и корректирует статью согласно замечаниям
- шаблон для написания статьи можно взять [тут](#), бесплатный редактор VK WordPad можно взять [тут](#)

Замечание редактора обязательно должно вызывать реакцию автора. Если автор не согласен с комментарием или исправлением редактора, он должен разъяснить вопрос в виде комментария к статье на форуме «Обсуждение статьи» или на ящик-копилку. Если редактор согласен с тем, что ошибся, вопрос снимается. Если редактор не согласен, решает группа.

С уважением, редакция журнала «ПРОграммист»



В этом месяце данная рубрика посвящена курьезным изобретениям и проишествиям, которые хоть и пользуются сомнительной славой, но кто-знает, может быть в недалеком будущем на них будут смотреть уже без улыбки. И конечно-же немножко ИТ новостей...

p.s.: редакции по-прежнему требуется активный ведущий данного раздела

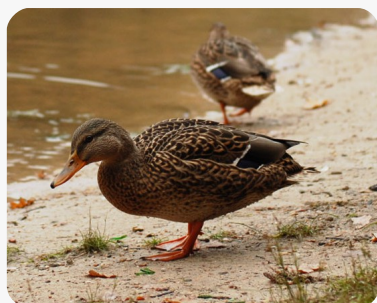
Систему перевода человеческой речи в воспринимаемые животными звуковые сигналы представила компания Google. Для платформы Android через каталог Android Market любой



желающий может загрузить соответствующее приложение «[Translate for Animals](#)».

Космический браузер анонсировала компания Opera (доступен перевод на русский язык). Для доступа к сети задействован специальный протокол INTP (Interplanetary Network Transport Protocol), изображение может проецироваться на сетчатку глаза, при работе с сенсорным экраном в скафандре браузер рассчитывает наиболее вероятную область на которую хочет нажать пользователь. Разработка полностью протестирована на пригодность использование в вакууме.

Прогноз осадков по характеру кровообращения в организме диких уток



РОССИЙСКОЕ АГЕНТСТВО
ПО ПАТЕНТАМ И ТОВАРНЫМ ЗНАКАМ
ЗАЯВКА НА ИЗОБРЕТЕНИЕ

Дата подачи заявки: 2003.05.05 и действует до сих пор.

Дата публикации заявки: 2004.12.10

Заявитель(и): Бурятская государственная сельскохозяйственная академия.

Метод определения сроков и массы выпадения осадков по характеру кровообращения в организме диких уток, прилетевших на данную территорию для вывода птенцов, заключается в определении мест кровоподтеков и интенсивности их на киле грудной кости.

Примечание редакции: птичку-у жалко

Nokia патентует технологию самоподзарядки аккумуляторов, позволяющую продлить время автономной работы мобильных устройств, за счет преобразования кинетической энергии в электрическую. Принцип работы генератора, на примере мобильного телефона, представляет собой следующую схему: плата и аккумулятор располагается на жесткой рамке, прикрепленной к корпусу мобильного телефона посредством одного или нескольких пьезоэлектрических элементов. Таким образом любые перемещения и колебания подвижного отсека вызывают генерацию электрических зарядов, перенаправляющихся через контроллер в аккумулятор.

Робо-унитаз поднимающий и опускающий сиденье в соответствии с положением тела, которое отслеживается с помощью инфракрасного луча. После выхода объекта из зоны слежения крышка автоматически закрывается.



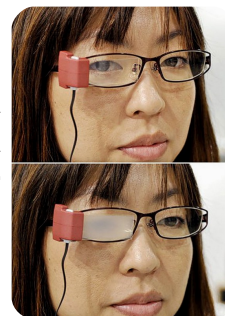
Комфорт для любимой кошки создаст специальный пульт для кошек, позволяющий им включать смыв в унитазе.



Деактивацию и перевод в режим сна заигравшихся в видеоигры детей осуществляет устройство PediSedate, путем введения наркоза через гарнитуру.



Очки для моргания для избежание сухого глаза во время пребывания за компьютером. Когда пользователь сохраняет глаза открытыми в течение 5 секунд, объектив затуманивается.



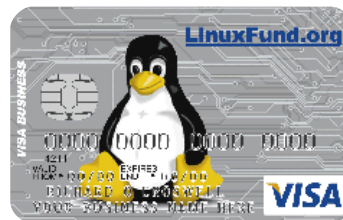
Почувствуй себя Бондом с помощью мегадевайса JetLev-Flyer, позволяющего парить над водой и землей благодаря двум мощным водяным струям.



Прозрачный бетон на основе обычного разработали венгерские инженера-исследователи. Секрет заключается в насыщенных оптических волокнами блоков, пропускающих свет.



Открыто казино linuxfund.org, весь получаемый доход которого будет тратиться на финансирование свободных проектов <http://linuxfund.org>.



Регистрация доменов в зоне .ru с 1 апреля немного усложнится. Теперь координационный центр может прекратить делегирование доменов в зоне .ru, владельцы которых не прошли паспортную идентификацию. Регистратор может в любой момент запросить у владельца документальные подтверждения личности.

.RU

Если домен подтвержден ранее, то на повторное подтверждение у него всего 10 дней, если домен в состоянии unverified, срок сокращается до 3 дней, после чего делегирование будет приостановлено. Данные меры могут быть применены и по требованию правоохранительных органов. По заявлениям, это должно помочь в борьбе сквоттерами, правда, не понятно каким образом.

В КНДР выпустили собственную ОС на основе Linux «Красная звезда». Работа над проектом была начата еще в 2002 году, но до сих пор ОС была скрыта от мира. Цена данной ОС приблизительно 5 долларов. Ввиду малого количества совместимых программ, «Красная звезда» еще не получила распространения. Компьютерные



специалисты из Института научной и технологической политики в Южной Корее, тестирующие ОС, говорят, что в систему встроены функции, позволяющие следить за действиями пользователей...

Продажи нового iPad от Apple стартовали

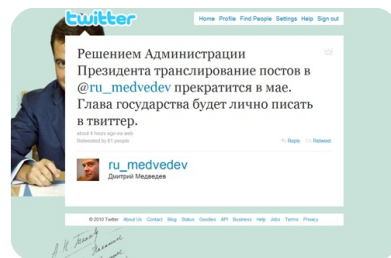


3 апреля. iPad – планшетный компьютер под управлением iPhone OS, оснащенный процессором

Apple A4 с тактовой частотой 1 ГГц, 256 МБ оперативной памяти и от 16 до 64 ГБ постоянной памяти. Что касается сети, то на данный момент iPad работает только с BT 2.1 и Wi-Fi, но в скором времени выйдет модель, поддерживающая также и 3G сети. Пока что стоимость данного девайса фигурирует в пределах от 500\$ до 700\$.

Дмитрий Медведев будет лично вести twitter. Ранее в микроблоге – ru_medvedev транслировались краткие версии официальных сообщений главы государства.

В начале этого месяца в этом блоге появилось следующее сообщение: «Решением администрации



Президента, транслирование постов с @ru_medvedev прекратиться в мае. Глава государства будет лично писать в твиттер». Для Дмитрия Медведева такой поступок не удивителен, учитывая, что у него есть свое сообщество в Живом Журнале и свой канал на Youtube.

Первый телевизор с Google Android

выпущен совместно Google и шведской компании People of Lava с названием Scandinavia и диагоналями 37, 42 и 55 дюймов. Телевизор поддерживает Full HD, обеспечивает доступ к Google Maps, Twitter, Youtube, Facebook. Ориентировочная цена 2500\$.





Статья посвящена стереотипному мышлению о высокой эффективности имеющихся механизмов ООП. Для программистов использующих ООП языки. Цель статьи – дать углубленное представление о базовых концепциях объектно-ориентированного программирования...

by Utkin www.programmersforum.ru

Каждый из тех, кто знаком с принципами ООП, прекрасно знает о тех преимуществах, удобствах и больших плюсах, которые оно представляет программистам. Но так ли это на самом деле? Все познается в сравнении, преимущества по сравнению с чем?

Что такое ООП?

Да, что такое ООП? Несмотря на значительное время существования данной концепции точного определения ООП не существует и по сей день. Есть определения ООП в рамках конкретных языков программирования, но все они различны, имеют свою терминологию, механизмы использования, особенности реализации и т.д. Любой учебник по ООП даст Вам либо определение в привязке к языку программирования, либо весьма туманное объяснение или же вовсе, с места в карьер, речь пойдет о принципах, определениях класса и объекта и т.д.

Не имея точного определения, обучаемый, словно Алиса проваливается в кроличью нору нового для него мира.

ООП программист – рядовой муравей, увеличивающий всемирную энтропию путем написания никому не нужного кода.

<http://absurdopedia.wikia.com/wiki/ООП>

Особенно остро это ощущается, если уже имел навыки программирования. Но ощущения субъективны, рассмотрим основные концепции ООП под другим углом зрения.

Базовые понятия ООП

Их все знают, это:

- . инкапсуляция
- . наследование
- . полиморфизм

Сейчас также добавляют еще понятие абстракции данных. Рассмотрим их всех по порядку...

Инкапсуляция – это принцип, согласно которому любой класс должен рассматриваться как черный ящик. Пользователь класса должен видеть и использовать только интерфейсную часть класса (т.е. список декларируемых свойств и методов класса) и не вникать в его внутреннюю реализацию. Поэтому данные принято инкапсулировать в классе таким образом, чтобы доступ к ним по чтению или записи осуществлялся не напрямую, а с помощью методов. Принцип инкапсуляции* (теоретически) позволяет минимизировать число связей между классами и, соответственно, упростить независимую реализацию и модификацию классов.

* определение взято из Википедии

http://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование

Но, инкапсуляция не ноу-хау ООП, она существовала и ранее – это обычное описание функций и процедур. Пример на Паскале:

```
Function Sum (a, b: Integer): Integer;  
Begin  
  Result := a+b  
End;
```

должен знать, что имя функции «Sum», что она принимает два параметра типа Integer (важен также порядок их следования), возвращает также Integer, а вот каким образом проводится сложение – это уже совершенно безразлично. Еще пример:

```
Type  
  TMas = record  
Data: Array of Integer;  
Count: Integer;  
End;  
  
Procedure Sort (var Mas: TMas);
```

Чтобы использовать функцию, мне не обязательно знать, как она устроена (и в ряде случаев такое знание даже противопоказано), достаточно лишь описание интерфейса «**Function** Sum (a, b: Integer): Integer». Я

Теперь, чтобы отсортировать такой вот массив, мне не нужно знать, как он устроен, сколько в нем элементов и т.д. Я просто передам его процедуре «Sort».

Так чем-же примечательна инкапсуляция? Справедливости ради, надо отметить, что инкапсуляция, как описание указанного выше

явления, получила признание только в ООП. Потому-что такое определение там является одним из главных особенностей построения программ. **Никаких преимуществ, в сравнении с теми же структурным или функциональным программированиями, инкапсуляция в ООП не несет.**

Однако, на этом инкапсуляция не заканчивается...

Сокрытие данных (взят из книги Тимоти Бадд «ООП в действии») – неотделимая часть ООП, управляющая областями видимости. Является логическим продолжением инкапсуляции. Целью сокрытия является невозможность для пользователя узнать или испортить внутреннее состояние объекта. Но это тоже существует в структурном программировании:

```
Function fact(x: integer): integer;  
var  
  i, n: Integer;  
begin  
  n:=1;  
  
  for i:= 1 to x do  
    begin  
      n:=n*i;  
    end;  
  fact:=n;  
end;
```

Разве я могу получить доступ к «i» и «n» не в рамках данной функции? Также стоит еще раз внимательно почитать определение – **невозможность для пользователя**. Если речь идет о программисте, то он испортить может все и вся и никакое сокрытие данных Вам не поможет, по одной простой причине – раз имеются данные, то также и имеются некоторые механизмы для их использования. Поэтому область видимости не защищает данные от ошибок в методах данного класса.

Пример на Дельфи:

Опытный программист уже догадался, о чем идет речь – любое обращение к MyData вызовет ошибку, поскольку перед использованием такие объекты нуждаются в инициализации (между прочим одна из распространенных ошибок начинающих программистов). Так что же дают

```
Type  
  MyClass = class (TObject)  
Protected  
  MyData: TStringList;
```



```
Private
Public
  Constructor Create;
  Destructor Destroy;
End;

Constructor MyClass.Create;
Begin
  Inherited Create
End;

Destructor MyClass.Destroy;
Begin
  Inherited Destroy
End;
```

сокрытие данных в рамках ООП предназначено для **программирования программиста**, а не реализации алгоритма. Я говорю о сокрытии данных, потому-что я боюсь допустить ошибку, которая в рамках структурного программирования **не может возникнуть в принципе**. Компилятору безразлично, в какой секции находится поле, он выдаст код в любом случае (речь идет о Дельфи версии 7), **все ограничения видимости введены для программиста**.

Если Вы думаете, что в С++ это невозможно, то сильно ошибаетесь и виной тому указатели (а в С++ указатели не менее важный механизм, нежели механизмы ООП). Имея указатель на класс, можно не только прочесть его приватные поля, но и модифицировать их:

Инкапсуляция тесно связана с таким понятием как **абстрагирование**. Это придание объекту характеристик, которые отличают его от всех других объектов, четко определяя его концептуальные границы. Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов. И здесь снова модули и структуры решают эту задачу без участия ООП – интерфейсы функций, процедур и модулей могут **полностью** скрывать внутреннее представление по реализации тех или иных задач. Сомневаетесь? Вот пример взаимодействия модулей (Дельфи). Поместите на форму одну кнопку:

игры с областями видимости?

Если-же говорить о методах других объектов, то доступа к «MyData» они не получают, но, согласно принципам ООП, они и **не должны его получать**. Иными словами, «MyData» никогда не должен находиться в секции Public (кстати, Дельфи это позволяет). Доступ к полям класса **всегда** должен осуществляться через методы либо свойства.

То есть, здесь **должна быть** аналогия с функциями и процедурами структурного программирования – объявление структур данных должно осуществляться аналогично секции «var». Это очень важный момент –

```
Class Sneaky
{
private:
int safe;
public:

// инициализировать поле safe
// значением 10
Sneaky () { safe = 10; }
Int &sorry() { return safe; }
}
```

И далее:
Sneaky x;
x.sorry () = 17;

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Unit2, StdCtrls;
```

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation
{$R *.dfm}

procedure TForm1.Button1Click(Sender: TObject);
begin
  AddX();
  Button1.Caption:=IntToStr(GetX());
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Init();
end;

end.
```

И подключите второй модуль:

```
unit Unit2;

interface

  procedure Init();
  procedure AddX();
  function GetX(): Integer;

implementation

var x: Integer;

procedure Init();
begin
  x:=0;
end;

procedure AddX();
```

```
begin
  x:=x+1;
end;

function GetX(): Integer;
begin
  result:=x;
end;

end.
```

Выполните программу, понажимайте на кнопку. Теперь попробуйте получить доступ к «X» без использования функций и процедур из Unit2. Модуль **хранит** в себе данные, **скрывает** их представление и **ограничивает** к ним доступ. При этом в Unit2 нет и намека на класс (и не думайте, что Вы сможете прочесть или изменить «X» обычными процедурами и функциями Unit1 (не из формы)).

