Независимый электронный журнал о разработке компьютерных игр

FPS

№12
2010

Blender
Шахматные компьютеры
Силой мысли...
Язык D
GLSL
Трассировщик лучей
Модели освещения
Reportlab

FPS

http://fps-magazine.narod.ru

http://twitter.com/fpsmag

Нас читают:

gcup.ru
Всё для начинающего и профессионального разработчика игр

make-games.ru
Портал создания игр

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

gdev.taba.ru
Сообщество разработчиков игр

www.mizzystic.ru
Крупнейший информационный Game Maker портал

xbobr.at.ua
Создание игр, программ, музыки

creat-game.ru
Все для создания игр

portalgame.net.ru
Игровой портал

gamesfpscreator.at.ua
Сайт для помещений игр и обсуждаловок

dapf.us
Форум для дизайнеров и программистов

gameshaker.ucoz.ru
Все для редактирования и создания игр.

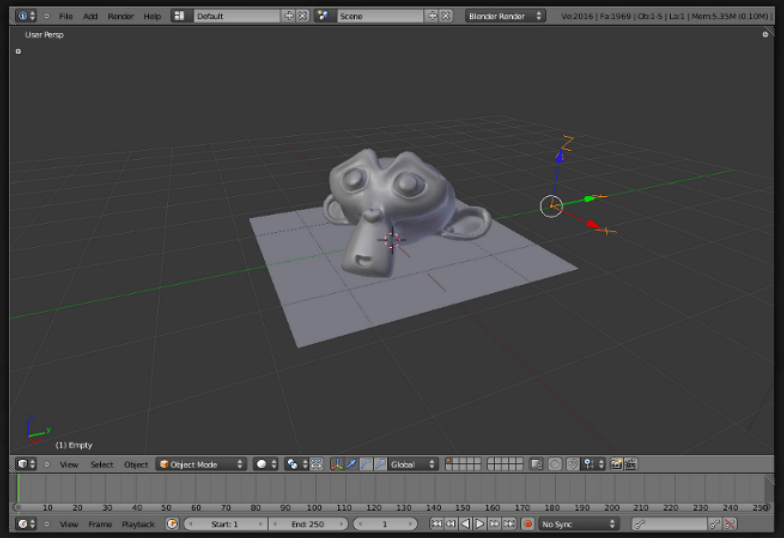
Главный редактор: Тимур Гафаров
Дизайн и верстка: Тимур Гафаров
По вопросам сотрудничества обращаться по адресу clocktower89@mail.ru или gecko0307@gmail.com.
Официальный сайт журнала: http://fps-magazine.narod.ru

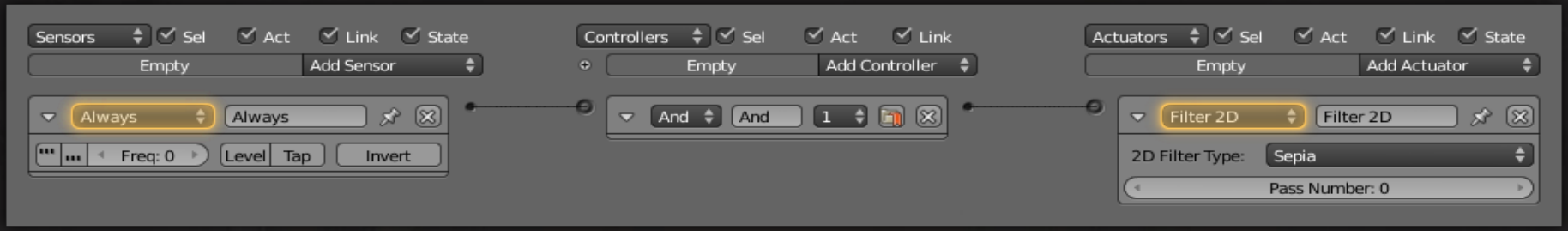
Blender 2.5

Экранные фильтры в Blender GE

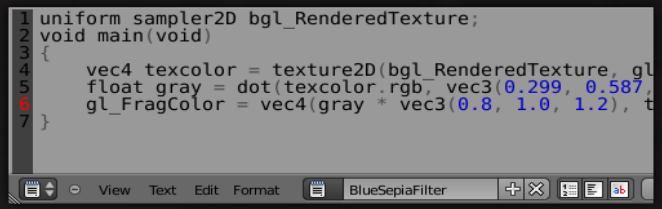
Многие современные игры используют пост-обработку — серию операций над изображением, полученным в результате рендеринга сцены. Это и модный нынче HDR, и glow, и размытие при движении, и многое другое. В игровом движке Blender (BGE) тоже имеется возможность применять такие эффекты — и, что самое приятное, создавать свои собственные.

Задействовать пост-обработку совсем несложно. Рекомендую тестировать ее на простой сцене, которая не займет много ресурсов для рендеринга — например, такой, как на скриншоте справа. За пост-обработку отвечает актуатор Filter2D, который нужно соединить с сенсором Always. Вы можете выбрать один из готовых фильтров: обращение цветов, сепия, оттенки серого, выделение края (Превитт, Лаплас, Собел), эрозия, дилация, повышение резкости, размытие (в том числе при движении). Убедитесь, что контроллер и актуатор добавлены в логику объекта, который всегда находится на сцене. Это может быть пустой объект (Empty).





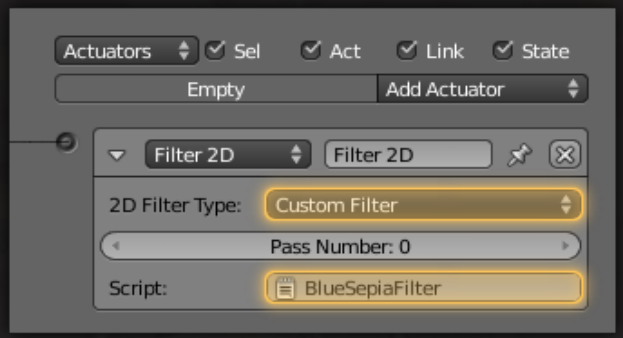
Чтобы создать собственный фильтр, перейдите в редактор текстов и создайте новый текст. Фильтры BGE — это не что иное, как фрагментные шейдеры на GLSL. Вот пример шейдера, реализующего фильтр «синей сепии»:

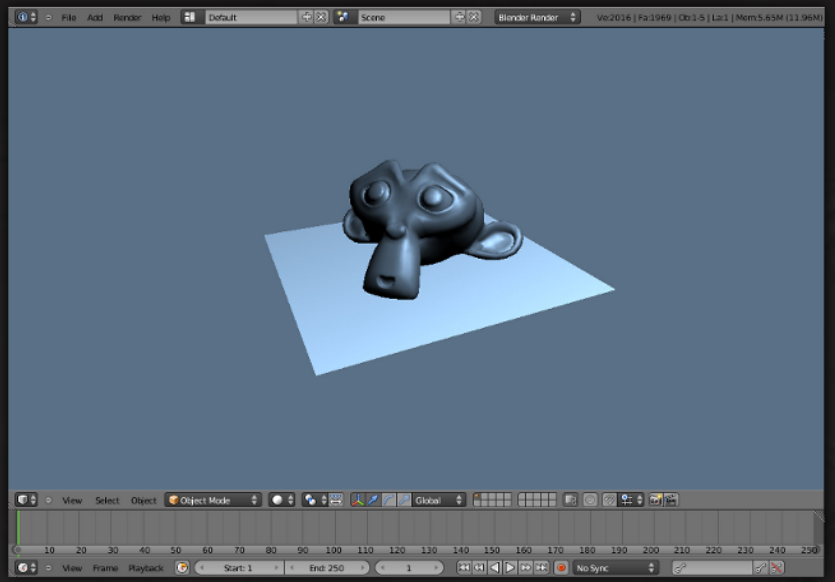



```
uniform sampler2D bgl_RenderedTexture;
void main(void)
{
   vec4 texcolor = texture2D(bgl_RenderedTexture, gl_TexCoord[0].st);
   float gray = dot(texcolor.rgb, vec3(0.299, 0.587, 0.114));
   gl_FragColor = vec4(gray * vec3(0.8, 1.0, 1.2), texcolor.a);
}
```

Обратите внимание на текстуру bgl_RenderedTexture — в нее передается отрендеренное изображение.

Чтобы применить созданный фильтр, вернитесь в редактор логики, выберите в списке фильтров Custom Filter и в поле Script укажите шейдер.





Python и интерфейс Blender

Одна из ключевых особенностей Blender 2.5 - полностью модифицируемый интерфейс. Абсолютно все панели, диалоги и прочие элементы интерфейса редактора теперь написаны на Python и поддаются редактированию. Кроме того, вы можете добавлять в программу совершенно новые инструменты. И для этого совсем необязательно даже править существующий код UI - создание кнопок и панелей осуществляется путем выполнения пользовательского скрипта прямо во время работы Blender!

