FPS

Blender
Кластерный рендеринг

Язык D
Шаблоны. Часть II
Стиль D

«Мой язык — лучший!»
Заблуждения в программировании

Что такое QR-код?
«Эти странные квадратики...»

и многое другое

№15 • 2011

Независимый электронный журнал о разработке компьютерных игр

Нас читают:

gcup.ru
Все для начинающего и профессионального разработчика игр

xtreme3d.narod.ru
Сайт движка Xtreme3D

make-games.ru
Портал создания игр

gamer-club.ucoz.com
Все для создания игр без программирования и не только

mizzystic.ru
Крупнейший информационный Game Maker портал

xbobr.at.ua
Создание игр, программ, музыки

portalgame.net.ru
Игровой портал

gamesfpscreator.at.ua
Сайт для помещений игр и обсуждаловок

gameshaker.ukoz.ru
Все для редактирования и создания игр

esate.ru
Мультимедиа-сообщество

www.mobkiosk.com
"Мобильный киоск"

dogames.ru
Мастерская игр

gamecreate.ru
Создай свою игру!

igrostroenie.net.ru
Игровые движки и ресурсы

rus.game-maker.ru
Русское сообщество Game Maker

gacon.ucoz.ru
Сайт, посвященный разработке игр

game.oxnull.net
Разработка игр на конструкторе Game Maker

bigslava.3dn.ru
Сообщество творческих людей

Главный редактор: Тимур Гафаров
Дизайн и верстка: Тимур Гафаров
По вопросам сотрудничества обращаться по адресу gecko0307@gmail.com.
Официальный сайт журнала: http://fpsmag.zymichost.com

ЭЛЕКТРОННЫЙ ЖУРНАЛ о разработке компьютерных игр

№ 15 '11

В этом выпуске:

Blender

Netrender

КЛАСТЕР – это набор вычислительных узлов, самостоятельных компьютеров, связанных высокоскоростной сетью и объединенных в логическое целое специальным программным обеспечением. Фактически простейший кластер можно собрать из нескольких персональных компьютеров, находящихся в одной локальной сети, просто установив на них соответствующее ПО. Традиционно для создания кластеров используется Linux (более 70% систем) и другие разновидности Unix (оставшиеся 30%).

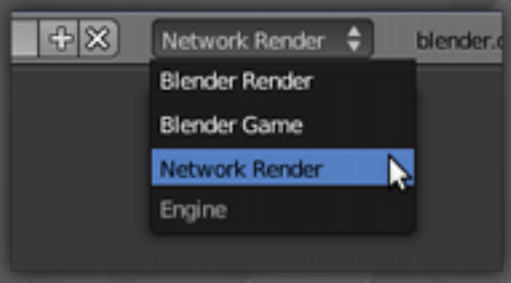
В чем идея подобного объединения? Кластеры, как правило, ассоциируются с суперкомпьютерами, круглые сутки решающими какую-нибудь сверхбольшую задачу, но на практике существует и множество куда более «приземленных» кластерных применений. Часто встречаются кластеры, в которых одни узлы, дублируя другие, готовы в любой момент перехватить управление, или одни узлы, проверяя получаемые с другого узла результаты, радикально повышают надежность системы. Еще одно популярное применение кластеров – решение задачи массового обслуживания, когда серверу приходится отвечать на большое количество независимых запросов. Такую систему обычно называют «серверной фермой».

Большинство суперкомпьютеров в мире построено на основе концепции параллельных вычислений. Большая вычислительная мощность достигается путем сложения мощностей отдельных вычислителей на задачах, которые можно хорошо разделить между этими вычислителями. Например, знаменитый Deep Blue, который выиграл в шахматы у Каспарова, представляет собой объединение нескольких сотен процессоров RS/6000 (подробнее о Deep Blue и других шахматных компьютерах читайте в «FPS» №12 '10).

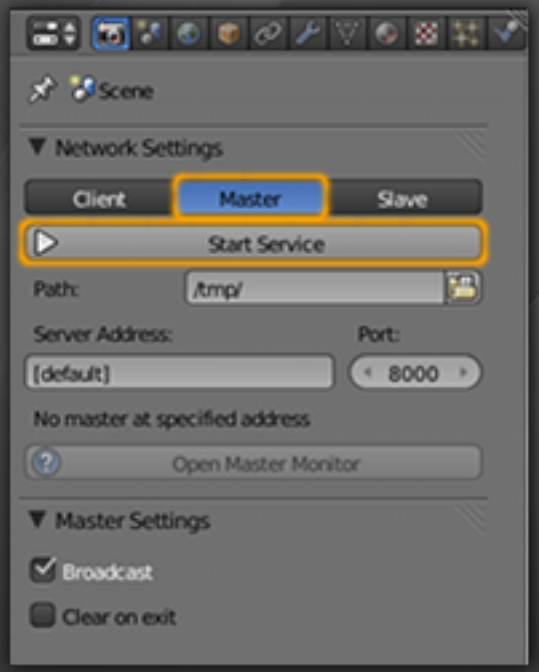
Многие голливудские компании, занимающиеся мультипликацией, например Pixar и Industrial Light and Magic, используют кластеры компьютеров для рендеринга. Просчитанные по сети предпросмотры и кэширование ускоряют процесс рендеринга в целом, от старта к финишу. Кластер, предназначенный для рендеринга компьютерной графики – сеть распределенного рендеринга – получил название «рендер-ферма» (renderfarm).

Все крупные профессиональные пакеты для работы с 3D-графикой имеют встроенную поддежку рендер-ферм (в частности, распределенный рендеринг имеется в Vray и Mental Ray). Blender также не остался в стороне. Сетевой рендеринг в Blender 2.5 реализован в качестве плагина (Add-Ons > Render > Network Renderer) – в версии 2.58 он помечен как стабильный, но находящийся на стадии активной разработки.