Наследование – один из четырех важнейших механизмов объектно-ориентированного программирования, позволяющий описать новый класс на основе уже существующего (родительского), при этом свойства и функциональность родительского класса заимствуются новым классом. Не то-чтобы я не согласен с таким определением, но давайте посмотрим очередной пример:

```
Type
  TMas = record
    Data: Array of Integer;
    Count: Integer;
  End;

Type
  TGroupMas = record
    Data: Array of TMas;
    Count: Integer;
  End;

Procedure Sort (var Mas: TMas);
```

Разве для построения «TGroupMas» я не опираюсь на «TMas»? Я ведь мог написать определение «TGroupMas» с нуля, но, в моем случае, каждый элемент массива Data структуры «TGroupMas» является ни кем иным как «TMas». Более того, мне ничего не стоит написать процедуру сортировки указанного элемента «TGroupMas». Все что требуется – это лишь правильно передать параметры процедуре «Sort». Таким образом, я описываю новую структуру «TGroupMas» данных на основании существующей «TMas» и я мог создать процедуру сортировки элемента массива на основании «Sort»:

При этом, согласно определению наследования, я заимствую все свойства «TMas» и функциональность «Sort». И все это в рамках структурного программирования, никакого ООП для этого не требуется. Вот пример, который часто любят давать в учебниках ООП (язык программирования Дельфи):

```
Procedure SortItem (var GroupMas:
  TGroupMas; Index: Integer);
Begin
  Sort (GroupMas.Data[Index]);
End;
```

```
Type
TMaterial = record
  Massa: Integer;
End;

Type
```

И пусть в меня кинут камнем, если программист в данной иерархии не является млекопитающим и не обладает его свойствами (имеет массу, принадлежит к определенному виду):

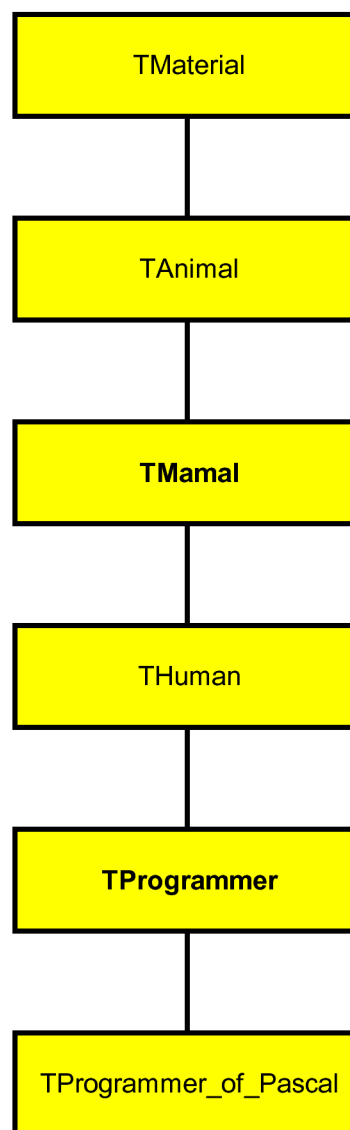
```
TAnimal = record
  Material: TMaterial;
  Sort: String;
End;

Type
TMamal = record
  Animal: TAnimal;
  Family: String;
End;

Type
THuman = record
  Mamal: TMamal;
  Race: String;
  Floor: String;
  Name: String;
  Family: String;
End;

Type
TProgrammer = record
  Human: THuman;
  Sertificate: String;
End;

Type
TProgrammer_of_Pascal = record
  Programmer: TProgrammer;
  IDE: String;
End;
```



Для использования механизма наследования не требуется использование объектно-ориентированного программирования. Достаточно, чтобы язык имел возможность организации структур данных определяемых программистом.

Множественное наследование полностью аналогично – я могу определить новую структуру: цвет глаз и включить ее в TMamal, и тогда «программист» обретет новые свойства. Кстати, множественное наследование одна из самых известных мозолей ООП, но именно поэтому о ней мы больше упоминать не будем. Цель данной статьи как раз показать те, моменты, о которых говорить не любят.

Полиморфизм – взаимозаменяемость объектов с одинаковым интерфейсом. Язык программирования поддерживает полиморфизм, если классы с одинаковой спецификацией могут иметь различную реализацию – например, реализация класса может быть изменена в процессе наследования. Кратко смысл полиморфизма можно выразить фразой: «Один интерфейс, множество реализаций».

Можно сказать, что полиморфизм – одна из самых загадочных концепций ООП. Итак, в некотором роде полиморфизм тесно связан с наследованием. Вы получаете некоторые методы от родительского класса и можете переопределить их функциональность. Адекватного

механизма в структурном программировании не существует, но какие собственно выгоды дает полиморфизм? Итак, это выглядит следующим образом – некий класс (допустим «программист») имеет в своем составе метод (допустим «пить чай»). Программист на Яве переопределяет метод и пьет чай марки Ява под именем метода «пить чай». То есть, мы подразумеваем, что когда программист на Ява пьет чай, то он пьет чай марки Ява.

Стандартно, формально, но теперь вопрос: «какие в этом плюсы?» Очевидно, они следующие:

1. Уменьшение сложности программ
2. Позволяет повторно использовать один и тот же код
3. Позволяет использовать в потомках одинаковые имена методов для решения схожих задач.

Не густо. Ну что-же, рассмотрим каждый пункт...

Первый – честно говоря, так и не увидел, в чем это выражается (хотя упоминается об этом практически везде), концепция структур и модулей достаточна для решения всех задач, которые могут быть решены полиморфизмом. С другой стороны, многочисленные одноименные методы усложняют программу. Результатом полиморфизма являются объекты, которые имеют и одноименные методы и могут работать с разными типами данных. Использование одноименных методов в таком случае не так тривиально, как хотелось бы. Пришлось срочно создавать новую концепцию (вернее сказать, воровать концепцию, поскольку ООП суть ряда заимствований от других парадигм) – RTTI. Проще всего ее можно представить как информацию о типе объекта во время выполнения программы. То есть перед запуском нужного метода используется явное определение того типа данных, с которым предстоит работать. Обычно такая ситуация возникает только в сложных программных объектах, но и полиморфизм в объектах с несложным поведением не имеет смысла и может быть заменен даже обычными операторами селекторов (например «case» в Дельфи) и введением дополнительных переменных. Более того, RTTI перечеркивает абстрагирование – для решения задачи динамически знать тип данных противопоказано, это увеличивает сцепляемость объектов – их сложнее заменять, переносить, модернизировать. RTTI также уменьшает такую возможность полиморфизма, как использование обобщенных алгоритмов (о параметризации речь ниже).

Второй – об этом уже упоминалось:

```
Procedure SortItem (var GropuMas:
TGroupMas; Index: Integer);
Begin
  Sort (GroupMas.Data[Index]);
End;
```

Я же не пишу функцию сортировки снова, просто передаю ей нужные параметры. Налицо, явное использование уже существующего ранее кода и совершенно без единого класса, поэтому никакого преимущества в сравнении скажем с функциональным программированием здесь нет. Если же речь идет о параметризации, то с

каждым новым набором параметров генерируется новая версия этой же функции, поэтому код в таком случае используется не повторно, а каждый раз новый.

Третий – можно написать кучу модулей для решения каждой задачи (все равно в классах каждое решение надо описывать явно) и вызывать их одноименные функции также через точечный синтаксис (в Дельфи), только в обратном порядке – «имя_функции.имя_модуля». И я бы не сказал, что сильно путаюсь в программе, если однотипные действия названы по-разному (причем без разницы два у меня метода или десятков). Если имеется достаточно полное описание методов (те самые интерфейсы, которые ООП также считает своим достижением), и они имеют

Организация тысячи – то же, что и организация одного. Это вопрос организации... / Конфуций

осмысленные имена, то никаких проблем между вызовом метода ProgrammTea и ProgrammJavaTea не возникает. ООП полностью игнорирует такие подходы как правильное оформление кода и правила именования объектов (хотя в нем же использование одноименных методов считается плюсом).

Кроме того, я не считаю, что выбор нужного метода осуществляется компилятором – все действия жестко прописаны в каждом классе, а поскольку любой класс является еще и типом, то он ничего не выбирает, выбирает программист в каком **классе**, какой чай должен пить объект-программист на этапе переопределения родительского метода. Вот я бы задал список методов вообще без привязки к конкретному классу, просто как набор функций и процедур в отдельном модуле, а уж компилятор сам вызывал бы соответствующий метод, тогда я бы согласился с этим утверждением на все 100. Далее в современные библиотеки классов, обычно содержат более 100 классов, которые могут содержать десятки методов, и все их держать в голове никакой полиморфизм еще никогда не помогал... Без разницы сколько мне надо знать 1000 методов или 10 000, все равно их все помнить в любой момент времени нет необходимости.

Что касается параметризации – это действительно мощный механизм. Но с теми же симптомами – это не универсальный рецепт, то есть имеет смысл его применять только в ряде случаев (часть из которых может быть решена комбинаторными алгоритмами) и преимущественно к простым типам. Вот пример алгоритма (C++):

Не трудно догадаться, что она возвращает максимальный элемент из двух указанных, но только в том случае, если программист сможет описать строгое и однозначное сравнение объектов, то есть удобно для типов, которые по умолчанию поддерживаются транслятором. Для сложных типов, определяемых программистом параметризация не дает никаких преимуществ в сравнении с традиционными подходами.

```
// Описание шаблонной функции
template <typename T> T max(T x, T y)
{
    if (x < y)
        return y;
    else
        return x;
}
```

Весьма проблематична параметризация, когда в качестве типа выступают объекты классов, хотя, казалось бы, их строгая иерархия, наследование полей и полиморфизм должны как раз способствовать написанию обобщенных алгоритмов для работы с такими типами. Да это возможно, но требует тщательного продумывания иерархии и написание индивидуальных методов для каждого класса может оказаться предпочтительней.... Здесь решение аналогично RTTI, но проблема не в ООП, это попытка перешагнуть через строгое ограничение типов (наряду с типом Variant в Дельфи). Можно провести следующую аналогию – сначала перед бегуном понаставили барьеров и сказали, что теперь он будет учувствовать в беге с препятствиями, а потом пытаются научить его прыгать, выдают ему специальные кроссовки, рекомендуют методики тренировки...

Также не все гладко с **перегрузкой** операторов – трансляторы языков программирования часто не могут предоставить нужные средства идентификации перегруженных операторов (а те, что могут, сложны и не так эффективны, порождают большой и медлительный код), что вызывает двусмысленность в толковании всего выражения. Самый простой пример – операция «вычитание». Дело в том, что она существует как минимум в двух видах: это унарная операция и бинарная («-x» и «x-y»). А в некоторых языках есть еще ее различные формы: инфиксная, постфиксная и т.д. Далее, необходимо определить приоритет операции, скажем, для строк определение подобных операций может быть не так очевидно, как для чисел. Не думайте что сложение в этом случае лучше. Например, сложение строк «x + y» не эквивалентно «y + x».

Также перегрузке свойственны общие беды полиморфизма – она не является обязательным элементом программирования (это значит, что нет алгоритмов, которые невозможно реализовать без использования перегрузки), и может привести к обратному результату – не упрощению, а усложнению программы. Подобно механизму RTTI перегрузка увеличивает связность кода – для понимания работы перегруженного оператора требуется знать тип объекта (или требуется его уточнение для понимания программистом) используемого в конкретной строке кода, отсюда всякие болячки – уменьшение переносимости, сложности при модификации отдельных объектов и т.д.

Это еще один пример, когда не программист создает алгоритм, а язык «**программирует программиста**». Вы должны следовать этому принципу, так как это приносит некоторые удобства, но при этом все умалчивают о том, что это не гарантия преодоления нарастающей сложности программ (и более того, такой механизм сам может являться источником усложнения программы). И еще один момент – полиморфизм не обязательная черта ООП. Это значит, что если Вы напишете программу, использующую классы и объекты, но не использующую полиморфизм, то это все равно будет ООП. Кроме того, если Вам не хочется отказываться от полиморфизма, его можно имитировать в рамках модулей (юнитов) программ.

Что еще не так, как надо

ООП «вещь в себе», и оно не вписывается в ряд задач – например механизм подключения DLL. Да, функция или процедура возвращает значение, но какого типа? Это отследить невозможно (и RTTI Вам не помощник). ООП очень плохо подходит для парсинга структурированных текстов (например, программ) в сравнении с функциональным программированием. ООП не дружит с рекурсией (не то чтобы она там не возможна, просто не эффективна в сравнении с тем же функциональным программированием), которая позволяет упростить представление многих задач. Также нужно учитывать, что программа, составленная без ООП, как правило, быстрее, чем с ним. ООП официально не поддерживает концепцию ленивых вычислений (вообще это черта языков функционального программирования) – потенциальное увеличение производительности, упрощение решения ряда задач.

Подведем итоги

Данная статья вовсе не призывает отказываться от принципов ООП, задача дать более полное представление о некоторых аспектах ООП. Без понимания этих моментов программисты уверены, что ООП существенный шаг вперед (на самом деле существенный шаг вперед – обширные библиотеки, которые в свете новых веяний написаны в стиле ООП), хотя возможно, всего лишь шаг в сторону...

Возьмите параметризацию – не стоит возлагать на нее большие надежды, она существует уже давно в том или ином виде в функциональном программировании (как и весь полиморфизм во всех его проявлениях) и пока не принесла кардинальных изменений в деле создания программ.

Список использованной литературы

- . Тимоти Бадд. Объектно-ориентированное программирование в действии. – С.Петербург, Питер, перевод по лицензии издательства Addison Wesley, 1997
- . Г. Шилдт. Теория и практика C++. – С.Петербург, БХВ-Петербург, 2001

- . Раздел википедии http://ru.wikipedia.org/wiki/Объектно-ориентированное_программирование
- . Раздел абсурдопедии <http://absurdopedia.wikia.com/wiki/ООП>
- . Э. Хювенен, И. Сеппянен. Мир Лиспа. – М., перевод с финского изд.Мир, в 2-х томах, 1990
- . Ф. Брукс. Серебряной пули не существует. – М., Символ-Плюс, 1995, юбилейное издание первой книги 1987 года, перевод по лицензии издательства Addison Wesley



Многие хотят стать программистами или же улучшить свои познания в этом увлекательном занятии. Но, как только человек хочет начать что-то осваивать, перед ним встает вопрос: «...а с чего начать?». Собственно, в данной статье я попытаюсь ответить на этот распространенный вопрос.

Денис Пискунов

by spamer www.programmersforum.ru

В связи с тем, что в Интернете, да и не только в нем, довольно часто можно встретить людей, которые далеки от программирования, но желают постигнуть его и которые знают некоторые азы сего занятия, но не знают, что им делать дальше, я и пишу данную статью.