В качестве примера рассмотрим создание простой панели в редакторе свойств (Properties > Object). Класс новой панели наследуется от bpy.types.Panel:

```
import bpy

class ObjectButtonsPanel(bpy.types.Panel):
    bl_space_type = "PROPERTIES"
    bl_region_type = "WINDOW"
    bl_context = "object"
    bl_label = "My new panel"
    def draw(self, context):
        layout = self.layout
        row = layout.row()
        row.label(text="Hello world!", icon='WORLD_DATA')
```

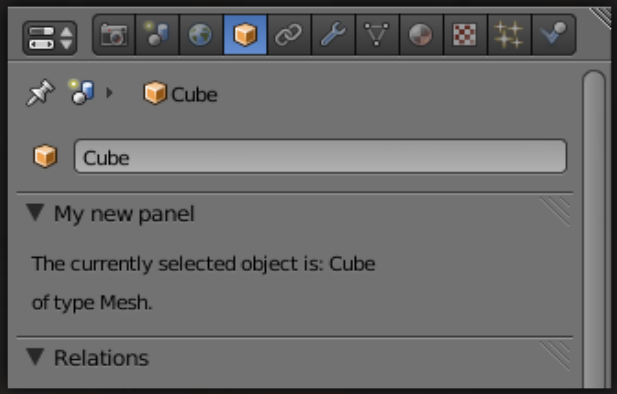
Теперь необходимо зарегистрировать созданный класс - и можно выполнять скрипт:

```
bpy.types.register(OBJECT_PT_hello)
```

Можно усложнить метод draw и добавить туда вместо традиционного приветствия "Hello, World!" нечто более информативное:

```
def draw(self, context):
    layout = self.layout
    ob = context.object
    type = ob.type.capitalize()
    col = layout.column()
    row = col.row()
    row.label(text="The currently selected object is: "+ob.name)
    row = col.row()
    row.label(text="of type "+type+".")
```

Теперь наша панель выводит имя и тип выделенного объекта!



Конференция Blender в Амстердаме

29-31 октября 2010 г. в Амстердаме прошла IX ежегодная конференция Blender Conference. На конференции, как обычно проведенной в здании театра-кафе «De Balie», выступили разработчики и пользователи Blender со всего мира, представив доклады по использованию Blender в анимационных проектах, в создании игр, в образовании и научных исследованиях. Прозвучали и доклады о интернет-маркетинге, робототехнике, сборке кластеров для рендеринга (renderfarm), новейших графических сетевых протоколах и CAD-системах. Не обошли стороной и дополненную реальность. Опытные художники и аниматоры провели ряд мастер-классов по моделированию (особое внимание было уделено моделированию и анимации людей и человекоподобных персонажей). Создатели «Sintel» поделились секретами эффектов освещения в фильме. Тон Роозендаал и Лука Бонавентура организовали «круглый стол» по обсуждению VFX в Blender, а также будущего открытого фильма (проект «Mango»).

На фестивале анимации Suzanne Award, традиционно являющемся частью конференции, участвовали талантливые авторы из Германии, Австрии, Финляндии, Польши, Венгрии, Испании, Великобритании, Аргентины, Бразилии, Индонезии и пр. Зрительским голосованием были выбраны победители в трех номинациях: «лучший короткометражный фильм» - «Lista» (Павел Лычковски, Академия изящных искусств, Варшава, Польша), «лучший художественный дизайн» - «John el Esquizofrenico» (Хуан Карлос Монтез, MoxStudios, Пуэрто-Рико), «лучшая анимация» - фильм без названия (Джарред Де Бир, Кейптаун, ЮАР).

Если отслеживать развитие Blender и нарастающий ажиотаж вокруг него, то тенденция в целом производит положительное впечатление. Blender — хороший пример удачного открытого проекта, который в достаточной мере поддерживается как программистами, так и пользователями. Свидетельств тому много: выход ветки 2.5x, успех открытых фильмов Blender Foundation, уважение со стороны профессиональной киноиндустрии и крупных образовательных учреждений, возросшая в последние годы популярность среди пользователей различных категорий и профилей. Кстати, не так давно Blender был выбран лучшей открытой графической программой по версии Packt Publishing, крупного интернет-магазина IT-литературы.

Шахматные компьютеры:
прошлое, настоящее, будущее

Многие думают, что компьютер научили играть в шахматы ради забавы: ну не с кем человеку сыграть в эту игру, так пусть хоть компьютер составит ему компанию. Но на самом деле эта задача зарождалась как область такого раздела информационных технологий, как искусственный интеллект.

История шахматных машин старше, чем история компьютеров. Идея создать машину, играющую в шахматы, датируется еще восемнадцатым веком. Около 1769 появился шахматный автомат «Механический турок». Он был предназначен для развлечения королевы Марии-Терезии. Машина действительно неплохо играла - внутри нее сидел сильный шахматист, который и делал ходы. Создание механических шахматных автоматов прекратилось с появлением цифровых компьютеров около 1950 г.

Первое исследование на тему шахматного программирования сделал в 1950 году американский математик Клод Шеннон. Он писал: «Хотя, возможно, это и не имеет никакого практического значения, сам вопрос представляется теоретически интересным, и будем надеяться, что решение этой задачи послужит толчком для решения других задач аналогичной природы и большего значения». Шеннон также отмечает теоретическое существование лучшего хода в шахматах и практическую невозможность его найти.

Один из первых шахматных алгоритмов разработал в 1951 г. Алан Тьюринг. С его помощью машина могла бы играть в шахматы, только в роли машины выступал сам изобретатель. Этот нонсенс даже получил название - «бумажная машина Тьюринга». Человеку надо было более получаса, чтобы сделать один ход. Алгоритм был довольно условный, и сохранилась даже запись партии, где «бумажная машина» Тьюринга проиграла одному из его коллег. За отсутствием доступа к компьютеру, программа ни разу не проверялась в работе.

Следующим шагом в развитии шахматного программирования стала разработка в ядерной лаборатории Лос-Аламоса в 1952 году на компьютере Maniac 1 (тактовая частота 11 кГц) шахматной программы для игры на доске 6x6, без участия слонов. Известно, что этот компьютер сыграл одну партию против сильного шахматиста, она продолжалась 10 часов и закончилась победой шахматиста. Еще одна партия была сыграна против девушки, которая недавно научилась играть в шахматы. Машина победила на 23-м ходу - для своего времени это было большое достижение.

Стоит также упомянуть труды Михаила Ботвинника, итоги которых, впрочем, были крайне сомнительны. Михаил Моисеевич утверждал, что напишет программу, которая сможет играть как гроссмейстер. И он верил, что можно заставить программу думать так, как это делает человек. Но даже после 20 лет разработки его программа так и не заиграла. По этому поводу кто-то остроумно заметил: «Ботвинник только думает, что знает, как он думает». Впрочем, это касается не только Михаила Моисеевича, но и человечества в целом: мы просто толком не знаем, как мы думаем, и поэтому вынуждены идти на обман, упрощать задачу, искать обходные пути. Машина же мыслить не может (по крайней мере сейчас). Но нет, скажете вы, а как же Deep Blue и Каспаров, проигравший компьютеру? На самом деле тут все немного иначе.

Во-первых, надо сказать, что в шахматы играет не компьютер, а программа. Программа строится на основе набора алгоритмов, называемых, как нетрудно догадаться, шахматными. Шахматные алгоритмы рассматривают ходы как игровое дерево. Теоретически, они должны оценивать все позиции, которые возникнут после всех возможных ходов, затем все возможные ходы после этих ходов и т.д. Каждый ход одного игрока называется «узел». Переборка ходов продолжается, пока программа не достигает максимальной глубины поиска или определяет, что достигнута конечная позиция (например мат или пат). И уже на основании оценки позиции выбирает оптимальную стратегию.

В каждой позиции количество возможных ходов игрока около 35. Для полного анализа четырех ходов (по два хода каждого игрока) нужно исследовать около полутора миллиона возможностей, для шести - почти два миллиарда. Анализ на 3 хода вперед - очень мало для хорошей игры.

Программисты пытаются по-разному ограничить массу ходов, который надо перебрать. Самым популярным является альфа-бета отсечение, в котором не рассматриваются позиции, имеющие меньшую оценку, чем уже оцененные.

Вторым распространенным методом является итерационное углубление. Сначала перебирается дерево игры до определенной глубины, после чего при помощи специальной оценочной функции выделяется несколько лучших ходов. Затем программа оценивает эти ходы применительно к большей глубине, чтобы узнать больше об их последствиях. Эта операция повторяется до наилучшего с точки зрения программы хода. Такой подход позволяет быстро отбросить немалый процент неперспективных вариантов игры. Например, не имеет смысла исследовать, что произойдет, когда обменять ферзя на пешку, если в позиции есть лучшие ходы.

Долгое время вопрос, сможет ли шахматная программа победить сильнейших шахматистов, оставался открытым. В 1968 г. Международный гроссмейстер Дэвид Леви поспорил, что ни один компьютер не сможет обыграть его в течение ближайших десяти лет. Он выиграл пари, победив в 1978 г. сильнейшую по тем временам шахматную программу Chess 4.7, но сознавал, что осталось не так уж много времени до того, когда компьютеры будут побеждать мировых чемпионов. В 1989 г. Леви проиграл программе Deep Thought. Но до появления Deep Blue программы все еще были значительно ниже уровня чемпиона мира.

Очень многие знают, что Deep Blue – это «тот самый компьютер, который обыграл Гарри Каспарова». И больше ничего. Отсутствие подробностей связано с маркетинговой политикой фирмы IBM. Отделу рекламы нужно, чтобы средний американец знал, что «IBM сделала непобедимый шахматный компьютер». Так что судить о Deep Blue (точнее, о Deep Blue II – последней машине из семейства) можно только по 6 сыгранным во втором матче с Каспаровым партиям, по редким публикациям в технических журналах, да по лекциям, которые устраивали разработчики.

Началом истории Deep Blue можно считать появление в 1989 г. шахматного компьютера Deep Thought, созданного представителями одного из американских вузов. Deep Thought играл в силу реального гроссмейстера, однако Каспарова он обыграть так и не смог. Впрочем, можно сказать, что для команды, работавшей над Deep Thought, все сложилось очень даже удачно: на их разработки обратила внимание компания IBM. После недолгих переговоров компьютерный чемпион (уже тогда Deep Thought стал чемпионом по шахматам среди компьютеров), так же как и его главный разработчик, переходит под крыло «голубого гиганта» («Big Blue» - так частенько называют IBM).