Принцип следующий: на одной машине вы запускаете так называемый master-сервер, а на всех остальных – slave-серверы.



1. На главной машине запустите master-сервер: активируйте плагин, переключите движок рендеринга на Network Renderer. В качестве режима операции выберите Master.



2. Нажмите Start Service. Строка статуса рендеринга будет отображать действия сервера. Остановить его работу можно нажатием клавиши Esc.

3. На всех остальных машинах кластера запустите slave-сервер: активируйте плагин, переключите движок рендеринга на Network Renderer. В качестве режима операции выберите Slave.

4. Нажмите кнопку со стрелками, чтобы автоматически оперелить IP-адрес master-сервера. Нажмите Start Service. Строка статуса рендеринга будет отображать действия сервера. Остановить его работу можно нажатием клавиши Esc.

5. На том компьютере, где находится ваш проект, запустите клиентский процесс: измените настройки рендеринга, сохраните blend-файл, активируйте плагин, переключите движок рендеринга на Network Renderer. В качестве режима операции выберите Client.

6. Нажмите кнопку со стрелками, чтобы автоматически оперелить IP-адрес master-сервера. Отправьте задачу на выполнение кнопкой Send Job.

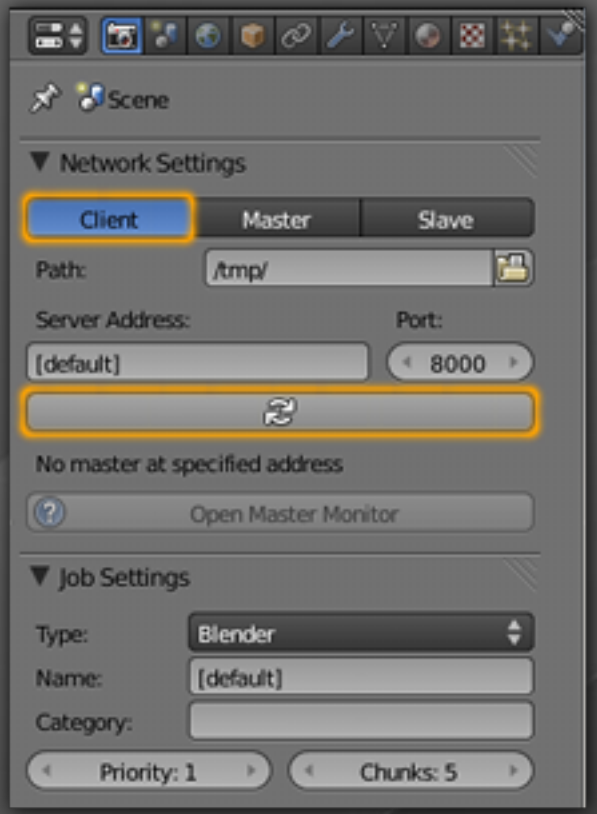
Теперь вы можете использовать комбинацию Ctrl+F12, чтобы отрендерить анимацию на кластере. Готовые кадры будут появлятся автоматически. Прирост скорости рендеринга будет обратно пропорционален количеству slave-серверов.

Можно также запустить серверы в консольном режиме. Сохраните master-сервер в файл *.blend и запускайте командой

```
$ blender -b master.blend -a
```

Аналогично – для slave-сервера:

```
$ blender -b slave.blend -a
```



becko
geckoo307@gmail.com

OpenGL 4.2

9 АВГУСТА 2011 г. организация Khronos Group представила обновленную версию спецификации OpenGL 4.2 и языка описания шейдеров GLSL 4.20. Новая версия спецификации обратно совместима с предыдущими версиями OpenGL и содержит улучшения, подготовленные на основе пожеланий разработчиков графических приложений и игр.

Компания NVIDIA выпустила тестовую версию проприетарных видеодрайверов с поддержкой OpenGL 4.2 сразу после публикации спецификации. Драйвер полностью поддерживает OpenGL 4.2 для карт NVIDIA GeForce 400/500 (Fermi) и доступен для платформ Windows, Solaris, Linux и FreeBSD. Компания AMD сообщила о намерении выпустить в ближайшие дни бета-версию драйверов AMD Catalyst с поддержкой OpenGL 4.2.

Поддержка OpenGL 4.2 в свободной библиотеке Mesa пока не планируется в обозримом будущем. В настоящее время в Mesa 3D полностью обеспечена поддержка OpenGL 2.1 и частично OpenGL 3.0, довести до конца работу над поддержкой всех возможностей OpenGL 3.0 планируется к концу года.

Из добавленных в OpenGL 4.2 улучшений можно отметить:

- Возможность использования в шейдерах атомарных счетчиков и атомарных операций модификации (атомарный цикл чтение-изменение-запись) для одного уровня текстур. Эти возможности могут быть использованы одновременно, например, для использования счетчика для каждого пикселя в буфере, используемом для однопроходной отрисовки, независимо от порядка выбора пикселов.

- Возможность геометрических преобразований с использованием тесселяции на стороне GPU и отрисовки нескольких экземпляров полученных преобразований, что позволяет эффективно менять позицию и воспроизводить копии для сложных объектов;

- Поддержка изменения произвольной части сжатой текстуры, без повторной загрузки в GPU текстуры целиком, что позволяет добиться существенного роста производительности;

- Поддержка упаковки нескольких 8- и 16-разрядных значений в одно 32-разрядное значение для эффективной обработки шейдеров со значительным сокращением используемого объема памяти и повышением пропускной способности. Например, подобная упаковка особенно полезна для организации передачи данных между различными стадиями выполнения шейдера.