«В первый раз - в первый класс»

Для начала, человеку желающему научиться программировать, необходимо скачать/купить книгу по какому-то языку программирования. Я бы советовал для начала скачать электронную книгу, потому-что вдруг вам это занятие не понравится, а деньги на бумажную версию будут уже потрачены. Теперь давайте определимся с языком.

Многие, уже знающие люди, начинают спорить на счет-того, какой язык лучше выбрать начинающим для изучения. Но в нашем случае, я спорить ни с кем не буду, а просто посоветую для начала выбрать язык программирования Pascal. С чем связан такой выбор? Да все очень просто, начинающему будет намного проще понять логику работы программы (алгоритма) в Pascal'e, чем скажем, например в C++ или Assembler.

Так, с языком определились. Теперь вернемся к выбору книги. Как в интернете, так и на прилавках магазинов, лежит огромное количество разнообразной литературы по программированию. А какой-же учебник скачать/купить нам? Скажу сразу, ни в коем случае не покупайте книги типа «Программирование для чайников». Полезного из такой брошюры вы не возьмете ничего, а вот представление о программировании, после ее прочтения, у вас будет неправильное, а то и вообще пугающее. Собственно по Pascal'ю советую следующие материалы и учебники **[1-4]**. В данной литературе предоставляется хорошее и понятное описание структуры языка, команд, структур данных и т.д. Также присутствуют примеры решения задач и задания для самостоятельного выполнения.

Выбираем среду разработки [5-7]

С языком и обучающим материалом определились. И вот теперь осталось выбрать и установить среду для написания программы или как правильнее - «Интегрированную среду разработки» (IDE, Integrated development environment). Собственно, что представляет собой IDE? Попросту, это набор программных средств, при помощи которых программист разрабатывает программное обеспечение. Так как изучать мы будем чистый Pascal, то и приложения мы будем писать консольные, посему я советую следующую среду разработки - Turbo Pascal 7.0 и кросс-платформенный компилятор FreePascal. Конечно, можно выбрать и что-то современнее, например из IDE сред: TurboDelphi или Delphi 2010 или другие альтернативные. Но для новичка в программировании, я считаю - это будет неправильно, так как в IDE Delphi увидеть логику работы программы, структуру языка и т.д., будет тяжело.

После вот таких приготовлений – садимся читать выбранную книгу, и хочу заметить, не просто читать, а читать, запоминать и разбираться в написанном. Если будете просто читать книжку, то вы потратите свое время в пустую. Поэтому, после получения некоторого теоретического материала, обязательно необходимо все полученные знания закрепить на практике. А точнее – садимся и пишем свою первую программу «Hello World»... Справились с этой задачей, ставим себе новую и реализуем, не знаете, что себе задать – в учебниках есть практические задания. После прочтения книги и при имеющихся знаниях – сделайте свой собственный не большой проект, например «Телефонный справочник», «вариант игрушки» и т.д.

Далее, после того, как вы чувствуете, что довольно хорошо владеете изученным языком, а возможно и уже некой технологией, необходимо решить для себя: «...а нравится-ли мне данная отрасль программирования?». Для ответа на этот вопрос, **с помощью любого поисковика** ищем информацию о следующих, так сказать, видах программирования:

- . системное программирование
- . прикладное программирование
- . веб - программирование

После прочтения соответствующей информации и при уже имеющихся знаниях в программировании – вы должны выбрать дальнейший вид поля своей деятельности. Если вы определились, тогда начинайте углубленное изучение* выбранного направления.

Вот еще такой нюанс – не надо думать, что программирование заключается только в знании языков программирования. Если вы хотите стать действительно хорошим программистом, то вам обязательно нужно знать дополнительные технологии. Например, можно полностью посвятить себя изучению программирования графики, попутно ознакомиться с разнообразными графическими библиотеками, алгоритмами, связанными с графикой и т.д. Следовательно, для достижения каких-либо целей, вам всегда необходимо читать соответствующую литературу, а также запомнить один из основных моментов – научиться пользоваться поиском. Так как большинство вопросов уже обсуждалось в Интернете, то правильный запрос в поисковую систему даст вам интересующий ответ.

* помимо чтения литературы, также желательно общаться на соответствующих форумах. Например, выберите для себя один или два форума и, так сказать – «живите на них». На таких ресурсах Интернета можно довольно много узнать полезной информации, поделиться с кем-то такой-же информацией. Также всегда можно попросить помощи у профессионалов, например, что бы вам объяснили непонятный момент при изучении.

Заключение

И не бойтесь спрашивать знающих людей о том, что не знаете сами – ничего предосудительного в этом нет. В общем, не надо ждать доброго дяденьку, который придет, все Вам разжует и в рот положит, а начинайте сами достигать поставленной цели. Так что, дерзайте.

Ресурсы

- . Т.А. Павловская. Паскаль. Программирование на языке высокого уровня: практикум. – С.Петербург, Питер-Юг, 2003
- . Валерий Попов. Паскаль и Дельфи. Самоучитель. – С.Петербург, Питер, 2003
- . В.В. Фаронов. Turbo Pascal 7.0. Начальный курс: учебное пособие. – М., КноРус, 2007

- . А.Я. Архангельский. Язык Pascal и основы программирования в Delphi. – М., Бином-Пресс, 2008
- . Скачать компилятор FreePascal <http://www.freepascal.org/download.var>
- . Скачать IDE DELPHI <https://downloads.embarcadero.com/free/delphi>



Статья предназначена для представления краткого обзора нововведений в язык Дельфи (2010) по сравнению с Дельфи 7. Подопытным кроликом выступила IDE от Embarcadero® Delphi® 2010 Version 14.0.3513.24210...

by Utkin www.programmersforum.ru

Благодаря активным попыткам компании Embarcadero влиять на рынок продуктов разработки программ язык Дельфи быстро развивается, однако это развитие направлено в основном на попытки наверстать все нововведения в современных языках программирования (таких как C#). Никаких принципиально новых разработок и концепций не внедряется.

Директива Inline (появилась в Дельфи 2005)

По аналогии с C++, функции и процедуры теперь могут быть встраиваемыми со всеми вытекающими последствиями. А именно использование данной директивы не гарантирует вставку тела функции вместо ее вызова. Кроме того, существует еще целый ряд ограничений (согласно справочной системе). Эта директива бесполезна:

- . при позднем связывании (virtual, dynamic, message)
- . для функций и процедур имеющих ассемблерные вставки
- . для конструкторов и деструкторов, там она работать не будет (о чем Вам компилятор обязательно пожалуется)
- . для главного блока программы, секций инициализации и финализации модулей
- . если метод класса обращается к членам класса с более низкой видимостью, чем сам метод (например, если public метод обращается к private методу, то для такого метода inline-подстановка осуществляться не будет)
- . для процедур и функций, которые используются в выражениях проверки условия циклов while и repeat

Как сделать процедуру встроенной?

Procedure Add (var x: Integer; y: Integer); **Inline**;

Регулировать поведение inline можно следующими директивами:

- { \$INLINE ON } – по умолчанию включена, разрешает работу Inline;
- { \$INLINE AUTO } – будет осуществлена попытка встраивание кода функций и процедур, если:
 - а) они помечены как Inline;
 - б) если их размер будет менее 32-х байт.
- { \$INLINE OFF } – не разрешает работу Inline.

Следует отметить, что и в классическом C++ Inline никогда не была высокоэффективным механизмом, а учитывая ограничения, накладываемые компилятором Дельфи, ее использование под большим вопросом.

Перегрузка операторов (появилась в Delphi.Net)

В отличие от C++ перегрузка осуществляется немного по-другому. Для перегрузки операторов перегружается не символ оператора, а его символическое обозначение (сигнатура). Перегружать можно только для операций с экземплярами классов (см. таблицу):

Таблица. Операторы и их сигнатуры

Оператор	Сигнатура	Тип оператора
Неявное преобразование	Implicit(a : type) : resultType;	Приведение
Явное преобразование	Explicit(a: type) : resultType;	Приведение
-	Negative(a: type) : resultType;	Унарный
+	Positive(a: type): resultType;	Унарный
Inc	Inc(a: type) : resultType;	Унарный
Dec	Dec(a: type): resultType	Унарный
not	LogicalNot(a: type): resultType;	Унарный
not	BitwiseNot(a: type): resultType;	Унарный
Trunc	Trunc(a: type): resultType;	Унарный
Round	Round(a: type): resultType;	Унарный
=	Equal(a: type; b: type) : Boolean;	Сравнение
<>	NotEqual(a: type; b: type): Boolean;	Сравнение
>	GreaterThan(a: type; b: type) Boolean;	Сравнение
>=	GreaterThanOrEqual(a: type; b: type): resultType;	Сравнение
<	LessThan(a: type; b: type): resultType;	Сравнение
<=	LessThanOrEqual(a: type; b: type): resultType;	Сравнение
+	Add(a: type; b: type): resultType;	Бинарный
-	Subtract(a: type; b: type) : resultType;	Бинарный
*	Multiply(a: type; b: type) : resultType;	Бинарный
/	Divide(a: type; b: type) : resultType;	Бинарный
div	IntDivide(a: type; b: type): resultType;	Бинарный
Mod	Modulus(a: type; b: type): resultType;	Бинарный
shl	ShiftLeft(a: type; b: type): resultType;	Бинарный
shr	ShiftRight(a: type; b: type): resultType;	Бинарный
and	LogicalAnd(a: type; b: type): resultType;	Бинарный
Or	LogicalOr(a: type; b: type): resultType;	Бинарный
xor	LogicalXor(a: type; b: type): resultType;	Бинарный
and	BitwiseAnd(a: type; b: type): resultType;	Бинарный
Or	BitwiseOr(a: type; b: type): resultType;	Бинарный
xor	BitwiseXor(a: type; b: type): resultType;	Бинарный

Нужно обратить внимание – TRUNC, ROUND, INC, DEC считаются операторами, а не процедурами и функциями.

Вот пример использования:

```

TMyClass = class
  class operator Add(a, b: TMyClass): TMyClass;           // Перегрузка сложение для
                                                         // TMyClass
  class operator Subtract(a, b: TMyClass): TMyClass;     // Вычитание для TMyClass
  class operator Implicit(a: Integer): TMyClass;         // Неявное преобразование
                                                         // Integer в TMyClass
  class operator Implicit(a: TMyClass): Integer;         // Неявное преобразование
                                                         // TMyClass в Integer
  class operator Explicit(a: Double): TMyClass;          // Явное преобразование
                                                         // Double в TMyClass

end;

// Пример описание сигнатуры Add для перегрузки сложения для типа TMyClass
TMyClass.Add(a, b: TMyClass): TMyClass;
begin
  ...
end;
```

```
var
x, y: TMyClass; begin
  x := 12;           // Неявное преобразование из Integer
  y := x + x;        // Вызов TMyClass.Add(a, b: TMyClass): TMyClass
  b := b + 100;       // Вызов TMyClass.Add(b, TMyClass.Implicit(100))
end;
```

Подробнее о перегрузке операторов можно почитать здесь:

<http://www.realcoding.net/articles/delphinet-peregruzka-operatorov.html>

Помощники класса (Class Helpers)

Интересный механизм (ответ Дельфи на расширители классов в C#), призванный решить некоторые проблемы в обход наследования. Служит для дополнения класса новыми методами и свойствами:

```
...
type
  TMyClass = class
    procedure MyProc;
    function MyFunc: Integer;
  end;

procedure TMyClass.MyProc;
var
  X: Integer;
begin
  X := MyFunc;
end;

function TMyClass.MyFunc: Integer;
begin
  ...
end;

...
type
  TMyClassHelper = class helper for TMyClass
    procedure HelloWorld;
    function MyFunc: Integer;
  end;

procedure TMyClassHelper.HelloWorld;
begin
  WriteLn(Self.ClassName); // Здесь будет возвращен тип TMyClass, а не TMyClassHelper
end;

function TMyClassHelper.MyFunc: Integer;
begin
  ...
end;
```

```
...
var
  X: TMyClass;
Begin
  X := TMyClass.Create;
  X.MyProc;           // Вызов TMyClass.MyProc
  X.HelloWorld;       // Вызов TMyClassHelper.HelloWorld
  X.MyFunc;           // Вызов TMyClassHelper.MyFunc
end;
```

По сути, вариация на тему множественного наследования, но есть одна особенность – помощники класса позволяют дополнять любой существующий класс, без создания нового. Обратите внимание, что механизм помощника класса не использует явного упоминания Self при обращении к полям класса (помогаемого класса). То есть, «HelloWorld» имеет право обращаться к полям «TMyClass» (просто в нашем примере их нет). Аналогично «TMyClass» также имеет доступ к полям «TMyClassHelper» (в случае, если класс и его помощник объявлены в одном модуле).

С практической точки зрения удобный механизм, кроме одной детали – класс должен иметь только одного помощника, имеет ли он помощника проверить во время выполнения программы нельзя. Если в классе имеется несколько помощников (неважно в каком юните, лишь бы он видел класс), считаться рабочим будет только самый последний из объявленных. Это значит, что если «TMyClass» уже имел помощника, то будут доступны методы именно «TMyClassHelper», поскольку именно он объявлен последним. Таким образом, в лучшем случае, два и более помощника для одного класса вызовут ошибку компиляции, в худшем трудно отлавливаемую ошибку, жалобы программиста на багги в IDE и компиляторе и много потерянного времени. Чем сложнее проект, тем труднее будет установить причину ошибки.

С теоретической точки зрения механизм противоречивый – он увеличивает сцепляемость объектов и юнитов между собой. Перед использованием помощника, я должен проверить все модули, из которых доступен данный класс на предмет проверки существования такого помощника (представьте большой проект). Это нарушает принципы инкапсуляции – если раньше перед использованием класса нужно было знать только его интерфейс, то теперь для использования помощников, я должен отслеживать существование класса во всех модулях, где имеется доступ к данному классу. С этого момента механизм интерфейсов уже не играет особой роли, поскольку, обращаясь к объекту какого-либо класса, всегда можно обнаружить такой букет неизвестных методов, что интерфейсная часть класса становится даже вредной. Это нарушает принцип сокрытия данных – благодаря помощникам я могу менять работу своих и чужих классов и могу иметь доступ к его полям (в рамках юнита). Кстати, это ограничение на доступ к полям в рамках юнита также сводит на нет многие его плюсы – проще вписать новые методы в сам класс (или наследовать новый), чем создавать путаницу в классе, юните и проекте.

Записи стали объектами

И теперь имеют свои методы, свойства и конструкторы. Но, сокращенная запись по-прежнему разрешена (поэтому старые проекты должны переноситься и с сокращенной формой записей):

```
type
  TMyRecord = record
    type
```

```

    TInnerColorType = Integer;
var
    Red: Integer;
class var
    Blue: Integer;
procedure printRed();
constructor Create(val: Integer);
property RedProperty: TInnerColorType read Red write Red;
class property BlueProp: TInnerColorType read Blue write Blue;
end;

constructor TMyRecord.Create(val: Integer);
begin
    Red := val;
end;

procedure TMyRecord.printRed;
begin
    writeln('Red: ', Red);
end;

```

Абстрактные классы

Разрешены полностью абстрактные классы (раньше допускались только конкретные методы), содержащие объявления методов для дальнейшего их перекрытия в потомках...

```

type
    TAbstractClass = class abstract
        procedure SomeProcedure;
end;

```

strict private и strict protected

Строгое **private** - метод или свойство для класса и невидимое никому, вне класса даже в рамках текущего юнита.