Вот тут-то и начинается самое интересное. Первым делом название Deep Thought поменяли на Deep Blue. Откуда взялось «Blue», думаю, уже понятно, а вот про «Deep» надо сказать еще пару слов. Большинство журналистов, далеких от компьютерного мира, переводят это как «глубокий», но это в корне неверно. На самом деле «Deep» указывает на мультипроцессорность программы. Сердцем нового шахматного суперкомпьютера стал сервер IBM RS/6000, в распоряжении которого имеется 32 процессора. К каждому из этих 32 подключили еще 8 специализированных шахматных процессоров. Таким образом, Deep Blue был укомплектован 256 дополнительными кремниевыми «мозгами». Процессоры были выполнены по 0,6-микронной технологии и работали со скоростью 2-2.5 млн. позиций в секунду.

Такое колоссальное быстродействие позволило разработчикам совместить сложную оценочную функцию и значительную глубину перебора. Deep Blue углублялся на 12 полуходов, после чего начинал просматривать наиболее выгодные позиции. Программа была написана на обычном C, и не очень отличалась от традиционных параллельных шахматных программ. Наиболее существенное отличие – при написании программы было известно, что система обладает огромной скоростью, соответственно, можно особо не экономить. При написании шахматной программы всегда велико желание какие-то варианты рассмотреть поглубже («углубиться»), но всегда приходится помнить, что если мы углубимся здесь, то останется меньше времени на рассмотр других вариантов. Соответственно, обычно приходится искать компромисс между более глубоким рассмотрением «тактических» вариантов (в которых только и работают классические углубления) и «тихих» вариантов (которые с точки зрения человека тоже могут быть тактическими, но только ходы не попадают под слишком узкое машинное определение тактических). Здесь же можно рассматривать более длинные варианты, не заботясь что другим времени не достанется – ресурсов хватит на всех. Поэтому «глубина углубления» могла составлять до 30 полуходов (напомним, что полуход — это один ход белых или черных).

Оценочная функция была намертво вшита в аппаратное обеспечение, что сделало ее очень быстрой. Утверждается, что она соответствовала примерно 40000 командам обычного процессора. При расчете выгодности той или иной позиции функция оперировала с 8000 коэффициентами, которые загружались из специальной базы при запуске компьютера.

И все же шахматный монстр снова проиграл Каспарову. Такой результат не удовлетворил специалистов из IBM, и через год состоялся матч-реванш. Процессоры были серьёзно переработаны, когда стало ясно, что в предыдущей версии отсутствуют многие факторы, которые необходимы для игр на таком уровне. И тут Каспаров уже проиграл. Справедливости ради надо заметить, что разрыв в очках был просто смешной – 3, 5:2, 5 в пользу Deep Blue.

Позже IBM обвинили, что во время партий фирма использовала человека-шахматиста, чтобы увеличить стратегическую силу компьютера. В 2003 году был снят документальный фильм, в котором исследовались эти упреки - «Матч окончен: Каспаров и машина» (Game Over: Kasparov and the machine). В нем утверждалось, что сильно раскрученная победа Deep Blue подстроена для увеличения рыночной стоимости IBM. Частично эти упреки были оправданными. Правила позволяли разработчикам изменять программу между играми. Deep Blue был изменен между партиями для лучшего понимания машиной стиля игры Каспарова, помогая избежать ловушки в эндшпиле, в которую дважды попадал искусственный интеллект.

В первых шахматных программах ходы, ведущие к потере какой-либо фигуры, считались очень нежелательными, и оценочная функция присваивала им кране низкий рейтинг. Однако в последней партии последней игры Каспарова против Deep Blue машина сделала крайне неожиданный ход: отдала своего коня, чтобы получить определенные позиционные преимущества. А через несколько лет другая программа (Deep Junior) умудрилась отдать слона за пешку, что позволило ей провести очень эффектную атаку. Кстати, Junior тоже играл партию с Каспаровым.

Подобные неожиданности наводят общественность на мысли о том, что машина действительно может думать, однако это просто отлично написанная оценочная функция, способная оперировать огромной базой данных, содержащей матчи между гроссмейстерами: программа считает ход, когда-либо сделанный гроссмейстером, очень выгодным и добавляет ему рейтинг.

Существуют также базы данных, которые используются лишь в завершающей стадии игры. Эти «эндшпильные таблицы», еще одно творение Кена Томпсона (главного проектировщика операционной системы UNIX), содержат все возможные позиции с шестью или менее фигурами на доске (уже начали появляться и семифигурные позиции). С помощью этих оракулов были обнаружены позиции, требующие для победного завершения игры более 200 точных ходов! О таком уровне сложности раньше не приходилось и мечтать, и он просто недостижим для человека.

Игра в эндшпиле долго была заметной слабостью шахматных программ, так как глубина поиска была недостаточной. Таким образом, даже программы, которые играли в силу мастера не в состоянии выиграть в эндшпильних позициях, где даже шахматист средней силы мог форсировать выигрыш. Но результаты компьютерного анализа иногда удивляли людей. В 1977 г. шахматная машина Томпсона Belle, используя эндшпильные базы данных король + ладья против короля + ферзь, была способна свести вничью теоретически проигрышные эндшпили против титулованных шахматистов.

После своей победы над Каспаровым Deep Blue был разобран и больше не играл. Оно и понятно: в случае поражения IBM потеряет имя «компании, создавшей непобедимый компьютер», что будет ей совсем не на руку. Но инициативу перехватила немецкая фирма Chessbase: у гроссмейстеров появились новые серьезные конкуренты.

Имея все большую вычислительную мощность, шахматные программы, запущенные на персональных компьютерах, стали достигать уровня лучших шахматистов. В 1998 г. программа Rebel 10 победила Вишванатана Ананда, который тогда занимал второе место в мире. Однако не все партии игрались со стандартным временным контролем. Из восьми партий матча, четыре играли с блиц-контролем (пять минут плюс пять секунд за каждый ход), которые Rebel выиграл со счетом 3-1. Еще две игры были с полу-блиц контролем (пятнадцать минут на каждого), которые программа также выиграла (1,5-1). Наконец, две последние партии были сыграны со стандартным турнирным временным контролем (два часа на 40 ходов и час на остальные партии) и тут выиграл уже Ананд со счетом 0,5-1,5. К тому времени в быстрых партиях компьютеры играли лучше людей, но при классическом временном контроле преимущество было уже не так велико.

2000 г. шахматные программы Deep Junior и Deep Fritz смогли свести в ничью матчи против предыдущих мировых чемпионов Гарри Каспарова и Владимира Крамника. В инструкции к Deep Fritz 8 было сказано, что при использовании достаточно быстрого компьютера она обыгрывает 99,99% людей.

В октябре 2002 Владимир Крамник и Deep Fritz соревновались в матче из восьми партий в Бахрейне. Матч закончился вничью. Крамник выиграл вторую и третью партии, используя традиционную противокомпьютерную тактику - играл осторожно, имея целью долгосрочное преимущество, которое компьютер не может увидеть в своем дереве поиска. И все Fritz выиграл пятую партию после грубой ошибки Крамника. Шестую партию много турнирных комментаторов назвали очень увлекательной. Крамник, имея лучшую позицию в начале миттельшпиля, попытался пожертвовать фигурой, чтобы создать сильную тактическую атаку (такая стратегия - очень рискованная против компьютеров). Fritz нашел сильную защиту, и эта атака значительно ухудшила позицию Крамника. Крамник сдал игру, не надеясь на победу. Однако последующий анализ показал, что Fritz вряд ли смог бы довести игру до своего выигрыша. Последние две партии закончились вничью.

В январе 2003 г. Гарри Каспаров играл против программы Junior в Нью-Йорке. Матч закончился со счетом 3-3. В ноябре 2003 Каспаров провел игру с X3D Fritz с результатом 2-2. В 2005 г. Hydra, специальный шахматный компьютер с 64 процессорами, победил Майкла Адамса, шахматиста, который в то время был на седьмом месте в мире по рейтингу ЭЛО, в матче из шести партий со счетом 5,5-0,5 (хотя подготовка Адамса была намного ниже, чем у Каспарова в 2002 году). Некоторые комментаторы верили, что Hydra наконец получит несомненное преимущество над лучшими шахматистами. В ноябре-декабре 2006 г. Владимир Крамник играл с Deep Fritz. Матч закончился со счетом 2-4.

Другим известным достижением компьютерных систем стал трехкруговой матч-турнир трех известных гроссмейстеров против трех известных шахматных программ осенью 2004 г. Играли Руслан Пономарев (чемпион мира 2003 года), чемпион мира среди юношей 12-14 лет Сергей Карякин и болгарин Веселин Топалов, третий в мировом шахматном рейтинге. Компьютеры представляли действующий чемпион мира Deep Junior 9, Deep Fritz и Hydra. Компьютеры выиграли с перевесом в три очка.

Нельзя утверждать, что это самые сильные на сегодня шахматные программы: в матче не участвовали серебряный и бронзовый призеры последнего компьютерного чемпионата мира по шахматам — немецкая Schredder 8.5 и Diep3d 7.5 соответственно (российские программы, увы, в этом чемпионате не участвовали). Большинство компьютеров, использовавшихся программами-участниками этого чемпионата, работали на 64-разрядных микропроцессорах AMD — Opteron и Athlon64. Вероятно, это связано с их высокой целочисленной производительностью. Остальные программы работали на Intel Pentium 4. Все три призера чемпионата были распараллелены и работали на четырехъядерных системах архитектуры SMP на базе Opteron. Тенденция распараллеливания шахматных программ ныне должна еще более усилиться: закону Мура, по всей видимости, приходит конец, а потому других путей повышения производительности, кроме распараллеливания, не остается.