Стиль D

КАК И ВО МНОГИХ других языках, в D есть свой стиль программирования – ряд правил «хорошего тона» для грамотного, читаемого кода. Эти правила ни к чему не принуждают – вы, разумеется, вольны писать программы как вам угодно. Однако их рекомендуется соблюдать – особенно в том случае, если с вашим кодом будут работать другие люди. К примеру, стиля D придерживаются библиотека Phobos и компилятор DMD.

http://www.digitalmars.com/d/2.0/dstyle.html

На каждую строку – не более одного утверждения.

•

Вместо аппаратных знаков табуляции следует использовать пробелы.

•

Длина отступа – четыре пробела.

Операторы отделяются от операндов пробелами.

•

Тела функций отделяются двумя пустыми строками.

•

Объявления переменных отделяются от утверждений одной пустой строкой.

•

Одно утверждение комментируется двумя косыми чертами:

```
statement; // comment
statement; // comment
```

•

Набор утверждений комментируется блоком:

```
/* comment
 * comment
 */
statement;
statement;
```

Неправильный код следует обрамлять вложенным блоком:

```
/+++++
/* comment
 * comment
 */
 { statement;
   statement;
 +++++/
```

•

Названия функций, переменных и перечислений (enum), состоящие более чем из одного слова, оформляются «верблюдом» (первая буква первого слова – строчная, у остальных слов – заглавная):

```
int mySuperFunction();
```

•

Названия классов и структур пишутся через заглавную букву:

```
class Foo;
class FooAndBar;
```

Названия не должны начинаться со знака подчеркивания («_»), если они не принадлежат private-членам классов.

•

Названия модулей не должны содержать заглавные буквы – во избежание проблем на файловых системах, нечувствительных к регистру.

•

По возможности следует избегать объявления бессмысленных псевдонимов, например:

```
alias void VOID;
alias int INT;
alias int* pint;
```

•

Одинаковые объявления следует группировать:

```
int[] x, y;
int** p, q;
```

•

При перегрузке операторов не следует назначать им нелогичную и неочевидную семантику – например, наделять оператор «+» смыслом, отличным от операции «сложение».

Язык D. Шаблоны

Часть II

МЫ ПРОДОЛЖАЕМ тему о шаблонах в языке D, начатую в «FPS» №13 ('11). Обобщенное программирование в D – это не просто параметризованные шаблоны функций и классов. D развивает эту идею, позволяя напрямую «общаться» с компилятором и получать информацию о типах на этапе сборки программы – причем средства для этого встроены в сам язык.

Начнем с простого – директивы pragma. Она уже знакома «плюсистам» – в C++ это нестандартная препроцессорная инструкция, и ее частое использование является признаком дурного тона. В D pragma – это часть стандарта языка. Например,

```
pragma(msg, "Hello, world!");
```

во время компиляции выведет строку «Hello, world!».
Это полезно для вывода на экран различных предупреждений, отладочных сообщений, для пометки deprecated-кода и т.д.

Один из способов ее применения – вывод типа того или иного значения:

```
pragma(msg, typeof([["foo"[]: "bar"[]]: 5]).stringof);
```

Эта команда выведет следующий текст:

AssociativeArray!(const(immutable(char)[])[string],int)

В D возможна экстренная остановка компиляции. Для этого используется специальная конструкция – static assert. Напомним, что assert обрабатывает исключение, проверяя заданное значение на истинность; если в assert передать 0 или false, он гарантированно остановит работу программы. К примеру, если ваша программа не поддерживает Linux, можно написать так:

```
version(linux)
{
   pragma(msg, "Linux is not supported");
   static assert(0);
}
```

Существуют различные способы пустить компиляцию «по другому руслу». Очень часто в шаблонах используется static if – статическая проверка условия. Она напоминает препроцессорную инструкцию #ifdef, но гораздо мощнее, так как имеет доступ к статическим данным, константам и параметрам шаблонов. Классический пример – вычисление факториала во время компиляции:

```
template factorial(uint n)
{
   static if (n < 2)
      const uint factorial = 1;
   else
      const uint factorial = n * factorial!(n-1);
}
```

Использование: uint f = factorial!10;

Непосредственно получение информации о классах и типах (интроспекция или рефлексия) реализовано в специальном расширении языка – traits. Мы рассмотрим частный случай интроспекции, когда необходимо узнать, скомпилируется ли тот или иной код.

Возьмем, к примеру, шаблон функции, которая содержит операцию конкатенирования:

```
void append (T) (ref T[] x, T y)
{
   x ~= y;
}
```

Обычно конкатенирование применяется для добавления нового элемента в массив – однако мы не будем связываться с массивами и, вместо этого, параметризуем шаблон двумя типами T и S, чтобы сохранить поддержку классов, перегружающих оператор « ~= »:

```
void append (T, S) (ref T x, S y)
{
   x ~= y;
}
```

Все бы хорошо, да вот беда – шаблон теперь можно инстанцировать любыми типами – даже теми, которые не поддерживают конкатенирование. Конечно, компилятор этого в любом случае не допустит. Но, оказывается, мы и сами можем осуществить проверку:

```
void append (T, S) (ref T x, S y)
{
   static if ( __traits (compiles, {x ~= y;}) )
      x ~= y;
   else static assert (0,
      "Operation is not supported for types " ~
      T.stringof ~ ", " ~ S.stringof);
}
```

Директива __traits с параметром compiles принимает любое выражение или утверждение, возвращая истину, если оно семантически правильно. В нашем случае это {x ~= y;}.