Строгое **protected** - методы в этой секции будут видимы самому классу и его наследникам.

Таким образом, полное объявление выглядит теперь так...

```

type
    TKlass = class(TForm)
        strict protected
        protected
        strict private
        private
        public
        published
        automated
end;

```

Не наследуемые классы

По аналогии с C#, в Дельфи 2010 существуют классы от которых дальнейшее наследование невозможно:

```

type
    TAbstractClass = class sealed
        procedure SomeProcedure;
end;

```

Весьма сомнительное удовольствие для рядового разработчика. Никаких реальных преимуществ такой класс не дает. Точные причины создания такого механизма не известны и преимущества от его использования

очень призрачны – наследование не разрушает класса предка, поэтому и запечатывать их особой необходимости нет. Считается, что запечатанные классы работают быстрее обычных (сам не проверял) и они применяются для .NET платформы (сугубо в утилитарных целях – не все обертки над низкоуровневыми операциями, такими как WinApi, можно сделать наследуемыми).

Классовые константы (возникло в Delphi 8)

Классы могут иметь константы – сами классы, а не порождаемые от них объекты:

```
type
  TClassWithConstant = class
  public
    const SomeConst = 'This is a class constant';
  end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  ShowMessage(TClassWithConstant.SomeConst);
end;
```

Классовые типы (возникло в Delphi 8)

Класс может теперь содержать описание типа, которое можно использовать только в пределах данного класса:

```
type
  TClassWithClassType = class
  private
    type
      TRecordWithinAClass = record
        SomeField: string;
      end;
  public

    class var
      RecordWithinAClass: TRecordWithinAClass;
    end;
    ...
  procedure TForm1.FormCreate(Sender: TObject);
  begin
    TClassWithClassType.RecordWithinAClass.SomeField := 'This is a field of a class
type declaration';
    ShowMessage(TClassWithClassType.RecordWithinAClass.SomeField);
  end;
```

Еще одно сомнительное удовольствие. Описание типа это не конкретная структура, зачем прятать его описание в тело класса?

Классовые переменные (возникло в Delphi 8)

Класс может содержать переменные по аналогии с константами:

```
Type
X = class (TObject)
Public
Var
    Y: Integer;
End;
```

данном случае влияют на область видимости данных переменных). Применение статических полей в классе делает Дельфи все более ориентированным в сторону C# (и менее в сторону Паскаля).

Пожалуй, единственное, где это может пригодиться – это работа с RTTI. Вообще классы в Дельфи стали больше напоминать юниты – такие вот юниты внутри юнитов.

Обратите внимание, что переменные класса могут находиться в любой секции (секции в

Вложенные классы

```
type
    TOuterClass = class
    strict private
        MyField: Integer;
    public
        type
            TInnerClass = class
            public
                MyInnerField: Integer;
                procedure InnerProc;
            end;
            procedure OuterProc;
        end;

    procedure
    TOuterClass.TInnerClass.InnerProc;
begin
    ...
end;
```

Теперь классы можно объявлять внутри классов, цель избежать конфликта имен, локализовать все связанные классы между собой.

Классы все больше перенимают концепцию модулей. Понятно, что данное нововведение дань .Net, но реальной пользы от него опять же не очень много – раньше конфликта имен избегали префиксами A и F. Не могу сказать, что новый механизм дал программистам новые возможности.

Также как и попытка использовать вложенные классы для складывания всего в одну большую кучу наряду с помощниками классов сильно напоминают «лебедь, рак и щуку», растаскивающие Дельфи в разные стороны.

Финальные методы класса

В классах можно создавать виртуальные методы, которые перекрыть нельзя:

```
TAbstractClass = class abstract
    public
        procedure Bar; virtual;
    end;
TFinalMethodClass = class(TAbstractClass)
    public
        procedure Bar; override; final;
    end;
```

Переопределить «Var» уже больше нельзя.

Статические методы класса

У классов могут быть статические методы класса, то есть методы, которые можно вызывать от типа класса. Такие методы не имеют доступа к полям класса (также как и не могут получить Self на конкретный экземпляр данного класса):

```
type
  TMyClass = class
    strict private
      class var
        FX: Integer;
    strict protected

    // Note: accessors for class properties must be declared class static
    class function GetX: Integer; static;
    class procedure SetX(val: Integer); static;
  public
    class property X: Integer read GetX write SetX;
    class procedure StatProc(s: String); static;
end;

TMyClass.X := 17;
TMyClass.StatProc('Hello');
```

Здесь же представлен пример организации свойств классов. Их использование полностью аналогично использованию переменных и констант класса.

for-element-in-collection

```
for Element in ArrayExpr do Stmt;
for Element in StringExpr do Stmt;
for Element in SetExpr do Stmt;
for Element in CollectionExpr do Stmt;

// вот развернутый пример
var
  A: Array [1..6] of String;
  I: String;

...
for i in A do
  begin
    Memol.Lines.Add(i);
  end;
```

Теперь компилятор способен распознавать итерации в контейнерах.

Обратите внимание, что I имеет тип String - это не индекс массива, а конкретные значения, которые будут получаться из массива.

Кое-где конечно автоматизирует, но представьте, что мне нужно написать некий метод, в котором происходит копирование одного массива в другой. Использовать все равно придется стандартный цикл «for», либо писать еще один метод - добавление элемента в массив.

Динамическая инициализация массивов

Теперь массивы получили свои конструкторы:

Type

`TMas = Array of String;`

Var

`Mas: TMas;``Mas:= TMas.Create('Hello','World','!');`

массивах, но также о записях и конструкторах класса). Так вот конструктор должен называться «Create» только для массивов. Для всех остальных имя конструктора не обязательно должно быть «Create» (но желательно, особенно для классов).

Дженерики

Шаблоны, они и в C++ шаблоны. Считается что первые шаблоны возникли в C++, но вообще-то они пришли из функционального программирования и правильное их название – параметрический полиморфизм. Явление, когда компилятор сам вырабатывает соответствующие варианты кода на основании обобщенного алгоритма:

```
TList<T> = class
private
  FItems: array of T;
  FCount: Integer;
  procedure Grow(ACapacity: Integer);
  function GetItem(AIndex: Integer): T;
  procedure SetItem(AIndex: Integer; AValue: T);
public
  procedure Add(const AItem: T);
  procedure AddRange(const AItems: array of T);
  procedure RemoveAt(AIndex: Integer);
  procedure Clear;
  property Item[AIndex: Integer]: T read GetItem write SetItem; default;

  property Count: Integer read FCount;
end;
```

Вот пример списка содержащего произвольные (но однотипные элементы). Тип элементов определяется на момент объявления переменной:

`ilist: TList<Integer>;`

То есть, мы создали список целых чисел (а можно, к примеру, список строк). Дженерики удобно использовать применительно к алгоритмам контейнеров данных и комбинаторным алгоритмам. Конкретные реализации алгоритмов можно посмотреть в модуле `Generics.Collections`, где есть `TArray`, `TList`, `TStack`, `TQueue`, `TDictionary`, `TObjectList`, `TObjectQueue`, `TObjectStack`, `TObjectDictionary` и `TEnumerator`, способные работать с разными типами данных.

Также необходимо отметить особенность дженериков (и шаблонов в C++) – обобщенные алгоритмы экономят время программиста, сокращая только его код, но для каждого типа (для каждой комбинации типов) всегда генерируется новая версия алгоритма (поэтому размер скомпилированных программ увеличивается).

Заключение

Большинство механизмов представленных здесь:

- . обеспечивают совместимость с .NET
- . дань моде
- . попытка угнаться за Microsoft Visual Studio

Язык не содержит принципиальных отличий и мощных механизмов, которые действительно были бы востребованы именно программистами на языке Дельфи. Все нововведения навязаны, искусственны и не всегда соответствуют концепциям ООП. Большое количество противоречивых инструментов может только запутать программистов и в течение ближайших лет можно ожидать некоторого количества критических статей в адрес языка программирования Дельфи.

Ресурсы

- . Хроники «айтишника» <http://skiminog.livejournal.com/33610.html>
- . Общество разработчиков Embecadero <http://edn.embarcadero.com>
- . Углубленный материал по перегрузке операторов в Дельфи <http://www.realcoding.net/articles/delphinet-peregruzka-operatorov.html>
- . Онлайн-перевод англоязычных материалов статьи <http://www.translate.ru>

Комментарий автора.

В дополнение к вышесказанному, впечатления о среде оставляют желать лучшего: сплошные недоделки (да и в 2009-м не лучше), ждать следующую версию наверно не стоит. FrameWork идет в комплекте, ничего дополнительно устанавливать не надо. Несмотря на заявленные требования не ниже 1 гигабайта ОЗУ, у меня и при 512-ти с тормозами, но работает.

Также, были отмечены «баги»:

- 1) при вводе длинных комментариев курсор начинает скакать (проявляется не всегда). Ощущение, такое, что компонент во время отрисовки неправильно проставляет положение курсора.
- 2) получил ошибку **debug failed**. Местоположение ошибки отсутствует, проект запустить не дает. Причина ошибки неизвестна (собственно долго писал код, а потом исправлял ошибки, что и привело в итоге к данной ошибке). Решил перезапуском IDE.
- 3) после **ELSE**, при написании **begin**, **end** автоматически не проставляется...



В этой статье я подробно рассмотрю создание игры «Змейка» с использованием ООП (объектно-ориентированного программирования). Делать игру мы будем средствами GDI на Delphi 7 (хотя должна подойти и любая другая версия). Для полного осознания того, о чем говорится в статье желательно иметь хоть какой-нибудь опыт программирования, знать что такое Canvas и TBitmap, что такое классы и в чем заключаются их особенности...

Вадим Буренков vadim_burenkov@mail.ru

Вообще, при создании любой игры можно выделить три этапа:

1. Создание идеи, примерный план реализации. При желании разработчиком пишется «диздок» (дизайн-документ – это документ, который описывает то, что вы хотите создать).
2. Алгоритмизация. То есть, мы представляем план строения игры, какие типы данных будут использоваться. Обдумываются алгоритмы работы и взаимодействия элементов игры.
3. Все вышесказанное мы записываем в виде кода, то есть программируем.

В принципе, игра не будет использовать наследования, поэтому можно-было обойтись и без классов, но мне захотелось сделать так.

Краткий экскурс...

При создании любой игры нужно придерживаться определенного строения кода. Каждая игра имеет следующие секции:

- . инициализация (тут происходит настройка окна для рендера (вывода изображения), загрузка всех игровых ресурсов и уровня)
- . обработка (всех игровых объектов, их взаимодействие друг с другом. Эта секция в отличие от инициализации и деинициализации выполняется постоянно, причем скорость выполнения измеряется в миллисекундах и обычно* равна 10-100 мс)
- . деинициализация (выполняется при завершении работы приложения и служит для очищения занятой приложением памяти)

Сделаем основу для игры

Откройте Delphi, создайте новое приложение. Сразу хочу заметить, что именам всех переменных, окон, файлов, и т.п. надо давать осмысленные имена. Конечно, кто-то скажет что и так запомнит, что такое Form1 и Label2, но когда пишется проект не на 10 а на 10000 строк и используется не одна, а пятнадцать форм... Также не надо мелочиться на комментарии, поскольку смысл определенного кода через месяц простоя не поймет и тот, кто его написал, не говоря уже о том – что вам может понадобится сторонняя помощь.

Назовем форму MainForm, модуль GameMain, а проект SnakeProject. Поскольку эта игра простая и небольшая, то я буду использовать один модуль, но в остальных случаях рекомендуется использовать свой модуль для отдельной игровой части. Создадим два события формы: OnCreate() и OnDestroy() которые и будут инициализатором и деинициализатором игры. За обработку игры будет отвечать

* Примечание.

Обратите внимание! Компонент TTimer имеет низкую точность. Ставить на нем низкие значения (менее 30 мс) не имеет смысла.

компонент-таймер TTimer (поместите его на форму из вкладки System, назовем его MainTimer), а именно его событие – OnTimer().

Я не буду делать игру очень гибкой в плане игрового поля (его размеры нельзя будет менять. Поле будет 16x12 клеток, но о них будет написано ниже, основные параметры будут константами. Игровое же поле 640x480 по центру экрана без возможности перемещения (без синей рамки). Объявим константы и установим параметры окна:

```
Const // разрешение игрового поля
SCR_WIDTH=640;
SCR_HEIGHT=480;

// установка размеров окна
MainForm.Width      := SCR_WIDTH;
MainForm.Height     := SCR_HEIGHT;
// параметры формы
MainForm.BorderStyle:= bsNone;           // без рамки
MainForm.Position   := poDesktopCenter;  // по центру экрана
```

Теперь заставим работать таймер (изначально параметр таймера должен быть enable = false):

```
MainTimer.Interval  := 40;
MainTimer.Enabled   := true;
```

Создадим и добавим в событие OnTimer() две процедуры (пока пустые) – Main_Update() и Main_Draw(). Для чего это? Мы ведь можем

писать весь код напрямую в таймер. Поясню. Например, если мы захотим сделать в игре меню, то должны быть отдельные процедуры обработки/рисования для меню и игры и выполняться они будут в зависимости от того, где игрок находится:

```
Scene: (sMenu, sGame);
if Scene = sMenu then Menu_Draw;
if Scene = sGame then Main_Draw;
```

В этой статье я не буду делать меню, но все же...

Отрисовка

Итак, мы получили некую заготовку игры. Теперь надо правильно настроить рендер. Canvas отличается простотой, но его недостаток – медленная скорость. Если просто выводить спрайты (изображения) на форму, то будет видно – что они выводятся не сразу, а один за другим и появляется эффект мерцания изображения. Чтобы избежать этого – нужно выводить спрайты не на экран, а в буфер памяти. После того как финальное изображение будет построено, можно вывести его на форму. После вывода буфер можно очистить и он будет готов для построения следующего кадра. Теперь реализуем данную систему. Объявим переменную – буфер scr_Buffer как Tbitmap. Перед использованием, буфер нужно проинициализировать и установить размеры:

```
// инициализация буфера
scr_Buffer      := TBitmap.Create;
scr_Buffer.Width := SCR_WIDTH;
scr_buffer.Height:= SCR_HEIGHT;
```

И в событии `Main_Draw()` напомним код отрисовки буфера (все рисование должно происходить до этих строчек):

```
MainForm.Canvas.Draw(0, 0, scr_Buffer); // копируем содержимое буфера на экран
scr_Buffer.Canvas.Rectangle(0, 0, SCR_WIDTH, SCR_HEIGHT); // очищаем буфер
// буфер можно не очищать, если рисование происходит на всей форме
```

Тут же можем проверить, как наша заготовка справляется с рисованием. Загрузим фон для игры в переменную `BackImage`:

```
// создание
BackImage:= Tbitmap.Create;
BackImage.LoadFromFile('fon.bmp');
// отрисовка
scr_Buffer.Canvas.Draw(0, 0, BackImage);
// очищение
BackImage.Free;
```

Нажимаем клавишу `<F9>` и любуемся на результат (см. рисунок 1):



Рис. 1. Вывод фоновой картинки игры

Концепция игры. Долгосрочная стратегия

Итак, мы сделали «скелет» игры, на который будем насаживать игровой код. Обдумаем, что же будет представлять из себя игра? Есть змейка, она ползет по уровню, маневрирует между

препятствиями и ест все съедобное на ее пути. Но змея не должна есть стены и себя, иначе она погибнет. Из выше написанного подобия «диздока» можно выделить два класса, которые мы будем использовать в игре: змея (назовем ее TSnake) и остальные объекты (TGameObject). Объекты будут иметь переменную, которая определяет их тип: либо это еда, либо препятствие. Каждый объект имеет координату. Поле будет поделено на клетки, это позволит легко определять столкновения (если координаты двух объектов совпадают, то они столкнулись). Размер каждой клетки будет определяться константой `CAGE_SIZE = 40`. Зная координату клетки можно получить координату на экране, умножив ее на размер клетки. Это нам понадобится при рисовании.