Интересные данные о распараллеливании современных шахматных программ представил Винсент Дипэвен, автор Diep3d. По его словам, эффективные шахматные алгоритмы используют общую таблицу хеширования (своеобразный кэш в оперативной памяти, чем он больше, тем, теоретически, лучше; на практике Diep3d использует кэш емкостью несколько сотен мегабайт). Поиск лучшего хода осуществляется итерационно — сначала на один полуход, затем на два и т. д. При просмотре позиций на один полуход глубже требуемое время возрастает в несколько раз, т. е. время расчета растет экспоненциально с глубиной анализа. Учитывая, что из-за перестановок в порядке ходов одни и те же позиции могут возникать в разных ветвях дерева поиска, процессоры при распараллеливании сразу должны проверять — не была ли уже рассмотрена получившаяся позиция.

Отказ от общей таблицы хеширования очень сильно ослабляет работу шахматной программы, что не может быть скомпенсировано распараллеливанием даже на большом числе процессоров. Поэтому эффективны компьютерные архитектуры с общим полем памяти и кластеры, если их межсоединение поддерживает распределенную общую память. Это справедливо, например, для межсоединения Quadrics.

При таком распараллеливании возникают частые обмены небольшими порциями данных между процессорами, т. е. важна задержка межсоединения многопроцессорной системы кластера. Типичной операцией, по данным Дипэвена, является чтение 4-64 байт из памяти удаленного узла кластера.

Знаменитый Deep Blue обрабатывал, как известно, 100 миллионов позиций в секунду, но из-за ограничений при работе с хеш-таблицей осуществлял поиск на глубину 12 полуходов. По оценке Дипэвена, глубина поиска современной шахматной программы на современном компьютере составляет до 20 полуходов (если не считать эндшпиля). Сегодня сильнейшие программы оценивают от 150 тысяч позиций в секунду (однопроцессорная версия Diep) до 1,5 миллионов позиций в секунду (Fritz) при работе на одном процессоре.

Последователем Deep Blue с аппаратной точки зрения можно считать программу Hydra. Она работает на кластере с 16 двухпроцессорными узлами, оснащенными процессорами Intel Xeon и программируемыми процессорами, реализованными в виде плат FPGA (именно аппаратной поддержкой «шахматной игры» отличалась, как известно, и Deep Blue).

В будущем, как сообщил Дипэвен, нас ожидает знакомство с настоящим шахматным монстром — компьютером Sheikh, который также будет работать с FPGA и включать 1024 процессора. Создание такого суперкомпьютера обходится очень дорого — его финансирует один из принцев Арабских Эмиратов.

А что же искусственный интеллект? Да, игра в шахматы — классическая область приложения методов ИИ, в которой применяется много эвристических алгоритмов для организации эффективного поиска наилучшего хода. Это реализуется, большей частью, с помощью метода «грубой силы» («brute force»), используя высокую производительность компьютеров и распараллеливание при эффективной организации переборной задачи. Впрочем, в широких массах определенно ждали несколько иной реализации искусственного интеллекта — с процессом принятия решения, напоминающим человеческий подход, с самообучением и т.д.

Кстати, любительские шахматные компьютеры сейчас доступны по очень низкой цене. Появилось много программ с открытыми исходными кодами, в частности Crafty, Fruit и GNU Chess, которые могут обыграть многих профессиональных шахматистов. На первом месте в компьютерных рейтинг-листах (таких, как CEGT, CCRL, SCCT) сейчас находится движок Rybka.

Gecko
gecko0307@gmail.com

Силой мысли…
Устройства ввода завтрашнего дня

Разговоры о принципиально новых устройствах ввода, радикально отличающихся от мышки и клавиатуры, возникают регулярно – например, одно время весьма популярна была тема грядущего управления голосом: компьютеры вот-вот должны были научиться распознавать не то что отдельные команды, а и вовсе связную и слитную речь. Однако прошли годы, а голосовое управление так и не стало не то что массовым, но и вообще хоть сколь-нибудь заметным явлением, упершись в технологический барьер: чтобы расшифровывать человеческую речь в реальном времени, с учетом отсутствия пауз между словами, разницы в произношении и интонации, непостоянного темпа и прочих особенностей, требуется недюжинной силы искусственный интеллект.

На фоне этого идея управления компьютером с помощью силы мысли вообще кажется относящейся разве что к фантастическим романам или, максимум, отдельным научно-медицинским лабораториям. При словах «управление с помощью силы мысли» в голове возникает картинка шлема, обильно утыканного электродами, от которого в компьютер уходит толстый пучок проводов…

Впрочем, утверждение «мысль материальна» на самом деле совсем недалеко от истины. Наверняка любой из читателей видел электроэнцефалограмму мозга (ЭЭГ), но откуда именно берутся эти волнообразные графики, знают далеко не все. Между тем, все очень просто: по нейронным связям мозга текут электрические токи, а мозг при этом испускает слабые электрические импульсы, которые давно уже научились обнаруживать и фиксировать. Эти импульсы представляют собой разночастотные колебания электрического потенциала. Для нас прежде всего важен тот факт, что мысль действительно можно превратить в сигнал для осуществления того или иного действия. Хотя бы на уровне примитивного «да/нет».

Более подробные исследования выявили у человека несколько групп волн, различающихся частотным диапазоном и возникающих в разных состояниях работы мозга. Так, различают следующие группы ритмических колебаний:

- **Альфа-ритмы**. Это колебания потенциала в диапазоне частот 8-13 Гц. Они возникают, когда мы отдыхаем, расслабляемся и как будто бы ни о чем не думаем. Также эти волны возникают, когда человек находится в состоянии медитации. Как только активность мозга увеличивается, альфа-ритмы сменяются бета-ритмами.
- **Бета-ритмы**. Это колебания потенциала в диапазоне частот от 14 Гц и выше. Перьевые самописцы, применяющиеся при снятии ЭЭГ, имеют предел фиксирования 35 Гц, поэтому часто можно слышать, что бета-ритмы ограничиваются именно этой частотой, хотя это не совсем так. Эти волны возникают во время физической и умственной активности, когда вы сосредоточены и напряжены. Блокируются бета-ритмы при тактильном раздражении, а также при движении конечностей в противоположных направлениях.
- Также различают **гамма-ритмы** (колебания потенциала с частотой выше 35 Гц, являющиеся фактически теми же бета-ритмами), **дельта-ритмы** (колебания потенциала с частотой 1-3,5 Гц) и **тета-ритмы** (колебания потенциала с частотой 4-7 Гц). Два последних типа волн возникают во время сна.

Идея управления машинами одной только силой мысли – не новая. Она появилась у писателей-фантастов много лет назад и постепенно, с ускорением в последние годы, приближается к своей массовой реализации. Возможность контролировать работу компьютеров или более сложных машин силой мысли уже давно является предметом активного изучения учеными разных специальностей.

Уже известны примеры управляемых силой мысли «печатных машинок»; систем управления курсором компьютера и даже игра Mindball, в которую можно играть с помощью мысленных команд. Над задачей управления техникой силой мысли работает много ученых, в том числе исследователи из Федеральной политехнической школы Лозанны (Ecole polytechnique federale de Lausanne (EPFL), Швейцария). Здесь создали мобильное кресло, управляемое мысленными командами.

Чтобы коляска выполняла нужные хозяину действия, использованы посредники – электроэнцефалограф и компьютер. Первый – через шапочку с электродами воспринимает электрические сигналы мозга, второй, снабженный искусственным интеллектом, - понимает эти сигналы и преобразует в команды для электромоторов коляски. В итоге человек, сидящий в коляске, может управлять её движением, будучи полностью неподвижным (парализованным). Для поворота, например, налево ему надо просто мысленно пошевелить левой рукой.

Испытуемому требуется несколько часов, чтобы приспособиться к управлению системой. Она способна адаптироваться к пассажиру и учитывать особенности его мозговой активности. Для безопасности пассажира коляска имеет собственные «глаза» – две маленькие камеры. Благодаря им она сможет распознавать препятствия, дверные проемы и других людей.

Отметим, что разработка подобных систем на основе интерфейса мозг–компьютер (Brain–Computer Interface – BCI) ведется в разных странах, в том числе и в России. Ученые Южного федерального университета (ЮФУ) выиграли грант Агентства по науке и инновациям РФ в 17 миллионов рублей на разработку систем мысленного управления компьютером, как сообщил руководитель проекта, профессор Борис Владимирский. Он отметил, что специальная аппаратура будет снимать электрические сигналы, идущие от мозга через шлем с электродами, а затем через усилитель эти сигналы будут направляться в компьютер. По словам Владимирского, создание такой системы также улучшит качество жизни инвалидов: «Человек, который не может говорить, сможет указывать на картинку «дать воды». А те, кто не могут двигаться, смогут отправлять электронные письма или даже управлять инвалидными колясками при помощи мысли». По его словам, для управления нужно будет научиться правильно формулировать мысленно команду, например, силой мысли двигать курсор мыши на экране.

Помимо улучшения качества жизни инвалидов, применений у систем мысленного управления может быть множество – от мысленного ввода текста в компьютер, управления освещением или кондиционером в доме, до управления сложной техникой, вроде самолета. Уже сейчас в продажу поступили такие устройства, как OCZ NIA (Neural Impulse Actuator). Этот приборчик регистрирует нейроимпульсы и транслирует их в команды управления компьютером. И при этом – никаких шлемов, лишь аккуратный обруч, надеваемый на голову. Системные требования – один свободный USB-порт.

Основной источник сигналов для NIA – мимические и глазные мышцы, точнее, управляющие ими нервные импульсы; кроме того, устройство «чувствует» альфа- и бета-ритмы головного мозга. Датчики NIA расположены на резиновом обруче, надевающемся на голову и затягивающемся с помощью пары шнурков на затылке.