Итак, наш шаблон append стал более «умным». Правда, не вполне удобно два раза писать искомый код – это становится более ощутимым, если он представляет собой нечто большее, чем одну операцию. Тут на помощь приходит подмешивание (mixin):

```
void append (T, S) (ref T x, S y)
{
   enum code = q{
      { x ~= y; }
   };
   static if ( __traits (compiles, mixin (code)) )
      mixin (code);
   else static assert (0,
      "Operation is not supported for types " ~
      T.stringof ~ ", " ~ S.stringof);
}
```

Мы прописываем «критичный» код только один раз, и затем подмешиваем его везде, где нужно. Прописываем в виде константной строки, которая доступна под именем «code». А для строк, содержащих D-код, предусмотрен специальный синтаксис – q{ … }. Это было введено для того, чтобы текстовые редакторы подсвечивали синтаксис таких строк, как если бы это был обычный код.

Однако теперь шаблон выглядит немного нелепо. Отдадим дань стилю D и переоформим его с учетом контрактной парадигмы. Финальная версия append будет выглядеть следующим образом:

```
void append ( T, S,
   string code = q{{x ~= y;}} ) (ref T x, S y)
in {
   static assert ( __traits (compiles, mixin (code)),
      "Operation is not supported for types " ~
      T.stringof ~ ", " ~ S.stringof);
}
body {
   mixin (code);
}
```

Проверка компилируемости осуществляется перед входом в тело функции, в контракте «in» – это значительно повышает чистоту и читаемость кода. Кроме того, «критичный» код теперь и вовсе вынесен в параметры шаблона, что позволяет инстанцировать его любой операцией:

```
int value = 6;
append!(int, int, "{ x += y; }") (value, 5);
```

Правда, при этом вызов функции стал более запутанным. В то же время, изначальное предназначение шаблона – добавление элемента в массив – остается все таким же простым:

```
string[] arr;
append(arr, "hello");
```

Конечно, данный пример достаточно далек от «реального мира» и представляет собой излишнее усложнение простейшей задачи. Но он наглядно демонстрирует впечатляющие возможности D в области метапрограммирования. Они позволяют писать более безопасный код, облегчают отладку программ и открывают дорогу всевозможным инновациям.

Перегрузка операторов в D

ПЕРЕГРУЗКА операторов – важный элемент полиморфизма в ООП-языках. Будучи всего лишь «синтаксическим сахаром» (syntactic sugar), перегрузка операторов оказывается очень полезной во многих ситуациях, делая поведение пользовательских типов более похожим на поведение встроенных. Вы можете, например, переопределить операторы арифметических действий для векторов или матриц:

```
vector3f c = a + b;
```

будет интерпретировано как

```
vector3f c;
c.x = a.x + b.x;
c.y = a.y + b.y;
c.z = a.z + b.z;
```

В языке D уделено внимание удобству и универсальности перегрузки операторов. Вы можете переопределять унарные и бинарные операторы, операторы присваивания и сравнения, вызова функций, чтения и записи по индексу или по диапазону. При этом широко используется обобщенное и контрактное программирование.

Наример, для перегрузки оператора «+» используется метод opAdd:

```
T opAdd(T b)
{
   ...
}
```

Но можно то же самое написать так:

```
T opBinary (string op) (T b) if (op == "+")
{
   ...
}
```

Для классов-контейнеров незаменима перегрузка оператора индекса:

```
T opIndex (int index)
{
   return get (index);
}

void opIndexAssign (T val, int index)
{
   set (val, index);
}

auto a = obj[10];
obj[4] = 100;
```

...и конкатенирования:

```
void opCatAssign(T k) {
   ...
}

obj ~= 10;

void opCatAssign(T[] k) {
   ...
}

obj ~= [5, 8, 10, 32];
```

В некоторых случаях бывает нужно перегрузить оператор in. Например, он может возвращать логическое значение:

```
bool opIn_r (T val) {
   if ( have(val) ) return true;
   else return false;
}

if (50 in obj) { ... }
```

...или указатель на объект:

```
T* opIn_r (string key) {
   return &storage[string];
}

auto p = "name" in obj;
```

Наконец, доступно перенаправление (forwarding) несуществующих членов класса при доступе к ним (так называемая диспетчеризация вызовов – фактически, перегрузка оператора «точка»):

```
struct S {
   void opDispatch (string s, T) (T i) {
      writefln("S.opDispatch('%s', %s)", s, i);
   }
}

S s;
s.foo(7);
```

Перенаправлять можно не только методы, но и свойства:

```
struct D {
   template opDispatch(string s) {
       enum int opDispatch = 8;
   }
}

D d;
writeln(d.foo);
```

Gecko
gecko0307@gmail.com

Singleton

Плюсы и минусы

В РАЗРАБОТКЕ программного обеспечения, шаблон проектирования или паттерн (англ. design pattern) — повторимая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста. Популярность паттернов во многом обязана своим ростом книге «Design Patterns — Elements of Reusable Object-Oriented Software», в которой были описаны 23 шаблона проектирования. Авторы книги — американские программисты Эрих Гамма, Ричард Хелм, Ральф Джонсон и Джон Влиссидс — стали известны общественности под названием «Банда четырех» (англ. «Gang of Four», часто сокращается до «GoF»).

В этой статье мы рассмотрим один из самых распространенных паттернов — singleton (одиночка). Это класс, который может иметь только один экземпляр. Как правило, singleton инстанцируется лениво — то есть, только когда его экземпляр будет действительно необходим. Singleton может пригодится для объекта-менеджера, который существует на всем протяжении работы программы и управляет другими объектами.