К примеру, еда может находиться в клетке с координатой (5, 6). Это значит, что она будет рисоваться в координате (200, 240) на экране. Эти вспомогательные функции делают следующее:

```
Function G_X(x:integer):integer;  
begin  
  result:= x*CAGE_SIZE  
end;
```

```
Function G_Y(Y:integer):integer;  
begin  
  result:= Y*CAGE_SIZE  
end;
```

Программируем поведение змейки

Обдумаем, как же будет работать змейка. Каждая деталь змейки – это отдельный элемент со своими координатами. Когда змейка движется – первый элемент (голова) занимает новую позицию, а второй элемент занимает позицию предыдущего до перемещения (там, где раньше была голова), третий позицию второго и т.д. Это самый логичный вариант, который приходит в голову. Но тогда надо сохранять координаты, где было тело змейки. Для упрощения кода есть более легкий способ – двигать змейку с хвоста. Перейдем к созданию класса змейки:

```
type  
TSnake=class  
  Sprite:TBitmap;           // спрайт составной части змейки  
  len:integer;              // длина змейки (из какого кол-ва частей состоит змейка)  
  BodyPos: array of vect2D; // динамический (!) массив с координатами частей змейки  
  Speed:vect2D;             // скорость змейки (сколько клеток она проходит за ход)  
  MoveTo:(left,right,up,down); // показатель направления движения  
  
  // создание змейки со спрайтом, позицией и длиной  
  Constructor Create(spr_name:string;pos:vect2D;size:integer);  
  Procedure SetSize(size:integer); // изменение размера  
  Procedure Drive;                 // управление змейкой  
  Procedure Update;                // один ход змейки  
  Procedure Draw;                  // отрисовка змейки  
  Destructor Free;                 // удаление змейки  
end;
```

Как и в любой игре, в этой нельзя обойтись без математики. Тип «Vect2D» облегчает нам задачу хранения координат и взаимодействия с ним:

```
type
vect2D = record
  X,Y: integer;
end;

Function v2(x,y:integer):vect2D;           // создание
begin
  result.X:= X;
  result.Y:= Y
end;

Function v2ident(v1,v2:vect2D):boolean;    // проверка одинаковых координат
begin
  result:= false;
  if (v1.x=v2.x) and (v1.y=v2.y) then result:=true
end;

Function v2add(v1,v2:vect2D):vect2D;      // сложение координат
begin
  result.X:= v1.x+v2.x;
  result.Y:= v1.y+v2.y
end;
```

И вот код всех процедур класса:

```
// мы создаем змейку
Constructor TSnake.Create(spr_name:string;pos:vect2D;size:integer);
Begin
  Sprite:= TBitmap.create;    // загружаем спрайт
  Sprite.LoadFromFile(spr_name);
  Sprite.Transparent:= true;
  SetSize(size);              // установка длины
  BodyPos[0]:= pos;           // установка положения
  MoveTo:= left;              // изначальное направление
end;

// установка длины массива частей змейки
Procedure TSnake.SetSize(size:integer);
begin
  len:=size;
  SetLength(BodyPos, len)
end;

// вот тут самое сложное и интересное
// относительно MoveTo устанавливается скорость змейки
// и происходит сдвиг змейки в сторону движения
procedure TSnake.Update;
var i: integer;
begin
```

```
if MoveTo=Up      then Speed:=v2( 0, -1);
if MoveTo=Down    then Speed:=v2( 0,  1);
if MoveTo=Left    then Speed:=v2(-1,  0);
if MoveTo=right   then Speed:=v2( 1,  0);

// двигаем хвост (downto означает, что мы идем от большего к меньшему)
for i:= len-1 downto 1 do
  BodyPos[i]:= BodyPos[i-1];
// двигаем первый элемент (голову) прибавляя скорость
BodyPos[0]:= v2add(BodyPos[0], Speed);

// проверка столкновения головы змейки с хвостом
for i:=1 to len-1 do if v2Ident(BodyPos[0],BodyPos[i]) then TryAgain;
end;
```

TryAgain() – процедура вызываемая при проигрыше. К ней вернемся позже. В следующей процедуре мы управляем змейкой, задавая ее направление движения через MoveTo(). Это отдельная от Tsnake.Update процедура, поскольку Update() будет вызываться реже таймера, чтобы контролировать скорость движения змейки. При этом, ловить нажатые клавиши управления надо постоянно. Определение нажатых клавиш происходит через функцию Key_Press():

```
function Key_Press(key: byte): boolean;
var
  keys: TKeyboardState;
begin
  result:=false;
  GetKeyboardState(keys);
  If (keys[key] = 128) or (keys[key] = 129) then result:= true
end;
```

Коды клавиш можно определить специальной программой [2]. Те, что нам понадобятся, я занес в константы:

```
KEY_UP    = 38;
KEY_DOWN  = 40;
KEY_LEFT  = 37;
KEY_RIGHT = 39;
KEY_ESC   = 27;
```

Поскольку змейка может только поворачивать влево и вправо, а не может изменять направления движения мгновенно на 180 градусов, введены проверки типа **v2Ident(v2(BodyPos[0].x,BodyPos[0].y-1),BodyPos[1])=false**, которые не позволяют ей этого сделать:

```
procedure TSnake.Drive;
begin
  if Key_Press(KEY_UP) then
    if v2Ident(v2(BodyPos[0].x,BodyPos[0].y-1),BodyPos[1])=false then MoveTo:= Up;
  if Key_Press(KEY_DOWN) then
```



```

if v2Ident(v2(BodyPos[0].x,BodyPos[0].y+1),BodyPos[1])=false then MoveTo:= Down;
if Key_Press(KEY_LEFT) then
  if v2Ident(v2(BodyPos[0].x-1,BodyPos[0].y),BodyPos[1])=false then MoveTo:= Left;
if Key_Press(KEY_RIGHT) then
  if v2Ident(v2(BodyPos[0].x+1,BodyPos[0].y),BodyPos[1])=false then MoveTo:= Right;
end;

```

И две последние процедуры. Отрисовка отобразит все части змейки по координатам (голова отрисовывается два раза для наглядности):

```

procedure TSnake.Draw;
var i: integer;
begin
  for i:=0 to len-1 do if not v2Ident(BodyPos[i],v2(0,0)) then begin
    // не отрисовываются части с нулевыми координатами,
    // так как их имеют новые части змейки до движения
    if i=0 then scr_Buffer.Canvas.draw(G_X(BodyPos[i].x)+
                                         Speed.x*5,G_Y(BodyPos[i].y)+Speed.y*5,sprite);
    scr_Buffer.Canvas.draw(G_X(BodyPos[i].x),G_Y(BodyPos[i].y),sprite);
  end
end;

// при удалении змейки очищается спрайт
destructor TSnake.Free;
begin
  Sprite.Free;
end;

```

Процедура TryAgain() будет вызываться при проигрыше. Напишем ее, но пока оставим пустой. Змейка написана и может работать (см. рисунок 2). О том как заставить работать класс змейки в игре написано далее...

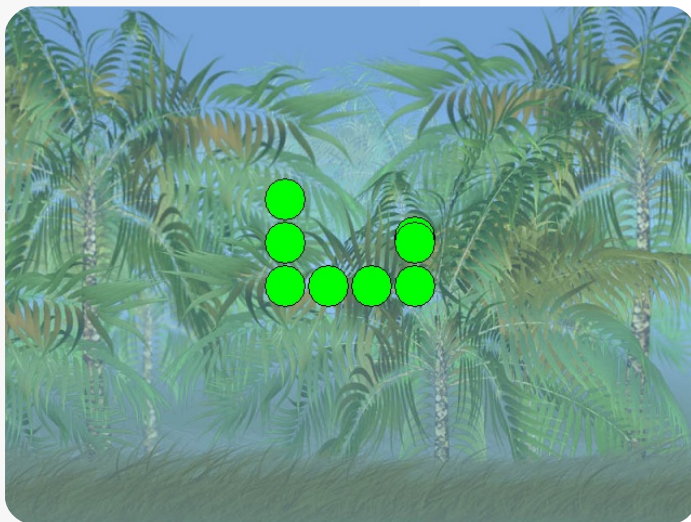


Рис. 2. Отображение нашей змейки

Объекты в игре. Не дадим змейке «помереть с голоду»

Теперь необходимо добавить объекты. Они будут двух типов:

```
Type
TObjType=(oWall,oFood);           // стены и еда

type
TGameObject = class
  ObjType:TObjType;               // тип объекта
  Sprite:TBitmap;                 // спрайт
  Pos:vect2D;                     // положение

  Constructor Create(spr_name:string;oType:TObjType;p:vect2D); // создание
  procedure Update(Snake:TSnake);  // обновление (проверка столкновения со змеей)
  procedure Draw;                 // отрисовка
  destructor Free;                // удаление
end;
```

Тут ничего сложного нет. Многие процедуры аналогичны змейке:

```
Constructor TGameObject.Create(spr_name:string;oType:TObjType;p:vect2D);
begin
  Sprite:=TBitmap.create;
  Sprite.LoadFromFile(spr_name);
  Sprite.Transparent:=true;
  pos:= p;
  ObjType:= oType
end;

Procedure TGameObject.Update(Snake:TSnake);
begin
  if v2Ident(Snake.BodyPos[0],pos) then begin // если змея столкнулась с объектом
    if ObjType = oWall then TryAgain;
    if ObjType = oFood then begin Snake.SetSize(Snake.len+1);
      pos:= RandomFieldPos
    end;
    // увеличиваем размер змеи,
    // перемещаем еду в другое место
    // что такое RandomFieldPos см. ниже
  end;
end;

Procedure TGameObject.Draw;
begin
  scr_Buffer.Canvas.Draw(G_X(pos.x),G_Y(pos.y),sprite)
end;

Destructor TGameObject.Free;
begin
  Sprite.Free
end;
```

Используем классы в игре

Итак, компоненты игры написаны. Теперь их надо создать, добавить обработку и отрисовку в `Main_Update()` и `Main_Draw()`. Вот полный код инициализации змейки и стен. Объявим следующие переменные:

```
MySnake:TSnake;           // змейка игрока
Wall: array of TGameObject; // все стены игры
WallNum:integer;          // их кол-во
Food:TGameObject;         // еда для змейки
```

И добавим в инициализацию следующий код:

```
// создаем змейку MySnake
MySnake:= TSnake.Create('snake.bmp',v2(7,7),2);
// теперь создам еду. А на ужин у нас аппетитненький пингвинчик
Food:=TGameObject.Create('food.bmp',oFood,RandomFieldPos);
```

Чтобы не вводить координату каждой стенки уровня, так как загрузчика уровней нет, я просчитал создание всех стен в операторе `for`:

```
WallNum:= 54;
SetLength(Wall,WallNum);
// верхняя
for i:=0 to 15 do Wall[i] := TGameObject.Create('wall.bmp', oWall, v2(i,0));
// нижняя
for i:=16 to 31 do Wall[i]:= TGameObject.Create('wall.bmp', oWall, v2(i-16,11));
// левая
for i:=32 to 42 do Wall[i]:= TGameObject.Create('wall.bmp', oWall, v2(0,i-32));
// правая
for i:=43 to 53 do Wall[i]:= TGameObject.Create('wall.bmp', oWall, v2(15,i-43));
```

Выше я писал о функции `RandomFieldPos()`, которая возвращает случайную координату на поле, где нет стен. В `OnCreate()` надо поставить `randomize()` для инициализации генератора случайных чисел:

```
function RandomFieldPos:vect2D;
begin
    // я просчитал допустимые значения
    result:=v2(random(13)+1,random(9))
    // по X от 1 до 14, по Y от 1 до 10
    // тут ничего сложного нет
end;
```

Собираем запчасти

Теперь надо добавить нашу змейку, стены и пингвинчика в обработку и отрисовку. Поскольку скорость движения змейки надо ограничить, мы заводим переменную-счетчик `WaitTime`. Она считает до 5 и выполняет процедуру движения и сбрасывает себя на 0. В итоге, `MySnake.Update()` срабатывает в 5 раз реже таймера. Ошибкой многих начинающих разработчиков является использование большого количества таймеров, что сильно усложняет

код. Чтобы из игры можно было нормально выйти, сделаем условие нажатия на клавишу <ESCAPE>:

```
procedure Main_Update;
var i: integer;
begin
  MySnake.Drive;
  for i:=0 to WallNum-1 do wall[i].Update(MySnake);
  Food.Update(MySnake);
  if WaitTime>5 then begin MySnake.Update; WaitTime:= 0; end else inc(WaitTime);
  if Key_Press(KEY_ESC) then Application.Terminate
end;

procedure Main_Draw;
var i: integer;
begin
  scr_Buffer.Canvas.Draw(0, 0, BackImage);

  MySnake.Draw;
  for i:=0 to WallNum-1 do Wall[i].Draw;
  Food.Draw;

  MainForm.Canvas.Draw(0, 0, scr_Buffer);
  scr_Buffer.Canvas.Rectangle(0, 0, SCR_WIDTH, SCR_HEIGHT);
end;
```

Напишем «правильное очищение» всех ресурсов игры:

```
procedure TMainForm.FormDestroy(Sender: TObject);
var i: integer;
begin
  scr_Buffer.Free;
  BackImage.Free;
  MySnake.Free;
  for i:=0 to wallnum-1 do wall[i].Free
end;
```

После чего остается процедура «проигрыша» TryAgain(). В ней сбрасывается длина змейки, а сама она перемещается на стартовую позицию:

```
procedure TryAgain; // сброс всех координат
begin
  MySnake.SetSize(0);
  MySnake.SetSize(2);

  MySnake.BodyPos[0]:= v2(7,7);
  MySnake.MoveTo:= left;

  Food.Pos:=RandomFieldPos
end;
```


Размер устанавливается два раза: первый для того, чтобы сбросить координаты частей змейки на нулевые значения, второй для установки начальной длины. И вот результат (см. рисунок 3):