С точки зрения системы, NIA выглядит обычным HID-устройством – к тому же классу относятся клавиатуры и мыши. Однако для нормальной работы с NIA требуется установить также специальное ПО, позволяющее откалибровать датчики и привязать сигналы с них к перемещению джойстиков или нажатию кнопок.

В качестве практического упражнения NIA предлагает игру, появившуюся в электронном виде задолго до персональных компьютеров – пинг-понг. На экране – две ракетки и летающий между ними мячик; правой ракеткой управляет компьютер, левую должны двигать вы – с помощью OCZ NIA. Эта игра из всего арсенала возможностей NIA задействует лишь один вертикальный джойстик, управляемый сигналами от лицевых мимических мышц. При расслабленных мышцах ваша ракетка будет находиться внизу экрана, стоит же вам наморщить лоб, как она подскочит вверх. В какую-то секунду ваш мозг сам собой абстрагируется от команд «поднять бровь!» и «опустить бровь!» и переходит на команды «переместить ракетку вон туда».

Конечно, NIA не читает мысли – но ощущение возникает ровно такое. Вы управляете не мышцами на лбу, вы управляете ракеткой на экране, сложив при этом руки на груди, удобно откинувшись в кресле и с усмешкой поглядывая на окружающих.

К сожалению, в более сложных играх, в которых задействованы сразу все доступные NIA типы нейросигналов и определено множество событий, получить такую же управляемость намного сложнее. На данный момент NIA недостаточно хорошо разделяет нейросигналы разных типов, поэтому в одних случаях требуются хитрости вроде ввода искусственных задержек, а в других – кропотливая подстройка чувствительности датчиков.

Более того, работа NIA сильно зависит не только от вашего упорства в тренировках, но и от типа вашей кожи и черт лица. Датчики NIA плохо работают на сухой коже, требуя использования увлажняющего крема, а людям со сколь-нибудь заметно выступающими надбровными дугами придётся, скорее всего, позабыть о горизонтальных джойстиках – расположить обруч так, чтобы он одновременно и фиксировал перемещения глаз, и обеспечивал надёжный контакт центрального датчика с кожей, будет трудно.

Таким образом, OCZ NIA крайне интересна в качестве экспериментального устройства, но вот практическая её ценность в играх находится под большим вопросом – как минимум, успех или неудача очень сильно зависят от конкретного человека. Намного сильнее, чем с любым другим типом манипуляторов. Разумеется, для людей, которым в силу тех или иных физических недостатков трудно пользоваться обычными мышками и клавиатурами, NIA может оказаться огромным подспорьем, но на этом возможности серьёзного её использования пока что и ограничиваются.

Gecko
gecko0307@gmail.com

Язык D

Перспективы использования D в разработке игр

C++ на сегодняшний день — основной ЯП практически во всех сферах информационных технологий. Популярность, которую он приобрел, обоснована вполне объективными причинами — трудно отрицать достоинства этого языка, объединившего высокую производительность C и мощь современных парадигм программирования. C++ сыграл значительную роль в индустрии программного обеспечения, за годы существования по нему накоплена громадная кодовая база, масса литературы и документации. Знание C++, безусловно, необходимо. Из всех адекватных современным тенденциям языков C++ является наиболее доступным (есть бесплатные и открытые компиляторы, отладчики, IDE, библиотеки и т.д.) и, что актуально, удобным при портировании проектов на другие платформы. Да, есть академисты, приверженцы Pascal, Fortran, Lisp — точно так же, как есть любители классической музыки или поэзии. В подобных языках есть своя гармония, они обладают строгой, математической структурой, педагогической наглядностью. Однако жизнь (рынок, бизнес, прогресс — нужное подчеркнуть) диктует свои законы. Для программиста-прагматика гораздо важнее продуктивность, нежели соответствие академическим канонам.

Именно в таких условиях родились C, C++, C#, Java и другие представители «семейства фигурных скобок», стоящих в авангарде практики разработки ПО. Стареющий C постепенно приобрел репутацию языка хакеров («второй ассемблер»). C# и Java, напротив, предназначены для корпоративной сферы и быстрой разработки различных сложных систем на основе готовых решений. В эту же группу можно отнести популярные у нас Delphi и Visual Basic, максимально упрощающие создание графических приложений с типовыми компонентами. Мультипарадигменный C++ можно назвать золотой серединой. Возможно, поэтому его и облюбовали разработчики игр — им не нужна ни излишняя низкоуровневость, ни нагромождение шаблонов и заготовок. Главное — производительность и поддержка мультимедийных технологий.

Однако в своем стремлении сохранить совместимость с традиционным C и одновременно не отстать от времени, «плюсы» накопили серьезное количество семантических недоразумений, синтаксических казусов, всевозможных тонкостей, исключений из правил и прочего балласта, мешающего языку развиваться. Не говоря уже о том, что все эти премудрости крайне сложны для понимания непосвященными, из-за них складывается впечатление, что C++ «не любит» своих пользователей. Во-первых, отчего на C++ так сложно писать стабильный безошибочный код? Ответ прост: «неправильные» конструкции на нем гораздо более просты и очевидны, чем «правильные» — тем более, что компиляторы C++ неприхотливы и слишком многое оставляют на совести программиста. Во-вторых, C++ создавался в эпоху C, а с тех пор многое изменилось, в том числе — и приоритеты в разработке ПО. Компьютеры стали достаточно мощными, чтобы нам не приходилось экономить каждый лишний такт процессора или байт памяти. Удобство и скорость разработки стали важнее оптимизации кода. И если программисту хочется работать с ассоциативными массивами или сборщиком мусора — дайте ему такую возможность. Незачем доказывать, что можно обойтись и без них (между прочим, люди когда-то обходились и без электричества). А поскольку подобные средства начали появляться сравнительно недавно (каких-то двадцать лет назад никому бы и в голову не могло прийти такое: как, доверить высвобождение памяти какому-то постороннему коду?! Ересь!), то консервативные программисты до сих пор смотрят на них, как на вздорную выдумку каких-то чудаков. Да, к C++ можно прикрутить собственный сборщик мусора. Да, стандартные библиотеки C++ реализуют то, чего нет в самом языке. И, надо сказать, делают это совсем неплохо. Но программисты все равно переходят на Python — устав «изобретать велосипеды» и пытаться мало-мальски эффективно реализовать то, чего не умеет ни сам язык, ни STL. А это уже повод задуматься. Прикладной программист должен решать свои задачи на необходимом для этого уровне абстракции, а не заниматься «грязной работой». C++, к сожалению, не способствует этому — чего не скажешь о интерпретируемых языках, специально созданных с таким девизом.

Я, конечно, не призываю всех переходить на Python. Во всяком случае, не сейчас. Если нынешние тенденции сохранятся, когда-нибудь придет время скриптовых языков — все клиентские прикладные программы будут писаться на них. Но современным программистам, в особенности игровым, все-таки необходима компиляция в машинный код. Так возможен ли компромисс? Думаю, да.

Таковым компромиссом может стать D — молодой, но перспективный язык, вобравший все лучшее от C++, и при этом лишенный его недостатков. D очень практичен: в этом языке уже предусмотрено все, что в обычных условиях приходится додумывать самостоятельно. D направлен на удовлетворение потребностей современного программирования: с учетом продуктивности, портируемости, совместной разработки, автоматического документирования, реализации компиляторов и т.д. Львиная доля функциональности STL здесь является частью самого языка, а стандартные библиотеки D — так вообще чуть ли не панацея на все случаи жизни. Правда, разработчикам D пришлось пожертвовать совместимостью с C и C++. Это не значит, что D будет непривычен для «плюсистов» (на самом деле, как раз наоборот — этот язык в значительной степени ориентирован на перешедших с C++ или Java). Скорее, это затрудняет портирование уже существующего кода. Но только затрудняет, а не делает невозможным: программы на D могут компоноваться с готовыми C-библиотеками, в том числе — стандартными (при желании можно использовать функции C вроде printf).

D позиционируется как системный язык, он компилируется в сильно оптимизированный машинный код. D поддерживает все основные стили (парадигмы) программирования: императивный, функциональный, объектно-ориентированный и обобщенный. По синтаксису D относится к «семейству фигурных скобок» и, на первый взгляд, здорово напоминает C++.

По традиции, в качестве первого примера привожу программу «Hello, World!»:

```
import std.stdio;
void main()
{
   writeln("Hello, World!");
}
```

Что интересно, в D нет привычного деления исходного кода на файлы заголовков-прототипов (*.h) и реализации (*.cpp). Здесь основная единица программы — модуль (*.d). В D не нужно объявлять функции по два раза, чтобы иметь к ним доступ из других файлов. Достаточно написать модуль, а затем импортировать его везде, где нужно оператором import — компилятор сам все поймет. Модули, расположенные в одной директории, объединяются в пакеты (например, модуль engine.math должен находится в папке /engine). Похожий принцип используется в Python и других модных языках. Благодаря такому подходу, кстати, ощутимо сокращается время сборки: компилятору не нужно каждый раз анализировать огромные объемы кода, достаточно импортировать таблицу имен.

Это, в свою очередь, ставит под вопрос необходимость наличия препроцессора. Есть мнение, что препроцессор C слишком мощный и опасный. В D его попросту нет: все, для чего он, в основном, используется (компиляция различных версий кода, макросы, псевдонимы, константы и пр.) в D успешно интегрировано в сам язык. Хотя многие считают отказ от препроцессора чересчур опрометчивым шагом. Поначалу его действительно будет не хватать, но со временем вы привыкнете. Например, условный переход для выбора платформозависимого кода в D оформляется следующим образом:

```
version (Windows)
{
   // Код для Windows
}
version (Linux)
{
   // Код для Linux
}
```

Возможно, программисты, ценящие чистоту языков, сочтут D слишком «пестрым». Функции в стиле Pascal, вроде writeln, у многих вызывают улыбку. Кроме того, в D есть конструкции сверхвысокого уровня, роднящие его со скриптовыми языками, но, в то же время, не забыт встроенный ассемблер и арифметика указателей. В определенных ситуациях низкоуровневые средства бывают незаменимы — так зачем же от них отказываться? D — язык прагматиков.