«Forgive Me Father, For I Have Singleton'ed»
Kevlin Henney

Весь код в статье приведен на языке D. Простейший singleton можно объявить и использовать следующим образом:

```
class Singleton
{
   private:
   static Singleton instance = null;
   this() { }

   public:
   static Singleton opCall()
   {
      if (instance is null)
         instance = new Singleton;
      return instance;
   }
}

auto st1 = Singleton();
auto st2 = Singleton();
assert (st1 == st2);
```

Однако за простотой кроется опасность: как только вы начнете использовать в своей программе более одного потока (а вы рано или поздно начнете), класс Singleton уже не будет гарантировать свое «одиночество». Два потока могут одновременно обратиться к методу opCall и создать два экземпляра Singleton. Иными словами, такая реализация потоково-небезопасна (not thread-safe). Как же быть?

К счастью, разработчики D все предусмотрели. Данные, безопасно разделяемые между потоками, в D обозначаются ключевым словом shared. Кроме того, язык предусматривает возможность заблокировать код для выполнения более чем одним потоком одновременно — для этого был введен спецификатор synchronized. Код, помеченный таким образом, снабжается мьютексом.

```
class Singleton {
   private:
   static shared Singleton instance = null;
   this() { }

   public:
   static shared(Singleton) opCall() {
      if (instance !is null)
         return instance;
      synchronized {
         instance = new shared(Singleton)();
         return instance;
      }
   }
}
```

Можно реализовать класс Singleton в виде шаблона, от которого будут наследовать другие классы, которые также должны существовать в единственном экземпляре:

```
class Singleton (T) {
   private:
   this() { }

   protected:
   static shared T instance = null;

   public:
   static shared(T) opCall() {
      if (instance !is null)
         return instance;
      synchronized {
         instance = new shared(T)();
         return instance;
      }
   }
}

class OnlyOne: Singleton!(OnlyOne)
{
   private this() {}
}

auto st1 = OnlyOne();
auto st2 = OnlyOne();
assert (st1 == st2);
```

Синглтоны в частности и паттерны в целом нередко подвергаются критике. Необходимо помнить, что шаблоны проектирования — всего лишь инструмент для понимания абстрактных концепций. Если вы обладаете этим пониманием, привязывать себя к «рецептам» из книги бессмысленно и даже вредно. Зачем ограничивать себя в возможности находить оригинальные, творческие решения? Паттерны хороши там, где их полезность доказана и проверена на практике. А слепое применение шаблонов из справочника, без осмысления причин и предпосылок выделения каждого отдельного шаблона, замедляет профессиональный рост программиста, так как подменяет творческую работу механической подстановкой.

Считается, что знакомиться со списками шаблонов необходимо тогда, когда программист «дорос» до них в профессиональном плане — и не раньше. Хороший критерий нужной степени профессионализма — выделение паттернов самостоятельно, на основании собственного опыта.

«Мой язык — лучший!»

Заблуждения в программировании

ЗА ПОЛВЕКА существования компьютеров инженерами было создано несколько тысяч языков программирования. Разумеется, далеко не все из них сейчас актуальны, поэтому сократим список хотя бы до ста языков, получивших более-менее широкое распространение. Примерно половину из этой сотни можно считать устоявшимися стандартами. Из них для создания основной массы программного обеспечения используется не больше двадцати. Таким образом, можно сокращать список языков по критерию распространенности. Но можно ли сказать, что самый распространенный язык – самый лучший?

На самом деле, лучшего языка нет – как нет и лучшей программы. Все они обладают своими достоинствами и недостатками. И первым критерием при выборе языка должен быть вопрос: подходит ли этот язык для решения поставленной задачи? Ведь не будете же вы, в самом деле, писать ядро ОС на Java или скрипт автоматизации – на ассемблере. Но, к сожалению, многие начинающие программисты не понимают этой простой истины, и в Сети очень часто можно встретить многостраничные треды словесных баталий, где эти «специалисты» доказывают друг другу, чем, к примеру, Lisp лучше C++. К слову сказать, если бы употребить эти затраченные на бессмысленную полемику энергию и время на написание того же по объему кода (на любом, заметьте, языке!) – вышло бы куда больше пользы.

Первое и самое главное заблуждение большинства программистов и тех, кто считает себя таковыми: *«Чтобы стать хорошим программистом, необходимо выучить язык XXX»*. Возникает ощущение, что все забыли суть программирования – построение алгоритмов для эффективного решения задач. Если вы можете это делать – вы можете стать хорошим программистом, вне зависимости от языка, которым пользуетесь. А знание языка – не самоцель и никогда ею не было.

Второе заблуждение является логическим продолжением первого: *«Чтобы стать лучшим программистом, необходимо с самого начала учить самый сложный язык»*. На самом деле, никогда не следует начинать со сложного. Никто не учится водить машину на болидах «Формулы-I». Вы можете начать с любого языка, который покажется вам проще. Выберите самый простой язык, который вы понимаете, изучите его полностью и станьте лучшим программистом на нем. И, только когда его возможностей перестанет хватать, переходите на более сложный.

Третье заблуждение, очень распространенное – даже в среде профессионалов, звучит так: *«Я могу написать что угодно на языке, который знаю. Изучать другие языки и даже смотреть в их сторону – пустая трата времени. Мой язык – лучший!»*. Это самое опасное заблуждение, так как не ведет ни

к чему, кроме застоя и деградации. Языки программирования – это всего лишь инструменты. Чем больше у вас разных инструментов, тем легче и удобнее вам выполнять работу. Хороший программист знает 3 - 4 языка, и даже больше – используя каждый из них там, где этот язык подходит лучше всего. К тому же, языки беспрерывно эволюционируют, аккумулируя концепции своих предшественников и приобретая новые возможности. Игнорируя эту эволюцию, вы рискуете отстать от жизни. Никогда не стесняйтесь изучать новые языки – более того, стремитесь к этому.