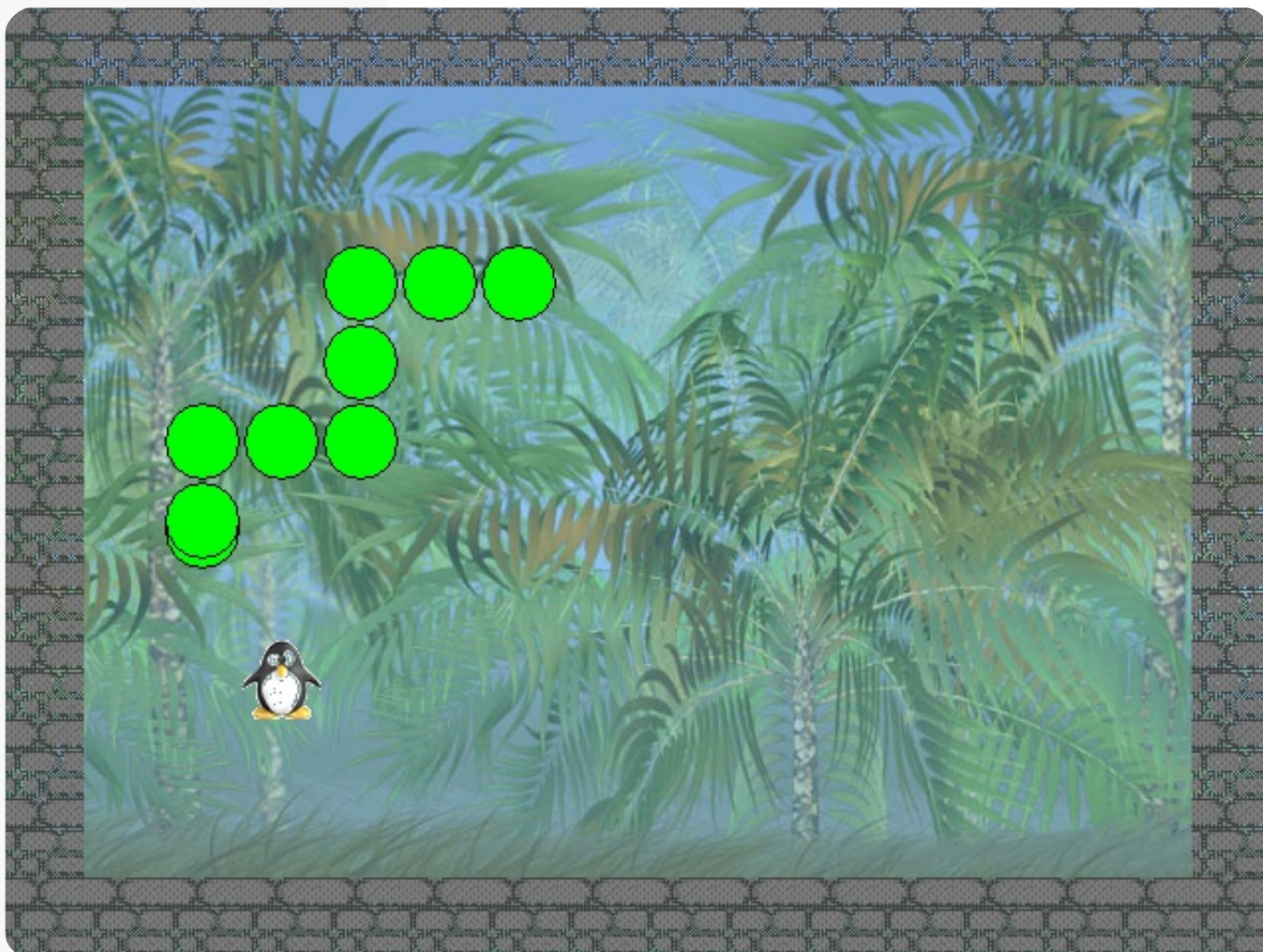


Рис. 3. Результат наших трудов. Любуемся игрой

Заключение

Теперь можно запустить игру и наслаждаться результатом. Посмотрев на финальный код можно увидеть, что разработка простой игры не требует каких-либо особых усилий и знаний. Мной эта игра для статьи была написана за 2-3 часа. Весь исходный код проекта, а также текстуры, используемые для создания игры можно найти на <http://www.programmersforum.ru> в разделе «Журнал клуба программистов. Второй выпуск».



Здравствуйте, уважаемые читатели. Сегодня мы с вами продолжим наши занятия по изучению методов работы с графикой в среде Дельфи для начинающих и добавим интерактивности в нашу «космическую стрелялку».

Продолжение. Начало цикла смотрите в [первом выпуске](#) журнала...

Владимир Дегтярь
by DeKot degvv@mail.ru

Одним из основных методов получения эффекта движущегося объекта является принцип уничтожения объекта на текущем месте и вывод его на новом с заданным приращением координат dX и dY . Однако прямолинейное применение такого метода на практике приводит к неприятному эффекту «мерцания» объекта во время движения. Что-же приводит к этому?

Создание движущихся объектов (еще немного теории). Урок 3

Причина этого в том, что при проведении этих операций непосредственно на канве объекта, где происходит движение, мы получаем двойную перерисовку участка канвы. Такая двойная смена цвета пикселей и является причиной. Сказанное наглядно видно на рисунке (см. рис.1), во время каждого такта движения происходит два процесса рисования: сначала старое изображение «затирается» фоном, а затем рисуется новое изображение движущегося объекта на новом месте...

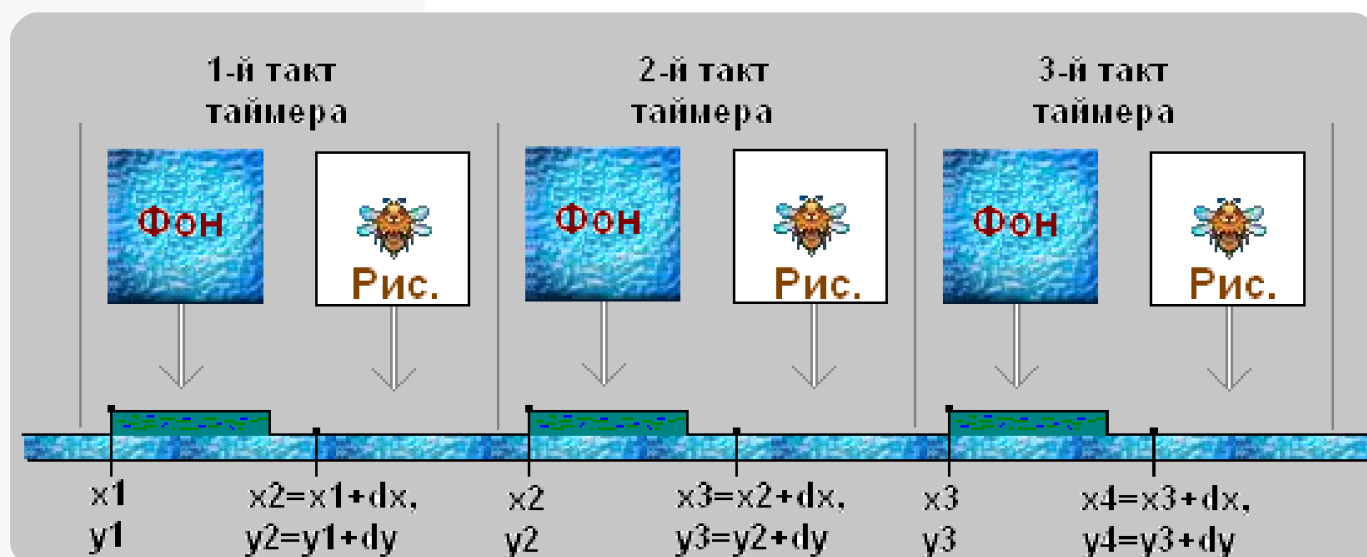


Рис. 1. Демонстрация эффекта «мерцания»

Избежать данной проблемы можно, объединив обе операции «стирания» изображения и вывода нового в одну. Суть этого состоит в том, чтобы на канве объекта, невидимого на экране, вывести стирающий участок фона, затем вывести изображение движущегося объекта на новом месте и уже после этих операций вывести полученную комбинацию изображения на видимую часть канвы экрана.

Для этого создаем дополнительный объект со свойством канвы (лучше всего подходит дополнительный буфер типа TBitmap), на котором и проводим предварительные операции. Размер такого дополнительного буфера определяется в зависимости от величины смещения движущегося объекта.

Возможны два варианта: когда старое и новое положение объекта имеют общую область (перекрывают друг друга) и не перекрывают (см. рис.2):

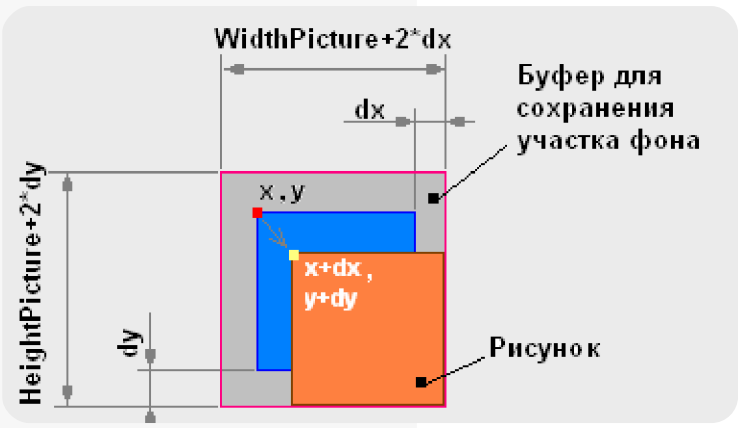


Рис. 2. Перекрытие объектов

смещением dX и dY (опять же по абсолютной величине) относительно нулевых координат буфера. После вывода изображения всего буфера на экран изображение объекта исчезнет и появится в новом положении.

Во втором случае, когда старое и новое изображения не имеют общей области также можно применить указанный метод. Но при этом, судя по рисунку 3, значительно увеличивается размер дополнительного буфера.

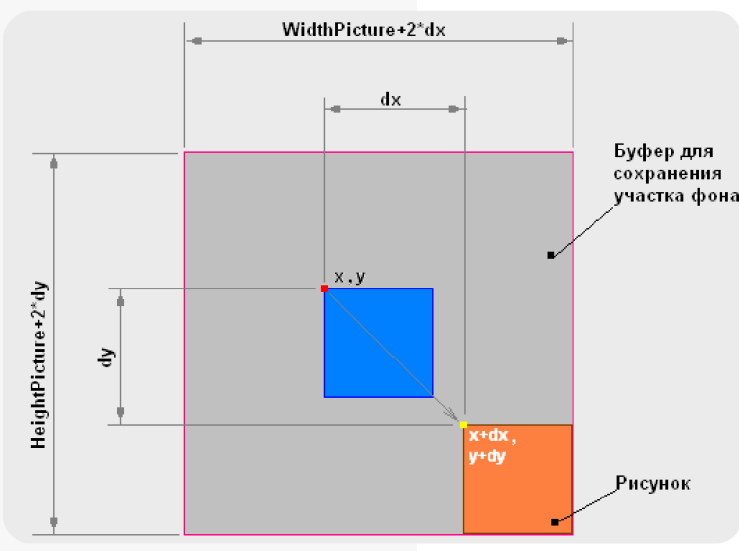
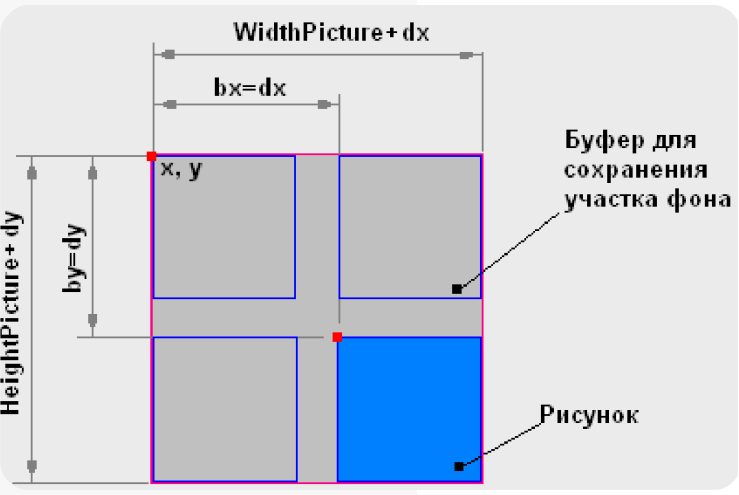


Рис. 3. Случай отсутствия перекрытия изображений



```
if dx>0 и dy>0, то bx=dx и by=dy;  
if dx>0 и dy<=0, то bx=dx и by=0;  
if dx<=0 и dy>0, то bx=0 и by=dy;  
if dx<=0 и dy<=0,то bx=0 и by=0;
```

Значения bx и by присваиваются, естественно по абсолютным величинам dx и dy. Рассмотрим код программы для первого случая (см. листинг 1):

Рис. 4. Принцип уменьшения буфера

ЛИСТИНГ -1

```
Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
  procedure FormActivate(Sender: TObject);
  procedure FormPaint(Sender: TObject);
  procedure FormKeyDown(Sender: TObject; var Key: Word;
    Shift: TShiftState);
  private
  end;

var
  Form1: TForm1;
  { битовые образы рисунка, фона и дополнит. буфера, соответственно }
  Pic, Fon, Buf : TBitmap;
  // области вывода фона
  RectFon, RectBuf : TRect;
  // текущие координаты
  x,y: integer;
  // приращения координат
  dx, dy: integer;
  // размеры дополнительного рисунка и буфера
  W,H,WB,HB: integer;

implementation
{$R *.dfm}

procedure TForm1.FormActivate(Sender: TObject);
begin // инициализация TBitmap и TRect
  // загружаем рисунок фона
  Fon:= TBitmap.Create;
  Fon.LoadFromFile('fon.bmp');
  { загружаем рисунок объекта, который будет двигаться }
  Pic:= TBitmap.Create;
  Pic.LoadFromFile('picture.bmp');
  // задаем прозрачность объекту
  Pic.Transparent:= true;
  Pic.TransparentColor:= Pic.Canvas.Pixels[1,1];
  { определяем и устанавливаем размеры дополнит. буфера }
  W:= Pic.Width;
  H:= Pic.Height;
  Buf      := TBitmap.Create;
  Buf.Width := W + 2*abs(dx);
  Buf.Height:= H + 2*abs(dy);
  WB:= Buf.Width;
```

```
HB:= Buf.Height;
// назначаем соответствие палитр цветов
Buf.Palette:= Fon.Palette;
Buf.Canvas.CopyMode:= cmSrcCopy;
{ определяем область в дополнит. Буфере для загрузки участка фона }
RectBuf:= Bounds(0,0,WB,HB);
// инициализация координат
x:=300; y:=150;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin // выводим на экран фон и объект
  Form1.Canvas.Draw(0,0,Fon);
  Form1.Canvas.Draw(x,y,Pic);
end;

{ перемещение объекта на один шаг происходит после каждого нажатия соответствующей клавиши }
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
begin
  case Key of
    37: begin dx:= -3; dy:= 0; end;
    38: begin dx:= 0; dy:= -3; end;
    39: begin dx:= 3; dy:= 0; end;
    40: begin dx:= 0; dy:= 3; end;
  end;
  { определяем область фона, которую надо запомнить }
  RectFon:= Bounds(x+dx,y+dy,WB,HB);
  // запомним фон в буфере
  Buf.Canvas.CopyRect(RectBuf,Fon.Canvas,RectFon);
  // наложим в буфере на фон рисунок с прозрачным фоном
  Buf.Canvas.Draw(abs(dx),abs(dy),Pic);
  // выводим на форму в новой позиции
  x:=x+dx; y:=y+dy;
  Form1.Canvas.Draw(x,y,Buf)
end;

end.
```