Не знаю, как другим, а мне в C++ всегда бывает трудновато при работе с памятью. Выделить динамическую память легко, а вот когда и как ее безопасно высвободить? Если посмотреть статистику работы различных запущенных процессов, особенно игр, то часто наблюдается медленное, но верное повышение потребления памяти. Будучи далеким от системного программирования, я не берусь компетентно судить о причинах этого явления. Однако очевидно, что прикладной программист контролирует только свой код и не может отвечать за работу системы. Для меня, например, работа ядра и то, как оно управляет памятью — темный лес. Поэтому со своей стороны следует делать все возможное, чтобы программа не пожирала память, засоряя ее неиспользуемыми объектами. Это многократно облегчается при наличии сборщика мусора — специального компонента рантайма, который периодически высвобождает неиспользуемую память. «Плюсисты» и «сишники», как правило, скептически относятся к данной фиче. Но практика показывает, что программы с автоматической сборкой мусора более эффективны.

Чем дальше вы углубляетесь в D, тем привлекательнее он становится. Например, работа с массивами здесь куда проще и очевидней, чем в «плюсах» (тот факт, что C++ не поддерживает массивы динамической длины, для меня в свое время был крупным разочарованием). В D есть четыре типа массивов: операции с указателями в стиле C (надо думать, оставлены, в первую очередь, для совместимости с библиотеками C), статические массивы, динамические массивы и ассоциативные массивы. Для последних трех доступно извлечение полезной информации — такой, как длина и размер в байтах.

Вот несколько самых ярких примеров возможностей при работе с массивами:

```
// Объявляем динамический массив:
int[] array = new int[10];

// Более изящный способ:
auto array = new int[10];

// Узнаем длину массива:
size_t length = array.length;

// Для доступа к последнему элементу массива,
// вместо неуклюжего array[array.length – 1]
// можно использовать такую конструкцию:
array[$ - 1] = 45;

// Можно изменить значения сразу всех элементов:
array[] = 21;

// ...или в определенном диапазоне:
array[5..8] = 21;

// Шедевр элегантности и выразительности –
// доступ к половине массива:
writeln( array[$ / 2 .. $] );

// Однотипные массивы можно соединять (конкатенировать)
// при помощи оператора '~' (тильда).
// Поскольку строки в D – это частный случай массивов,
// на них тоже распространяется эта возможность.
int[] a = [0, 10, 20];
int[] b = [3, 7, 1];
int[] c = a ~ c;
string s = "hello" ~ "world";

// Оператор конкатенирования можно использовать
// также для добавления нового элемента
// в динамический массив:
auto a = [87, 40, 10];
a ~= 42;

// Копировать массив до смешного просто:
int[] a = new int[128];
int[] b = new int[128];
b[] = -1;
a[] = b[];
```

```d
// Можно оперировать массивами как числами,
// выполняя операцию для всех элементов одновременно
// (а в C пришлось бы организовывать цикл):
auto a = [0.5, -0.5, 1.5, 2.0];
auto b = [3.5, 5.5, 3.0, -1.0];
auto c = new float[4];
c[] = (a[] + b[] + 1.0) / 2.0;

// Кстати, о циклах. В D есть чрезвычайно удобный
// оператор цикла foreach:
auto array = new int[128];
foreach (index, value; array)
{
   array[index] = value + 1;
}

// Многомерные массивы — головная боль начинающих
// «сишников» и «плюсистов». В D это не проблема.
// Кстати, можете визуально представить себе
// четырехмерный массив? Я — нет :)
float[5][5][5][5] array = 10;
array[4][1][2][0] = 3;
writeln(array[4][1][2][0]);

// Ассоциативные массивы D позволяют использовать
// в качестве индекса любой тип, будь то строка,
// другой массив или даже класс:
int[string] b;
b["hello"] = 3;

// Можно удалять элементы:
b.remove("hello");

// Оператор 'in' возвращает указатель на элемент,
// если он присутствует в ассоциативном массиве,
// или null в противном случае:
int* p;
p = ("hello" in b);
if (p != null) *p = 10;
```

Для D уже существует множество готовых библиотек, а также привязок к популярным существующим. Игровых движков немного, но есть специальные фреймворки для быстрой разработки мультимедийных приложений. Например, Derelict — джентльменский набор геймдевелопера, коллекция привязок к OpenGL, OpenAL, SDL, ODE, FMOD, DevIL, FreeType, Ogg/Vorbis, PortAudio и т.д. Все библиотеки компонуются динамически в рантайме — это облегчает отладку и портирование программ, а также использование различных версий библиотек. При желании Derelict можно расширить, добавив привязки к другим C-библиотекам. На D можно разрабатывать графические программы — для этого используются привязки и абстракции к различным тулкитам (Windows API, Gtk, Qt, wxWidgets). Для создания расширяемых приложений можно использовать привязки к скриптовым языкам (Python, Lua и др.)

Есть две версии языка — D1 и D2. D2 современнее и мощнее, но зато под D1 больше примеров кода, библиотек, документации и т.д. Стандартных библиотек тоже две. Для D1 это Tango, для D2 — Phobos. Они несовместимы между собой, что порой создает определенные трудности. Поэтому стоит определиться с самого начала: какую версию D выбрать для работы. Второй вопрос: выбор компилятора. Разработкой D на данный момент занимается организация Digital Mars, и лучшим компилятором пока считается ее продукт — DMD (есть версии для Windows, Linux, Mac OS X, FreeBSD). Существуют альтернативы — LDC (LLVM D Compiler), GDC (GNU D Compiler), D.NET (для платформы .NET/Mono).

D может стать прекрасным выбором для игрового программиста. Но к нему нужно прийти осознанно, тщательно взвесив все «за» и «против». На сайте проекта есть краткий обзор основных принципов D, в котором приведен остроумный список тех, для кого язык предназначен и кому противопоказан («Who D is For» и «Who D is Not For»). Попробуйте найти в нем себя :) Я, например, попадаю под категорию «тех, кого вдохновляет выразительность и мощь C++, но расстраивает необходимость уделять много внимания работе с памятью».

У D есть будущее, а у программистов есть право выбора. Выбрав этот язык сейчас, мы сами можем сформировать его будущее. Так почему бы не стать частью истории?

Сайт Digital Mars: www.digitalmars.com
Крупнейший хостинг D-проектов: www.dsource.org
Русское сообщество программистов на D: dprogramming.ru

Трассировщик лучей на D

Искушенные в трехмерной графике читатели наверняка знакомы с понятием «raytracer» - «трассировщик лучей». Это общее обозначение для класса программ, моделирующих излучение, отражение и преломление света. Современные инструменты для работы с трехмерной графикой широко используют трассировщики при оффлайн-рендеринге, а в последние годы, с появлением мощных графических процессоров, трассировка лучей стала применяться и в графике реального времени.

Принцип работы трассировщика сводится к очень простому алгоритму: для каждой точки растрового изображения проводится луч в некоем известном направлении (это может быть перпендикуляр к плоскости экрана или, для достижения перспективной проекции, вектор от позиции наблюдателя). Затем происходит перебор всех объектов-примитивов трехмерной сцены. Трассировщик проверяет луч на пересечение с примитивом и получает координаты точки пересечения. Если на сцене присутствуют источники света, проводятся дополнительные лучи от полученной точки к источникам света. На основании полученной информации (а также заранее определенных констант), вычисляется цвет пикселя в данной точке. Для упрощенного (biased) рендеринга могут быть использованы формулы Ламберта, Фонга, Блинна и т.д. В случае бескомпромиссного (unbiased) рендеринга используются методы глобального освещения.

Написание трассировщика — очень интересное и полезное упражнение. Оно позволяет лучше понять работу движков рендеринга, расширить знания математики и физики. В конце концов, нет ничего приятнее, чем видеть картинку, отрисованную твоей собственной программой!

В этой статье я рассматриваю простой вариант трассировщика, который поддерживает только один вид примитивов — сферы. Сцена описывается тремя массивами — для камер, примитивов и источников света (источники света — точечные). В качестве языка используется D, но код несложно будет переписать на C++.