Четвертое заблуждение проистекает от третьего: *«В языке XXX нет возможности YYY, которая есть в моем языке. Поэтому он неполноценен и не годится для серьезной работы»*. Любой язык спроектирован с определенными идеями и целями. Одна из таких идей (причем, не такая глупая, как многим может показаться) – минимализм и простота. Если некая возможность не была встроена в язык, значит, на это была своя причина. Создатели языка, учитывая его дизайн и структуру, посчитали эту возможность излишней. Если лично вам непременно нужна такая возможность, просто выберите другой язык. Какой смысл в том, что все языки будут превращаться в «комбайны», наполненные редко используемыми функциями? «Комбайны» нужны далеко не всем и не всегда.

Следующее заблуждение является основной причиной постоянных споров: *«Язык XXX быстрее языка YYY»*. Языки не могут быть быстрыми или медленными. Быстрыми или медленными бывают программы. А скорость программ – величина относительная. Одна и та же программа на разных машинах будет выполняться с разной скоростью. Более того, оптимизация скорости программ – настоящее искусство, требующее высокого профессионализма и мастерства. А программист, обладающий достаточным мастерством и опытом, сумеет писать быстрые программы на любом языке. Конечно, бывает так, что хорошо написанный код на одном языке будет работать быстрее столь же хорошо написанного кода на другом. Но это становится критичным только в редких случаях, и практически никогда – в обычном прикладном программировании. Как правило, если ваш код работает слишком медленно – с высокой долей верятности, в этом виноват не язык, а вы сами.

Аналогичное заблуждение: *«Язык XXX мощнее языка YYY»*. Ни один язык не создавался для решения всех проблем на свете. Каждый из них эффективен в своей предметной области. И совсем не обязательно должен быть эффективным в других областях. Поэтому понятие «мощности» языка имеет смысл только применительно к определенному кругу задач, для решения которых он был спроектирован.

Помните, что самого лучшего языка не существует. А раз так – нет смысла искать его, сравнивая языки друг с другом. Просто используйте их по назначению. Нет языков хороших и плохих, есть хорошие и плохие программисты.

Gecko
gecko0307@gmail.com

Вечный вопрос

Пробелы против табуляции

ПРИ НАПИСАНИИ исходного кода очень важно использовать отступы. Ваш стиль программирования и оформление кода говорят о вас очень многое. Иногда даже больше, чем готовая рабочая программа. Достаточно взглянуть на код, чтобы понять, какой перед вами программист – трудолюбивый и аккуратный или ленивый и небрежный.

Поскольку для компилятора форматирование и отступы не имеют абсолютно никакого значения (если, конечно, речь не идет о языках, где отступы являются частью синтаксиса – например, Python), многие забывают, что их программу может прочитать не только компилятор, но и другие люди, и не уделяют индентации должного внимания. В результате получается совершенно нечитаемый, «индусский» код.

Однако даже если вы используете отступы, громадное значение имеет вопрос: как именно это делать. Уже много лет идут споры, что использовать для этой цели – пробелы или символ табуляции (клавишу Tab)? Табуляцию все текстовые редакторы обрабатывают по-своему. Обычно она эквивалентна 2, 4 или 8 обычным пробелам – в зависимости от пользовательских настроек.

Часто можно услышать, что «табуляция – зло» или «пробелы – зло». На самом деле все, конечно, зависит от ваших собственных предпочтений. Пробелы могут кого-то раздражать, но и табуляция – не панацея. Представьте, например, такую ситуацию:

```
enum {
    TABS = 2,
    SPACES = 1
}
```

Нет ничего дурного в использовании табуляции для отступа *всей* строки. Однако многие используют их, чтобы выравнивать значения:

```
enum {
    TABS   = 2,
    SPACES = 1
}
```

Это нужно делать строго при помощи пробелов! Используя при этом знаки табуляции, вы всего лишь добиваетесь нужного результата в *вашем* текстовом редакторе и при *ваших* настройках. Если у вас длина табуляции – 8 пробелов, такое «форматирование» не сохранится при табуляции в 2 или 4 пробела. Попробуйте и убедитесь сами:

```
enum {
  TAB   = 0,
  SPACE = 1
}
```

Другая распространенная ошибка: многие включают в редакторе опцию «записывать пробелы вместо знаков табуляции». Нажимая клавишу Tab, пользователь на самом деле вставляет 2, 4 или 8 пробелов. Это может быть не страшно, если так диктуют правила дизайна проекта, над которым вы работаете. Но не стоит делать этого для кода, который вы специально пишете для других людей – для примеров, демонстраций, уроков, учебников и т. д. Так как этот код будут использовать множество людей с самыми разными предпочтениями, они вряд ли помянут вас хорошим словом, если им придется потратить много времени на переформатирование. Используйте в таких случаях обычную табуляцию. Если кому-то не нравится, он всегда может запустить автоматический поиск и замену знаков табуляции на N-ное количество пробелов – а вот обратное не всегда справедливо.

Gecko
gecko0307@gmail.com

Что такое QR-код?

НАВЕРНЯКА вы заметили, что с определенного момента вам на глаза стали попадаться странные квадратики с каким-то непонятным кодом. Они встречаются на сайтах, в рекламе, на билбордах и на визитках. Что же это за код и как его расшифровать?

Эти квадратики – так называемый QR-код: матричный (двумерный) штрихкод, разработанный японской фирмой Denso-Wave в 1994 г. В этом штрихкоде кодируется разнообразная информация, состоящая из символов (включая кириллицу, цифры и специальные знаки). Информация, вообще говоря, любая: адрес сайта, телефон, электронная визитка, географические координаты и так далее. Один QR-код может содержать 7089 цифр или 4296 букв.