А это, для второго случая (см. листинг 2):

```
unit Unit1;
interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms, Dialogs;

type
  TForm1 = class(TForm)
```

ЛИСТИНГ - 2

```
procedure FormActivate(Sender: TObject);
procedure FormPaint(Sender: TObject);
procedure FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
private
end;

var
  Form1: TForm1;
  { битовые образы рисунка, фона и дополнит.буфера, соответственно }
  Pic, Fon, Buf : TBitmap;
  // области вывода фона
  RectFon ,RectBuf : TRect;
  // текущие координаты
  x,y: integer;
  // приращения координат
  dx, dy: integer;
  { смещение изображения объекта
    в дополнительном буфере }
  bx, by: integer;
  // размеры дополнит.рисунка и буфера
  W,H,WB,HB: integer;

implementation
{$R *.dfm}

procedure TForm1.FormActivate(Sender: TObject);
begin // инициализация TBitmap и TRect
  // загружаем рисунок фона
  Fon:= TBitmap.Create;
  Fon.LoadFromFile('fon.bmp');
  { загружаем рисунок объекта, который будет двигаться }
  Pic:= TBitmap.Create;
  Pic.LoadFromFile('picture.bmp');
  // задаем прозрачность объекту
  Pic.Transparent:= true;
  Pic.TransparentColor:= Pic.Canvas.Pixels[1,1];
  { определяем и устанавливаем размеры дополнит. буфера }
  W:= Pic.Width;
  H:= Pic.Height;
  Buf:= TBitmap.Create;
  Buf.Width:= W + abs(dx);
  Buf.Height:= H + abs(dy);
  WB:= Buf.Width;
  HB:= Buf.Height;
  // назначаем соответствие палитр цветов
  Buf.Palette:= Fon.Palette;
  Buf.Canvas.CopyMode:= cmSrcCopy;
  { определяем область в дополнительном буфере для загрузки участка фона }
  RectBuf:= Bounds(0,0,WB,HB);
```



```
// инициализация координат
x:=300; y:=150;
end;

procedure TForm1.FormPaint(Sender: TObject);
begin // выводим на экран фон и объект
  Form1.Canvas.Draw(0,0,Fon);
  Form1.Canvas.Draw(x,y,Pic);
end;

{ перемещение объекта на один шаг происходит после каждого нажатия соответствующей
клавиши }
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
begin
  case Key of
    37: begin dx:= -3; dy:= 0; end;
    38: begin dx:= 0; dy:= -3; end;
    39: begin dx:= 3; dy:= 0; end;
    40: begin dx:= 0; dy:= 3; end;
  end;
  if (dx>0) and (dy>0) then begin bx:=abs(dx); by:=abs(dy); end;
  if (dx>0) and (dy<=0) then begin bx:=abs(dx); by:=0; end;
  if (dx<=0) and (dy>0) then begin bx:=0; by:=abs(dy); end;
  if (dx<=0) and (dy<=0) then begin bx:=0; by:=0; end;

  { определяем область фона, которую надо запомнить }
  RectFon:= Bounds(x+dx,y+dy,WB,HB);
  // запомним фон в буфере
  Buf.Canvas.CopyRect(RectBuf,Fon.Canvas,RectFon);
  // наложим в буфере на фон рисунок с прозрачным фоном
  Buf.Canvas.Draw(bx,by,Pic);
  // выводим на форму в новой позиции
  x:=x+dx; y:=y+dy;
  Form1.Canvas.Draw(x,y,Buf)
end;

end.
```

В принципе обе эти программы можно объединить, введя процедуру проверки на совпадение старой и новой областей изображения. Но это уже другая тема.

В данных примерах подразумевалось, что движущийся объект состоит всего из одного изображения 'picture.bmp'. Но, на практике, обычно таких изображений несколько. Наиболее часто, изображения хранятся в файле в виде набора спрайтов. Вот здесь, есть один небольшой подводный камень. Обычно, выбор необходимого спрайта производится процедурой CopyRect() из буфера типа TBitMap, куда предварительно загружены изображения всех спрайтов. Но, при таком методе вывода изображения в дополнительный буфер с участком фона нельзя присвоить изображению свойство прозрачности.

Для обхода этого препятствия следует ввести еще один дополнительный буфер типа TBitMap, куда методом CopyRect() загрузить изображение нужного спрайта, а затем уже, присвоив

свойство прозрачности, методом Canvas.Draw() вывести изображение на канву дополнительного буфера с участком фона.

Следует отметить, что для создания «эффекта» движения можно оставить изображение графического объекта неподвижным, а двигать сам фон. При этом размер фона должен быть больше размера окна формы. Также можно применять одновременно или поочередно сдвиг объекта и сдвиг фона.

Проект с движением звездолета. Урок-4

Аналогично «Уроку 2» создадим новый проект и сохраним его под именем <Lesson2>. Параметры формы Form1 также установим аналогично «Уроку 2», изменив только заголовок Form1 на «Урок по графике № 2».

Для получения движения объекта применим метод, когда сам объект (звездолет) неподвижен на форме, а «двигается» сам фон. Фон создадим в два раза больше по высоте размера окна формы. Для этого используем файлы 'star1.bmp' и 'star2.bmp'. Нам также понадобится еще один дополнительный буфер BufFonDop. Кроме этого, введем еще переменные – координаты вывода на форму фона и звездолета, а также приращения к координатам (см. листинг 3):

ЛИСТИНГ - 3

```
var
Form1: TForm1;
BufFon, BufFonDop, BufSpr, BufPic, Buffer: TBitmap;

xs1, ys1: integer; // координаты звездолета 'ship1'
xf, yf: integer;   // координаты вывода общего буфера на форму
dyf: integer = 2;  // приращение изменения координаты yf по вертикали
dxs1, dys1: integer; // приращение координат звездолета по горизонтали и вертикали
ns: byte;          // номер спрайта

{ Здесь-же вводим начальные данные для переменных }

xs1 := 250; ys1 := 1006; // начальные значения переменных-
dxs1:= 4;   dys1:= 2;
xf  := 0;   yf:= -540; dyf:= 2;
...

{ Инициализацию буферов проведем в процедуре OnCreate() формы }

procedure TForm1.FormCreate(Sender: TObject);
begin
  BufFon  := TBitmap.Create;
  BufFonDop:= TBitmap.Create;
  BufSpr  := TBitmap.Create;
  BufPic  := TBitmap.Create;
  Buffer   := TBitmap.Create;

  BufFonDop.LoadFromFile('data/star1.bmp'); // загрузка 1-го рисунка фона из файла
  BufSpr.LoadFromFile('data/ship1.bmp');   // загрузка спрайтов из файла
  BufFon.Width:= BufFonDop.Width;         // размеры общего буфера фона
```

```

BufFon.Height:= (BufFonDop.Height) * 2;    // по высоте = двум Height одного
                                              // рисунка фона
BufFon.Canvas.Draw(0,0,BufFonDop);          // заносим 1-й рисунок фона
BufFonDop.LoadFromFile('data/star2.bmp');    // загрузка 2-го рисунка фона из файла
BufFon.Canvas.Draw(0,540,BufFonDop);        // заносим 2-й рисунок фона

Buffer.Width := BufFon.Width;               // размеры общего буфера = размерам буфера фона
Buffer.Height:= BufFon.Height;

BufPic.Width := round((BufSpr.Width) / 2); // размеры буфера рисунка одного спрайта
BufPic.Height:= BufSpr.Height;

```

В процедуре DrawShip1(i) добавлено условие прозрачности фона рисунка звездолета (см. листинг 4):

```

// загрузка одного спрайта в буфер рисунка
procedure DrawShip1 ( i: byte);
begin
  BufPic.Canvas.CopyRect(bounds(0, 0, BufPic.Width, BufPic.Height),
                          BufSpr.Canvas,bounds( i * 66, 0, BufPic.Width,
                          BufPic.Height));
  BufPic.transparent    := true;
  BufPic.transparentcolor:= BufPic.canvas.pixels[1, 1]
end;

```

ЛИСТИНГ-4

«Эффект» движения организован в обработчике таймера. В каждом такте таймера 50 мсек:

- . смещаем координату вывода общего буфера yf на dyf вниз (+ 2)
- . координату yS1 вывода звездолета смещаем на dyS1 вверх (- 2). В этом случае изображение звездолета будет неизменным в координатах формы, а двигаться будет только фон
- . в общий буфер (Buffer) выводим фон (при этом «уничтожается» изображение звездолета на старом месте)
- . выводим в общий буфер на фон новое изображение звездолета со смещением dyS1
- . выводим общий буфер на форму

В процедуре DrawShip1(ns) через переменную ns, принимающую попеременно значения «0» или «1», выводятся поочередно спрайты изображения звездолета (см. листинг 5):

```

// организация эффекта движения
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  ns:= ns + 1; if ns > 1 then ns:= 0;           // смена спрайтов
  DrawShip1(ns);

  yf:= yf + dyf; if yf > 0 then begin yf:= -540; yS1:= 1006; end; // приращение yf
  yS1:= yS1 - dyS1; // приращение звездолета обратно приращению фона
  Buffer.Canvas.Draw(0,0,BufFon); // выводим фон в общий буфер
  Buffer.Canvas.Draw(xS1,yS1,BufPic); // выводим рисунок спрайта поверх фона
  Form1.Canvas.Draw(xf,yf,Buffer); // выводим все на форму
end;

```

ЛИСТИНГ-5

```
// движение звездолета по горизонтали в пределах окна формы
// организовано в обработчике нажатия клавиши OnKeyDown()
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);
begin
    case Key of
    37: begin                                // клавиша "стрелка" - влево
        if xS1 <= 0 then EXIT;                // если звездолет у левого края формы
                                            // xS1 не изменяется
        xS1:= xS1 - dxS1;                    // движение звездолета влево
    end;
    39: begin                                // клавиша "стрелка" - вправо
        if xS1 >= 630 then EXIT;              // если звездолет у правого края формы
                                            // xS1 не изменяется
        xS1:= xS1 + dxS1;                    // движение звездолета вправо
    end;
    end;
end;
```

Запустим проект на компиляцию и взглянем на результаты нашего труда (см. рис.5):



Рис. 5. Движение звездолета по-горизонтали

Заключение

В следующих уроках мы добавим еще несколько движущихся объектов в проект.

Рассматриваемые в данной статье проекты полностью приведены в ресурсах к статье на <http://www.programmersforum.ru> в разделе «Журнал клуба программистов. Второй выпуск» (папка Lesson2).

Комментарий автора.

Перед запуском в среде Дельфи скопируйте в папку с проектом папку <data> с графическими файлами.



В данной статье рассматриваются базовые принципы работы с расширением SSE...

by Ivan_32 www.programmersforum.ru

С момента создания первого математического сопроцессора(FPU) для x86-процессоров, прошло уже около 30 лет. Целая эпоха технологий и физических средств их воплощения прошла, с тех пор, и нынешние FPU стали на порядок быстрее, энергоэффективней и компактней того первого FPU - 8087. С тех пор FPU стал частью процессора, что, конечно же, положительно сказалось на его производительности. Тем не менее, нынешняя скорость выполнения команд FPU оставляет желать лучшего. К счастью это лучшее уже есть. Им стала технология под названием SSE.

Аппаратное введение

SSE - Streaming SIMD Extensions - был впервые представлен в процессорах серии Pentium III на ядре Katamai.

SIMD - Single Instruction Multiple Data. Аппаратно, данное расширение состоит из 8 (позже 16, для режима Long Mode-x86-64) и конечно контрольного регистра MXCSR.

В последующих расширениях: SSE2, SSE3, SSSE3, SSS4.1 и SSE4.2 появлялись только новые инструкции, в основном нацеленные на специализированные вычисления.

К 2010-му году появились первые процессоры с поддержкой набора инструкций для аппаратного шифрования AES, этот набор инструкций тоже использует SSE-регистры.

Регистры SSE называются XMM и наличествуют XMM0-XMM7 для 32-битного Protected Mode и дополнительными XMM8-XMM15 для режима 64-битного Long Mode. Все регистры XMM-регистры 128-битные, но максимальный размер данных, над которым можно совершать операции, это FP64-числа. Последнее, обусловлено предназначением данного расширения - параллельная обработка данных.

Программное введение

Когда я только начинал работать с FPU, меня поразила невообразимая сложность работы с ним. Во-первых, из всех 8-ми регистров, прямой доступ был только к двум. Во-вторых, напрямую загружать данные в них нельзя было, то есть, скажем, инструкция fld 100.0 не поддерживается. А, в-третьих, из регистров общего назначения тоже нельзя было загрузить данные. Если вторая проблема в SSE не решена, то о первой и третьей подобного сказать нельзя.

В данном обзоре рассматриваются только SISD инструкции, призванные заменить FPU аналоги. Начнем-с. Перво-наперво стоит узнать, как же можно записать данные в xmm-регистр. SSE может работать с FP32 (float) и FP64(double) IEEE754-числами. Для прямой записи из памяти предусмотрены инструкции MOVSS и MOVSD.

Их мнемоники расшифровываются так:

MOVSS - **MOVE** SCALAR(Bottom) **SINGLE**

MOVSD - **MOVE** SCALAR(Bottom) **DOUBLE**

Данные инструкции поддерживают только запись вида XMM-to-MEMORY и MEMORY-to-XMM. Для записи из регистра общего назначения в регистр XMM и обратно есть инструкции MOVD и MOVQ.

Их мнемоники расшифровываются так:

MOVD - MOV DOUBLE WORD(DWORD)

MOVQ - MOV QUAD WORD(QWORD)

Перейдем к основным арифметическим операциям:

```
;; Сложение
ADDSS - ADD SCALAR SINGLE
ADDSD - ADD SCALAR DOUBLE
;; Вычитание
SUBSS - SUB SCALAR SINGLE
SUBSD - SUB SCALAR DOUBLE
;; Умножение
MULSS - MUL SCALAR SINGLE
MULSD - MUL SCALAR DOUBLE
;; Деление
DIVSS - DIV SCALAR SINGLE
DIVSD - DIV SCALAR DOUBLE
```

Примечание.