Приведенная реализация использует модули, которые не рассматриваются в статье (в частности, векторную алгебру и буферы изображений) — их можно скачать отдельно с сайта журнала:

```
import std.stdio, std.math;
import image.color, image.buffer;
import math.vector, math.util;
```

Луч описывается двумя точками (начальной и конечной), а также расстоянием до пересечения (t):

```
class ray
{
   vector3f p0;
   vector3f p1;
   float t;
   this(vector3f begin = new vector3f(),
        vector3f end = new vector3f())
   {
      p0 = begin;
      p1 = end;
   }
}
```

Объекты сцены (примитивы, камеры, источники света) удобно наследовать из общего класса:

```
class sceneObject
{
   vector3f position;
   string type;
}
```

Класс камеры включает метод создания луча для заданной точки:

```
class sceneCamera: sceneObject
{
   float zoom;
   this() {
      zoom = 120.0f;
      type = "camera";
      position = new vector3f();
   }
   ray getRay(buffer buf, float x, float y) {
      return new ray(position, position +
         new vector3f(x-buf.getWidth/2,
                      y-buf.getHeight/2,zoom) );
   }
}

class sceneLight: sceneObject
{
   int lightType = 0;
   vector3f color;
   this() {
      type = "light";
      color = new vector3f(1.0,1.0,1.0);
      energy = 1.0f;
      position = new vector3f();
   }
}
```

Абстрактный класс примитивов содержит методы для обнаружения пересечения с лучом и вычисления нормали поверхности в заданной точке:

```
class sceneShape: sceneObject
{
   this() {
      type = "shape";
   }
   vector3f intersect(ref ray R) {
      return null;
   }
   vector3f normal(vector3f pos) {
      return null;
   }
}
```

Класс сферы наследует от класса примитивов:

```
class shapeSphere: sceneShape
{
   float radius;
   this(float r = 1.0f) {
      super();
      radius = r;
   }
   vector3f intersect(ref ray R) {
      vector3f dir = R.p1 - R.p0;
      dir.normalize();
      vector3f dist = position - R.p0;
      float B = dot(dist,dir);
      float D = radius * radius - dot(dist,dist) + B*B;
      if (D < 0.0f) return null;
      float t0 = B - sqrt(D);
      float t1 = B + sqrt(D);
      if (t0 > 0.0f) {
         R.t = t0;
         return (R.p0 + dir * t0);
      }
```

```
      if (t1 > 0.0f) {
         R.t = t1;
         return (R.p0 + dir * t1);
      }
      return null;
   }
   vector3f normal(vector3f pos) {
      return (pos - position)*(1/radius);
   }
}
```

Наконец, класс сцены, содержащий ассоциативные массивы для объектов сцены и функцию, совершающую трассировку (ее мы рассмотрим поподробнее):

```
class scene
{
   sceneCamera[string] cameras;
   sceneShape[string] shapes;
   sceneLight[string] lights;
   this()
   {
      cameras["camera_default"] = new sceneCamera();
   }

void render(buffer buf, sceneCamera cam = null)
{
```

В функцию опционально передается камера, которую следует использовать для трассировки. Если камера не указана явно, используется камера по умолчанию, созданная ранее в конструкторе:

```
sceneCamera currentCamera;
if (cam) currentCamera = cam;
else currentCamera = cameras["camera_default"];
```

В целях упрощения реализация не учитывает отдельные материалы для примитивов — вместо этого задается стандартный общий для всех материал из четырех привычных компонентов:

```
auto materialAmbient = new vector3f(0.05,0.05,0.05);
auto materialDiffuse = new vector3f(0.6,0.6,0.6);
auto materialSpecular = new vector3f(1.0,1.0,1.0);
float materialShininess = 32.0;
```

Перебираем все пиксели заданного буфера двойным циклом:

```
for (int x=0; x<buf.getWidth; x++)
for (int y=0; y<buf.getHeight; y++)
{
ray currentRay = null;
vector3f intersection = null;
float t = 1000;
sceneShape currentShape = null;
```

Перебираем все примитивы сцены:

```
foreach (shp; shapes)
{
```

Для каждого примитива находим точку пересечения с лучом:

```
auto newray = currentCamera.getRay(buf,x,y);
auto inters = shp.intersect(newray);
```

Отсекаем примитивы, находящиеся слишком далеко от камеры:

```
if (newray.t < t) {
   t = newray.t;
   currentShape = shp;
   currentRay = newray;
   intersection = inters;
}
```

Если было обнаружено пересечение, вычисляем цвет точки:

```
if (intersection)
{
```

Находим векторы наблюдателя, нормали и отражения:

```
vector3f v, n, l, r;
v = currentRay.p1 - currentRay.p0; v.normalize();
n = currentShape.normal(intersection);
r = v - 2.0 * n * dot(n,v);
```

Для вычисления конечного освещения точки перебираем все источники света:

```
float diffuse, specular;
vector3f finalColor = materialAmbient;
foreach (lght; lights)
{
   l = lght.position - intersection; l.normalize();
   diffuse = clamp!float(dot(l,n),0.0,1.0);
   specular = pow(fmax(dot(r,l),0.0),materialShininess);
   finalColor = finalColor + lght.color *
      (materialDiffuse*diffuse + materialSpecular*specular);
}
```

В приведенной реализации используется модель Ламберт + Фонг. В полноценных трассировщиках, как правило, моделей гораздо больше — они либо встроены в программу и являются свойством материала (как, например, во внутреннем трассировщике Blender), либо реализуются при помощи специальных языков, похожих на языки описания шейдеров. Примеры: RenderMan Shading Language, NVIDIA Gelato, а также развивающийся Open Shading Language.

Записываем полученный цвет в обрабатываемый пиксель:

```
buf.setPixel(x,y,vecToColor(finalColor));
} } } } }
```

Осталось только создать сцену и отрендерить ее. Все трассировщики имеют собственный формат сцены — обычно на основе XML (например, Toxic) или скриптов (PovRay). Мы же просто создадим все нужные объекты вручную в главной функции:

```
void main(string[] args)
{
   auto bgcol = vecToColor(new vector3f(0.0f,0.0f,0.0f),1.0f);
   buffer image = new buffer(320, 240, 3, bgcol);

   scene Scene = new scene();

   auto sphere1 = new shapeSphere(20.0);
   sphere1.position = new vector3f(-30.0,0.0,50.0);
   Scene.shapes["sphere1"] = sphere1;

   auto sphere2 = new shapeSphere(15.0);
   sphere2.position = new vector3f(20.0,0.0,50.0);
   Scene.shapes["sphere2"] = sphere2;

   auto light1 = new sceneLight();
   light1.position = new vector3f(70.0f, 0.0f, -50.0f);
   light1.color = new vector3f(1.0,1.0,1.0);
   Scene.lights["light1"] = light1;

   auto light2 = new sceneLight();
   light2.position = new vector3f(-70.0f, 40.0f, 0.0f);
   light2.color = new vector3f(1.0,0.0,0.0);
   Scene.lights["light2"] = light2;

   Scene.render(image);
   image.write(BMP,"image.bmp");
}
```

GLSL

Общие сведения для ознакомления

GLSL (OpenGL Shading Language) — мощный высокоуровневый язык для написания шейдеров под OpenGL. GLSL поддерживает вершинные и фрагментные (пиксельные) шейдеры — программы, которые выполняются на вершинном и фрагментном процессорах.

Вершинный процессор — это программируемый модуль GPU, оперирующий атрибутами вершин — такими, как позиция, цвет, текстурные координаты и пр. Вершинный процессор за раз может обрабатывать данные только об одной вершине и не может заменить собой графические операции, которые требуют наличия топологических данных.

Геометрический процессор имеет доступ к данным о геометрии обрабатываемых примитивов (отрезки, треугольники, квады и т.д.). Программа для геометрического процессора — геометрический шейдер — не входит в спецификацию GLSL; это мультивендорное расширение, доступное на видеокартах nVidia GeForce 8 и выше.

Фрагментный процессор обрабатывает каждый фрагмент изображения (пиксель на экране), прошедший все включённые тесты (stencil, alpha, depth и т.д.). Результатом его работы является значение итогового цвета фрагмента или же отмена его отрисовки. Благодаря появлению фрагментных шейдеров, появилась возможность реализации различных моделей освещения и всевозможных графических эффектов, недоступных ранее в трехмерной графике реального времени.

Синтаксис GLSL близок к C. Язык расширен за счет дополнительных типов данных (векторы и матрицы) и набора стандартных функций, работающих с этими типами. Объявление переменных аналогично C — они могут объявляться в любом месте программы, и возможность доступа к ним зависит от области видимости, в которой они были объявлены.

Типы данных

void - функции, не возвращающие значения, должны быть описаны как возвращающие void. Ключевое слово void больше нигде не может быть использовано;
bool - логический тип, принимающий значения true (истина) и false (ложь);
int - целочисленное значение со знаком. Точность может варьироваться в зависимости от оборудования и реализации GLSL для него (гарантируется точность в 16 бит);
float - вещественное значение;
vec2 - двумерный вектор вещественных значений;
vec3 - трехмерный вектор вещественных значений;
vec4 - четырехмерный вектор вещественных значений;
bvec2, bvec3, bvec4 - вектор логических значений;
ivec2, ivec3, ivec4 - вектор целочисленных значений;
hvec2, hvec3, hvec4 - вектор половинных значений (half);
mat2x2, mat2 - матрица вещественных значений 2x2;
mat2x3 - матрица вещественных значений 2x3;
mat2x4 - матрица вещественных значений 2x4;
mat3x2 - матрица вещественных значений 3x2;
mat3x3, mat3 - Матрица вещественных значений 3x3;
mat3x4 - матрица вещественных значений 3x4;
mat4x2 - матрица вещественных значений 4x2;
mat4x3 - матрица вещественных значений 4x3;
mat4x4, mat4 - матрица вещественных значений 4x4;
sampler1D - одномерная текстура;
sampler2D - двумерная текстура;
sampler3D - трехмерная текстура;
samplerCube - кубическая текстурная карта;
sampler1DShadow - одномерная карта глубин;
sampler2DShadow - двумерная карта глубин.

В GLSL введены также специальные спецификаторы переменных:

uniform - внешние данные для передачи в шейдер (только для чтения). Должны объявляться в глобальной области видимости.
varying - промежуточные данные для передачи из вершинного шейдера во фрагментный (для чтения/записи). Varying-переменные должны быть объявлены в обоих шейдерах в глобальной области видимости.
attribute - атрибуты вершины (только для чтения). Доступны только для вершинного шейдера.
const - константа (только для чтения).

Примеры объявления переменных:

```
int i, j = 10;
int k = 0xFF;
varying float a, b;
float c = 1.5, d = 1e-5;
uniform vec2 v;
vec4 w = vec4(1.0, 1.0, 0.0, 0.5);
mat3 mv;
```

Стандартные переменные

В вершинном шейдере определены три стандартные глобальные переменные:

vec4 gl_Position - однородные координаты вершины. Запись обязательна;
float gl_PointSize - размер точки для растеризации. Запись необязательна;
vec4 gl_ClipVertex - координаты для использования с плоскостями отсечения. Запись необязательна.