Аббревиатура QR производна от англ. quick response, что переводится как «быстрый отклик». Основное достоинство QR-кода — это легкое распознавание сканирующим оборудованием (в том числе и фотокамерой мобильного телефона), что дает возможность использования этой технологии в торговле и рекламе. Сегодня QR-коды больше всего распространены в Японии – стране, где штрих-коды пользовались такой большой популярностью, что объем информации, зашифрованной в них, вскоре перестал устраивать индустрию. Японцы начали экспериментировать с новыми способами графического кодирования информации.

В настоящее время QR-код широко распространен в странах Азии, постепенно развивается в Европе и Северной Америке. Наибольшее признание он получил среди пользователей мобильной связи – установив программу-распознаватель, абонент может моментально заносить в свой телефон текстовую информацию, добавлять контакты в адресную книгу, переходить по web-ссылкам, отправлять SMS-сообщения и т.д.

В Японии подобные коды наносятся практически на все товары, продающиеся в магазинах, их размещают в рекламных буклетах и справочниках. С помощью QR-кодов организовывают различные конкурсы и ролевые игры. Японцы размещают QR-коды даже на кладбищах – они содержат информацию об усопшем.

QR-коды активно используются и в туризме. Например, во Львове (Украина), местное объединение «Туристическое Движение Львова» разместило QR-коды более чем на 80

туристических объектах. Это позволяет индивидуальному туристу легко ориентироваться в городе, даже не зная украинского языка, так как QR-коды установлены на нескольких языках.

На данный момент есть достаточно широкий выбор программ для распознавания QR-кодов. Ниже приведен перечень наиболее популярных программ для мобильных платформ:

Android: Barcode Scanner, Barcode2file, QR Droid, NeoReader, ixMAT Scanner;
Apple iOS: QR Reader for iPhone, Barcode Scanner;
Windows Mobile: QuickMark, I-Nigma;
Symbian: Kaywa reader, Nokia barcode reader, I-Nigma, QuickMark, UpCode;
Java: Kaywa reader, I-Nigma, UpCode;
Maemo: mbarcode;
Bada: BeeTagg.

Теперь остался вопрос, каким образом вы можете сгенерировать QR-код для своих целей. Можно воспользоваться каким-нибудь бесплатным онлайновым сервисом — например, qrcoder.ru. Кроме того, существуют программы и скрипты, предназначенные для локальной генерации QR-кодов. К примеру, для языка Python такой скрипт можно найти на http://github.com/hcvst/pyqr.

Рекомендации по созданию QR-кодов

Чтобы все мобильные устройства корректно распознавали тип информации, зашифрованной в QR-коде, были введены специальные стандарты.

При кодировании адреса веб-страницы нужно обязательно указывать полный URL, включая протокол (HTTP://). URL-адреса нечувствительны к регистру, поэтому рекомендуется писать адрес в верхнем регистре (алгоритм это учитывает и кодирует эффективнее):

```
HTTP://FPSMAG.ZYMICHOST.COM
```

При кодировании адреса электронной почты нужно указывать префикс mailto:

```
mailto:gecko0307@gmail.com
```

При кодировании телефонного номера, нужно указывать префикс tel и указывать полный номер в международном формате, с кодом страны. Например, номер 212-555-1212 (США) следует представить как

```
tel:+12125551212
```

Для кодировании контактной информации, можно придерживаться форматов MECARD или BIZCARD. Например, визитка с текстом «Sean Owen, address "76 9th Avenue, 4th Floor, New York, NY 10011", phone number "212 555 1212", e-mail "srowen@example.com"», должна быть представлена как

```
MECARD:N:Owen,Sean;ADR:76 9th Avenue, 4th Floor,
New York, NY 10011;TEL:+12125551212;
EMAIL:srowen@example.com;;
```

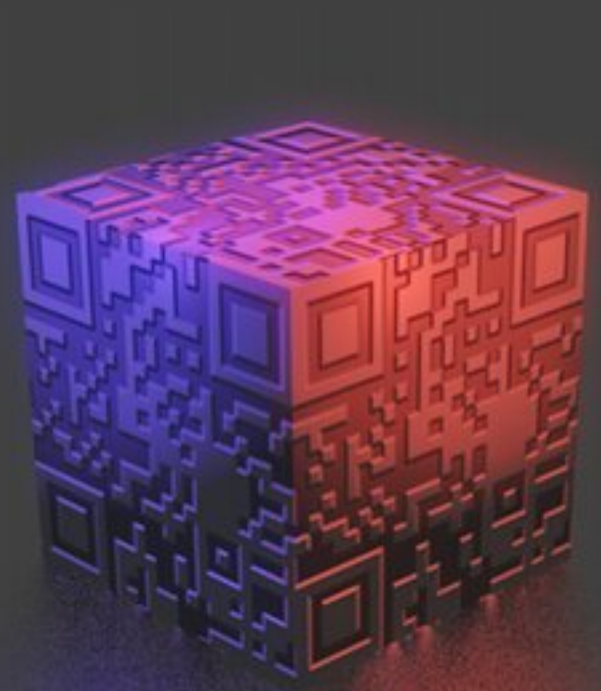
или

```
BIZCARD:N:Sean;X:Owen;A:76 9th Avenue, 4th Floor,
New York, NY 10011;B:+12125551212;
E:srowen@example.com;;
```

Географические координаты, например, офиса Google в Нью-Йорке (40.71872° северной широты и 73.98905° западной долготы, в 100 м. над уровнем моря) кодируются следующим образом:

```
geo:40.71872,-73.98905,100
```

Gecko
gecko0307@gmail.com



Модели освещения

Cook-Torrance

ОДНОЙ из наиболее точных и согласованных с физикой является модель освещения Кука-Торренса. Она также основана на модели поверхности, состоящей из микрограней, каждая из которых является идеальным зеркалом. Модель учитывает коэффициент Френеля и взаимозатенение микрограней.