XMM-регистры могут быть разделены на два 64-битных FP64 числа или четыре 32-битные FP32 числа. В данном случае SINGLE и DOUBLE обозначают FP32 и FP64 соответственно. SCALAR - скалярное значение, выраженное одним числом, в отличие от векторного. В случае работы со скалярными значениями используется нижний SINGLE или DOUBLE (т.е. нижние 32 или 64-бита соответственно) XMM-регистров.

Недостаток SSE заключается в том, что среди инструкций нет тригонометрических функций. Sin, Cos, Tan, Ctan - все эти функции придется реализовать самостоятельно. Правда, есть бесплатная Intel Aproximated Math Library, скачать ее

можно по адресу: www.intel.com/design/pentiumiii/devtools/AMaths.zip.

В связи с данным фактом, в качестве алгоритма для практической реализации был выбран ряд Тейлора для функции синуса. Это, конечно, не самый быстрый алгоритм, но, пожалуй, самый простой. Мы будем использовать 8 членов ряда, что предоставит вполне достаточную точность.

В связи со спецификой Protected Mode, а именно - невозможностью прямой передачи 64-битных чисел через стек (нет, конечно можно, только по частям но неудобно), рассмотрим еще одну инструкцию, которую мы задействуем в нашей программе:

CVTSS2SD - ConVerT Scalar Single 2(to) Scalar Double

И ее сестра делающая обратное:

CVTSD2SS - ConVerT Scalar Double 2(to) Scalar Single

Данная инструкция принимает два аргумента, в качестве второго аргумента может выступать либо XMM-регистр либо 32-битная ячейка памяти - DWORD.

Примеры использования SSE-команд:

```
movss xmm0, dword[myFP32]
movss xmm0, xmm1
movss dword[myFP32], xmm0

movsd xmm0, qword[myFP64]
movsd xmm0, xmm1
movsd qword[myFP64], xmm0
```

```
add/sub/mul/div:
addss xmm0, dword[myFP32]
subsd xmm0, xmm1
mulss xmm0, dword[myFP32]
divsd xmm0, xmm1
```

Математическое введение

В качестве тестового алгоритма мы будем использовать ряд Тейлора для функции синуса. Алгоритм представляет собой простой численный метод (см. формулу-1):

В нашем случае мы используем 8 членов этого ряда, это не слишком много и вполне достаточно для того, что бы обеспечить довольно точные вычисления. Во всяком случае, отклонение от fsin (аппаратная реализация Sin - FPU) минимально. При этом, используемая формула выглядит так (см. формулу-2):

$$\sin(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)} x^{2n+1}; \quad (1)$$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - \frac{x^{11}}{11!} + \frac{x^{13}}{13!} - \frac{x^{15}}{15!}; \quad (2)$$

Программная реализация

В случае с SSE мы воспользуемся всеми восемью регистрами, а что касается FPU - мы будем использовать только st0 и st1. Благо, использование памяти в качестве буфера оказалось эффективней, чем использование всех регистров FPU, к тому-же так проще и удобней.

Вычисление будет проходить так. Сначала мы вычислим значения всех членов ряда, а потом приступим к их суммированию. Подсчет факториалов проводить не будем, так как это пустая трата процессорного времени в данном случае. Программная реализация на SSE:

```
proc sin_sse angle
;Нам понадобятся два экземпляра аргумента:
    cvtss2sd xmm0,[angle] ;; Первый будет выступать как результат возведения в
    степень
    movq xmm1,xmm0        ;; Второй как множитель, это сделано для того,
                           ;; чтобы минимизировать обращения к памяти

;; xmm0 = angle.
;; xmm1 = angle.        ;; далее x=X=Angle
    mulsd xmm0,xmm1      ;; Возводим X в третью степень
    mulsd xmm0,xmm1      ;
    movq xmm2,xmm0       ; xmm2 = xmm0 = x^3

    mulsd xmm0,xmm1      ;; Продолжаем возведение
    mulsd xmm0,xmm1      ;; Теперь уже в пятую степень
    movq xmm3,xmm0       ; xmm3 = xmm0 = x^5

    mulsd xmm0,xmm1
    mulsd xmm0,xmm1
    movq xmm4,xmm0       ;; xmm4 = xmm0 = x^7

    mulsd xmm0,xmm1
    mulsd xmm0,xmm1
    movq xmm5,xmm0       ;; xmm5 = xmm0 = x^9

    mulsd xmm0,xmm1
```

```

mulsd xmm0,xmm1
movq xmm6,xmm0    ;; xmm6 = xmm0 = x^11

mulsd xmm0,xmm1
mulsd xmm0,xmm1
movq xmm7,xmm0    ;; xmm7 = xmm0 = x^13

mulsd xmm0,xmm1 ;; Наконец возводим X в 15-ю степень и заканчиваем возведение
mulsd xmm0,xmm1 ;; xmm0 = x^15

;; Переходим к делению всех промежуточных результатов X ^ n, на n!.
divsd xmm0,[divers_sd+48] ; X^15 div 15!
divsd xmm7,[divers_sd+40] ; X^13 div 13!
divsd xmm6,[divers_sd+32] ; X^11 div 11!
divsd xmm5,[divers_sd+24] ; X^9 div 9!
divsd xmm4,[divers_sd+16] ; X^7 div 7!
divsd xmm3,[divers_sd+8]  ; X^5 div 5!
divsd xmm2,[divers_sd]    ; X^3 div 3!
subsd xmm1,xmm2 ; x - x^3/3!
addsd xmm1,xmm3 ; + x^5 / 5!
subsd xmm1,xmm4 ; - x^7 / 7!
addsd xmm1,xmm5 ; + x^9 / 9!
subsd xmm1,xmm6 ; - x^11 / 11!
addsd xmm1,xmm7 ; + x^13 / 13!
subsd xmm1,xmm0 ; - x^15 / 15!

;; В EAX результат не поместится
movq [SinsdResult],xmm1
;; Но если нужно добавить функции переносимости, есть два варианта
cvtsd2ss xmm1,xmm1
mov eax,xmm1
ret
SinsdResult dq 0.0
divers_sd dq 6.0,120.0,5040.0,362880.0,39916800.0,6227020800.0,1307674368000.0
endp

```

Что касается FPU версии данной функции, то в ней мы поступим несколько иначе. Мы воспользуемся буфером в виде 16*4 байт. В последний QWORD запишем результат. И в качестве делителя будем использовать память, это не страшно т.к. данные будут расположены на одной и той же странице, а это значит, что данная страница уже будет прокеширована и обращения к ней будут довольно быстрыми. Суммирование и вычитание членов ряда так же будет проведено в конце. Программная реализация на FPU:

```

proc sin_fpu angle

    fld [angle]          ; загружаем X. st0=X

    fmul [angle]
    fmul [angle]          ; st0 = X^3
    fld st0              ; st1 = st0

```

```
fdiv [divers_fpu] ; Делим  $X^3$  на 3! не отходя от кассы
fstp qword[res]   ; легким движением стека FPU, st1 превращается в st0 :)
;; qword[res] =  $x^3 / 3!$ 

fmul [angle]
fmul [angle]
fld st0           ; st0 = st1 =  $X^5$ 
fdiv [divers_fpu+8]
fstp qword[res+8]
;; qword[res+8] =  $x^5 / 5!$ 

fmul [angle]
fmul [angle]
fld st0           ; st0 = st1 =  $X^7$ 
fdiv [divers_fpu+16]
fstp qword[res+16]
;; qword[res+16] =  $x^7 / 7!$ 

fmul [angle]
fmul [angle]
fld st0           ; st0 = st1 =  $X^9$ 
fdiv [divers_fpu+24]
fstp qword[res+24]
;; qword[res+24] =  $x^9 / 9!$ 

fmul [angle]
fmul [angle]
fld st0           ; st0 = st1 =  $X^{11}$ 
fdiv [divers_fpu+32]
fstp qword[res+32]
;; qword[res+32] =  $x^{11} / 11!$ 

fmul [angle]
fmul [angle]
fld st0           ; st0 = st1 =  $X^{13}$ 
fdiv [divers_fpu+40]
fstp qword[res+40]
;; qword[res+40] =  $x^{13} / 13!$ 

fmul [angle]
fmul [angle]      ; st0 = st1 =  $X^{15}$ 
fdiv [divers_fpu+48]
fstp qword[res+48]
;; qword[res] =  $x^{15} / 15!$ 

fld [angle]        ; st0 = X
fsub qword[res]     ; X -  $x^3/3!$ 
fadd qword[res+8]   ; +  $x^5 / 5!$ 
fsub qword[res+16]  ; -  $x^7 / 7!$ 
fadd qword[res+24]  ; +  $x^9 / 9!$ 
```



```
fsub qword[res+32] ; - x^11 / 11!  
fadd qword[res+40] ; + x^13 / 13!  
fsub qword[res+48] ; - x^15 / 15!  
  
fstp qword[res+56] ; Сохраняем результат вычислений  
  
ret  
res_fpu dq 0.0  
res dd 14 dup(0)  
divers_fpu dq  
6.0,120.0,5040.0,362880.0,39916800.0,6227020800.0,1307674368000.0  
endp
```

Обе функции были протестированы в программе WinTest и вот ее результаты: sin_FPU - 145-150 тактов в цикле на 1000 итераций и около 1300-1800 при первом вызове при использовании FP64 и 150-165 для FP80.

Такая потеря скорости связана с тем, что при первом вызове память еще не прокеширована. sin_SSE - около 140-141 тактов в цикле на 1000 итераций, при первом вызове результат аналогичный FPU.

Примечание. Основываясь на результатах тестирования, я могу сказать, что разница в результатах может быть лишь погрешностью. Это было проверено опытным путем. Я несколько раз перезагружал компьютер и в разных случаях выигрывал SSE или FPU. Это дает повод предположить, что имела место небольшая погрешность и разница в результатах является ее и только ее порождением. Но Intel Optimization Manual говорит об обратном. По документации разница между SSE и FPU командами около 1-2 тактов в пользу SSE, т.е. SSE-команды на 1-2 такта в среднем, выполняются быстрее.

оказалась очень медленной инструкцией, а SSE не помогло даже то что страница с данными находилась в кэше.

На заметку: также я тестировал SSE через память (аналогично FPU-алгоритму) и FPU через использование всех регистров, в обоих случаях имела место серьезная потеря производительности. 220-230 тактов для SSE-версии с использованием буферов и около 250-300 для FPU через регистры. FXCH -

Заключение

Как показала практика, при использовании SSE в качестве FPU мы почти ничего не теряем. Важно то, что такое однопоточное SISD использование не является эффективным. Всю свою настоящую мощь SSE показывает именно в параллельных вычислениях. Какой смысл считать за N тактов, 1 FP32 сложение/вычитание или любую другую арифметическую операцию, если можно посчитать за те же N-тактов целых четыре FP32 или 2 FP64. Вопрос остается лишь в распараллеливании алгоритмов. Стоит ли использовать SSE? Однозначно стоит. Данное расширение присутствует во всех процессорах, начиная с Pentium III и AMD K7.

Важно:

Регистры XMM предположительно не сохраняются при переключении задач и точно не сохраняются при использовании API. Тот же GDI+ не восстанавливает их значения.

Nota Bene:

1. Тестирование проводилось на процессоре с не самым старым ядром. Еще при релизе мелькала фраза о масштабных оптимизациях во всех блоках. При схожей частоте данный процессор в ряде приложений оказывается быстрее, чем, скажем Core 2 на ядре Conroe(первое поколение Core 2). Это собственно к чему: SSE не всегда было таким быстрым, как и FPU. На разных архитектурах вы увидите как выигрыш от использования SSE так и серьезный проигрыш.
2. Данный численный метод не является самым быстрым, он даже не распараллелен. Аппаратный FSIN выполняется за 80-100 тактов с FP80/FP64 точностью. Существуют так же другие численные методы для нахождения тригонометрических и других функций, которые намного эффективней данного и практически позволяют сделать эти вычисления более быстрыми, нежели FSIN.

Программно-аппаратная конфигурация:

CPU: Intel Core 2 Duo E8200 2.66 Ghz @ 3.6 Ghz 450.0 FSB * 8.0

RAM: Corsair XMS2 5-5-5-18 800Mhz @ 900Mhz 5-6-6-21. FSB:MEM = 1:2

MB: Gigabyte GA P35DS3L (BIOS неизвестен - никогда не изменялся)

GPU: Sapphire Radeon HD5870 1GB GPU Clock = 850 Mhz Memory Clock = 4800(1200 Phys)

PSU: Cooler Master Elite 333 Stock PSU 450 Wt

OS: Windows 7 Ultimate x86

FASM: 1.67 recompiled by MadMatt(Matt Childress)

Ресурсы

- . Полезный ресурс <http://users.egl.net/talktomatt/default.html>
- . Программа для тестирования времени выполнения
<http://programmersforum.ru/showthread.php?t=55270> (автор данной программы некто **bogrus**.
Его профиль есть на форуме WASM.RU но, он неактивен уже 3-й год)

«Компьютер никогда не заменит человека»
(с) Людоед.

- С жидкокристаллическим монитором у меня не лады.
- А что так?
- Не выкристаллизируется.

486 дней вокруг света.

CPU not found! Users software emulation!

Не забуду мать родную - ZX Spectrum навсегда!

В Китае создан новый суперкомпьютер, объем оперативной памяти 40 миллионов счетных палочек.

Верным курсором идете, товарищи...

Винчестеры? Беру! А где патроны?

Национальность - русский, вероисповедание - программист.

Спама бояться - соб@ку не заводить.

Если программа полезна, ее нужно доработать, а если - бесполезна, то следует разработать к ней документацию.

Жили у бабуси два веселых GUS'я...

Идет подготовка к зависанию компьютера...

Компьютеры делают ЭТО в ASCII...

Купите себе более мощный компьютер, чтобы быстрее перезагружаться.

Готовой работающей программой называют код, содержащий пока еще не обнаруженные ошибки.

Рукописи не горят - их форматируют тупые юзеры...

А у вас Интернет кошерный?

Настоящий программист всегда уверен в том, что он ни в чем не уверен.

Знаете ли вы, что у компьютеров старше 60 лет операционная система уже не встает?

Произвожу балансировку компакт-дисков...

...я еще из тех, кто учился азбуке по букварю, а не по клавиатуре.

Смерть - это всего лишь программа заложенная в организм человека и как любая программа она имеет свои коды доступа.

Документооборот становится по настоящему электронным когда из его програмного обеспечения убирается кнопка «Печать».

В 20 - все ищут Форму, в 30 - Содержание, а когда Форма приобретает Содержание,... почему-то виснет поисковый сервер...

Вороне Бог на вход послал 4 Вольта...

Свойство зеркала имело - DirectDraw оно умело...

Enter нажат, к чему теперь рыдания...

Там были лишние файлы, которые не давали работать...

Хороший программист характеризуется умением доказать почему задачу невозможно выполнить, когда ему просто лень её выполнять...

Тех, кто презирает программистов, программисты презирают сильнее, чем те, кто презирает программистов, презирают программистов, которые презирают тех, кто их презирает...