В данной модели угол между нормалью к микрограни и нормалью ко всей поверхности подчиняется закону распределения Бэкмена (см. FPS №14 '11). Формула модели выглядит следующим образом:

$$I = \frac{DFG}{E \cdot N}$$

где I – интенсивность отраженного света, D – распределение Бекмана, F – коэффициент Френеля, G – коэффициент геометрического взаимозатенения микрограней, E – вектор наблюдателя, N – нормаль поверхности.

Значение G вычисляется по следующей формуле:

$$G = min\left(1, \frac{2(H \cdot N)(E \cdot N)}{E \cdot H}, \frac{2(H \cdot N)(L \cdot N)}{E \cdot H}\right)$$

где E – вектор наблюдателя, N – нормаль поверхности, L – вектор на источник света, H – половинный вектор.

Для вычисления коэффициента Френеля в графике реального времени обычно используют аппроксимацию Шлика.

Реализация на GLSL*

Вершинная программа:

```
varying vec4 V_eye;
varying vec4 L_eye;
varying vec4 N_eye;

void main(void)
{
    gl_Position = ftransform();

    V_eye = gl_ModelViewMatrix * gl_Vertex;
    L_eye = gl_LightSource[0].position - V_eye;
    N_eye = vec4(gl_NormalMatrix * gl_Normal, 1.0);

    V_eye = -V_eye;
}
```

** - предполагается, что в приложении уже включены и настроены соответствующие параметры OpenGL (позиция источника света, свойства материала и др.) Приведенная реализация в целях упрощения не учитывает интенсивность источника света. Исходный код предоставлен как общественное достояние (Public Domain) и может быть использован безо всяких ограничений.*

Фрагментная программа:

```
varying vec4 V_eye;
varying vec4 L_eye;
varying vec4 N_eye;

const float roughness = 1.0;

void main(void)
{
   vec4 Ca = gl_FrontMaterial.ambient;
   vec4 Cd = gl_FrontMaterial.diffuse;
   vec4 Cs = gl_FrontMaterial.specular;
   float Csh = gl_FrontMaterial.shininess;

   vec3 V = normalize(vec3(V_eye));
   vec3 L = normalize(vec3(L_eye));
   vec3 N = normalize(vec3(N_eye));
   vec3 H = normalize(L + V);

   float NL = max(0.0, dot(N, L));
   float NH = max(1.0e-7, dot(N, H));
   float NV = max(0.0, dot(N, V));
   float VH = max(0.0, dot(V, H));

   float diffuse = clamp(NL, 0.0, 1.0);

   float NH_sq = NH * NH;
   float NH_sq_r  = 1.0 / (NH_sq * roughness * roughness);
   float roughness_exp = (NH_sq - 1.0) * NH_sq_r;
   float beckmann = exp(roughness_exp)*NH_sq_r/(4.0*NH_sq);
```

```
    float geometric = min( 1.0, (2.0 * NH / VH) * min(NV, NL) );

    float fresnel = 1.0 / (1.0 + NV);

    float Rs = min(Csh, (beckmann * fresnel * geometric) / (NV * NL + 1.0e-7));

    gl_FragColor = Ca + NL * Rs * (Cd + Cs);
    gl_FragColor.a = 1.0;
}
```

Gecko
gecko0307@gmail.com

Toon shading на GLSL

ЭФФЕКТ toon shading (мультипликационное затенение) используется для придания графике «рисованного» стиля. Это, наверное, самый известный метод нефотореалистичного рендеринга. К сожалению, в играх он используется не очень часто, более успешно прижившись в области неинтерактивной анимации. Сейчас все почему-то стремятся к достижению полного фотореализма – забывая о том, что в графике важен не столько реализм, сколько художественный вкус.

Приведенная реализация toon shading дискретизирует освещенность в три компонента (тень, свет, блик), а также поддерживает отрисовку контуров.

Вершинная программа:

```
varying vec4 V_eye;
varying vec4 L_eye;
varying vec4 N_eye;
void main(void)
{
   gl_Position = ftransform();
   V_eye = gl_ModelViewMatrix * gl_Vertex;
   L_eye = gl_LightSource[0].position - V_eye;
   N_eye = vec4(gl_NormalMatrix * gl_Normal, 1.0);
   V_eye = -V_eye;
}
```

Вершинная программа:

```
varying vec4 V_eye;
varying vec4 L_eye;
varying vec4 N_eye;

const vec4 HighlightColour = vec4(0.8,0.8,1.0,1.0);
const vec4 MidColour = vec4(0.2,0.2,1.0,1.0);
const vec4 ShadowColour = vec4(0.0,0.0,0.5,1.0);

const float HighlightSize = 0.05;
const float ShadowSize = 0.1;
const float OutlineWidth = 0.3;

void main(void)
{
   vec3 V = normalize(vec3(V_eye));
   vec3 L = normalize(vec3(L_eye));
   vec3 N = normalize(vec3(N_eye));

   float lambert = dot(L,N);
   vec4 colour = MidColour;

   if (lambert>1-HighlightSize) colour = HighlightColour;
   if (lambert<ShadowSize) colour = ShadowColour;
   if (dot(N,V)<OutlineWidth) colour = vec4(0,0,0,1);

   gl_FragColor = colour;
   gl_FragColor.a = 1.0;
}
```

Это все!

Надеемся, номер вышел интересным. Если так, поддержите FPS! Отправляйте статьи, обзоры, интервью и прочее на любые темы, касающиеся игр, графики, звука, программирования и т.д. на gecko0307@gmail.com.