Любому шейдеру доступен ряд целочисленных констант, определяющих ограничения, накладываемые как самим графическим ускорителем, так и драйвером для него.

gl_MaxLights - Наибольшее допустимое количество источников света (>=8);
gl_MaxClipPlanes - Наибольшее допустимое количество плоскостей отсечения (>=6);
gl_MaxTextureUnits - Количество текстурных блоков (>=2);
gl_MaxTextureCoords - Наибольшее допустимое количество текстурных координат (>=2);
gl_MaxVertexAttribs - Количество вершинных атрибутов, включая стандартные (>=16);
gl_MaxVertexUniformComponents - Наибольшее допустимое количество uniform-переменных в вершинном шейдере (>=512);
gl_MaxVaryingFloats - Наибольшее допустимое количество varying-переменных (>=32);
gl_MaxVertexTextureImageUnits - (>=0);
gl_MaxCombinedTextureImageUnits - (>=2);
gl_MaxFragmentUniformComponents - Наибольшее допустимое количество uniform-переменных во фрагментном шейдере (>=64);
gl_MaxDrawBuffers - Количество доступных буферов для отрисовки (>=1).

Стандартные аттрибуты для вершинного шейдера

vec4 gl_Color - основной цвет вершины;
vec4 gl_SecondaryColor - дополнительный цвет вершины;
vec4 gl_Normal - вектор нормали вершины;
vec4 gl_Vertex - позиция вершины в объектном пространстве;
vec4 gl_MultiTexCoord0,
gl_MultiTexCoord1,
gl_MultiTexCoord2,
gl_MultiTexCoord3,
gl_MultiTexCoord4,
gl_MultiTexCoord5,
gl_MultiTexCoord6,
gl_MultiTexCoord7 - текстурные координаты для [0, 7] текстурных блоков;
float gl_FogCoord - величина затуманивания вершины.

Стандартные varying-переменные:

vec4 gl_FrontColor - основной цвет для лицевой стороны граней;
vec4 gl_BackColor - основной цвет для обратной стороны граней;
vec4 gl_FrontColor - вторичный цвет для лицевой стороны граней;
vec4 gl_BackSecondaryColor - вторичный цвет для обратной стороны граней;
vec4 gl_TexCoord[...] - массив текстурных координат. Индекс массива — номер текстурного блока;
float gl_FogFragCoord - координаты тумана.

В GLSL есть циклы - **for**, **while**, **do**..**while**; операторы перехода - **continue**, **break**, **return**, **discard** (запрещение вывода фрагмента во фрагментном шейдере).

Функции

Программа на GLSL должна иметь главную входную функцию void main(), с которой начинается выполнение. Вы можете определять и вызывать свои собственные функции точно так же, как и в C. Есть и стандартные функции, являющиеся частью языка:

radiants(deg) - перевод значения угла из градусов в радианы;
degrees(rad) - перевод значения угла из радиан в градусы;
sin(x) - синус угла (задается в радианах);
cos(x) - косинус угла (задается в радианах);
tan(x) - тангенс угла (задается в радианах);
pow(x, y) - x в степени y;
exp(x) - e в степени x;
exp2(x) - 2e в степени x;
log(x) - натуральный логарифм для x;
log2(x) - log2 x;
sqrt(x) - квадратный корень из x;
inversesqrt(x) - 1/sqrt(x): большинство графических процессоров могут выполнять эту операцию за один такт;
abs(x) - модуль x;
min(a, b) - минимальное из двух величин;
max(a, b) - максимальное из двух величин;

Параметры функции можно объявлять со спецификаторами доступа:

in - создается локальная копия параметра для чтения-записи (спецификатор по умолчанию), если присутствует const, то запись запрещена;
inout - локальная копия не создается, чтение/запись происходит с внешней переменной (как по ссылке);
out - так же, как и inout, только чтение параметра запрещено.

```
vec4 myFunc1(const in vec4 v1) { ... };
void myFunc2(in vec4 v1, out float f1) { ... };
```

Структуры и массивы

GLSL поддерживает структуры в стиле C: структура может быть типом переменной, которым можно пользоваться так же, как и встроенными типами.

```
struct Light
{
   vec3  pos;
   float intensity;
   vec4  color;
};
Light light;
```

Массивы объявляются также аналогично C:

```
float freq[4];
Light light[3];
```

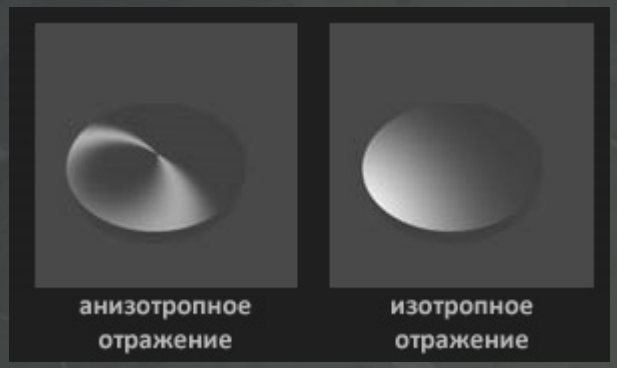
Для использования программ на GLSL требуется поддержка следующих расширений:

GL_ARB_shader_objects;
GL_ARB_shading_language_100;
GL_ARB_vertex_shader;
GL_ARB_fragment_shader.

Модели освещения

Блики по Фонгу

Благодаря зеркальному отражению на блестящих предметах появляются световые блики. Зеркальные отражения могут быть изотропными (isotropic) и анизотропными (anisotropic). Анизотропные отражения вытянуты в направлении перпендикулярно углублению, в отличие от равномерных невытянутых изотропных отражений. В простых моделях освещения обычно пользуются эмпирической изотропной моделью Бу Тонга Фонга. Она не соответствует точному физическому описанию отражения света, но в большинстве случаев позволяет достичь приемлемых реалистичных результатов. Формула Фонга основана на простом наблюдении: блестящие поверхности дают маленькие и резкие блики, в то время как матовые — большие и размытые. Из-за того, что зеркально отраженный свет сфокусирован вдоль вектора отражения, блики при движении наблюдателя тоже перемещаются.



Модель Фонга имеет вид:

$$I = I_t k_s \cos^m \alpha$$

где I — интенсивность отраженного света, I_t — интенсивность точечного источника света, k_s — коэффициент зеркального отражения, α — угол между направлением света и вектором отражения, m — параметр отражаемости поверхности (*shininess*).

Косинус угла между вектором отражения и вектором наблюдателя соответствует скалярному произведению этих двух векторов (при условии, что они нормализованы, т.е. имеют единичную длину):

$$\cos^m \alpha = (r,v)^m$$

где v — вектор наблюдателя, r — вектор, получающийся при идеальном отражении вектора направления света относительно нормали поверхности:

$$r = 2n(n,l) - l$$

где n — нормаль, l — вектор направления на источник света.

На практике модель Фонга используется совместно с каким-либо методом диффузного рассеивания, например по Ламберту (*см. «FPS» №11/2010*). Общее уравнение освещения одним и более источниками света будет выглядеть следующим образом (приведенная ниже реализация модели Фонга учитывает только один источник света):

$$I = I_a k_a + \sum_{t \,\epsilon\, lights} l_t (k_d (n,l_t) + k_s (r_t,v)^m)$$

Реализация на GLSL*

Вершинная программа:

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

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

Фрагментная программа:

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

vec3 reflect (vec3 N, vec3 L)
{
   return 2.0 * N * dot(N, L) - L;
}

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 R = reflect ( N, L );

   float diffuse = clamp (dot (N, L), 0.0, 1.0 );
   float specular = pow (max (dot (R, V), 0.0 ), Csh );
   gl_FragColor = Ca + (Cd*diffuse) + (Cs*specular);
   gl_FragColor.a = 1.0;
}
```

** - 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.*

Репортаж о Reportlab

Система подготовки документов на Python

На сегодняшний день стандартом печати и сетевого документооборота де-факто является PDF (Portable Document Format). Последние спецификации этого формата поддерживают сложное форматирование, векторную графику, встраиваемые шрифты, различные алгоритмы сжатия, элементы контроля, шифрование и многое другое. PDF разработан компанией Adobe, но свободен от патентных отчислений (royalty free) — разработчикам программ просмотра, создания и редактирования PDF не нужно платить Adobe за право использования этого формата. PDF широко распространен и отлично поддерживается на всех основных платформах.

Для создания документов в формате PDF обычно используется виртуальный принтер (Adobe Acrobat Distiller или его аналоги). Такой способ годится для подготовки докуметов к физической печати, но имеет определенные недостатки. Печать в PDF не всегда гарантирует WYSIWYG, является платформозависимой и не слишком хорошо подходит для генерации оптимизированных документов для распространения по сети (отсутствует сжатие изображений, вследствие чего «отпечатанные» на виртуальном принтере документы могут занимать слишком много места, а также ссылки, метаданные и другое специфичное для электронного документа содержимое). Поэтому часто становится необходим прямой экспорт данных в PDF.

Экспорт в PDF — стандартная функция практически всех систем компьютерной верстки. Популярные офисные пакеты также постепенно осваивают это направление. Последние версии MS Office поддерживают сохранение документов как PDF. Его свободный аналог OpenOffice.org всегда славился отличной поддержкой PDF. Менее известный Corel WordPerfect Office так вообще умеет напрямую редактировать PDF (в OpenOffice такую возможность предоставляет сторонний плагин).

Куда менее распространено невизуальное создание PDF при помощи языков программирования. Это, скорее всего, связано с тем, что дизайнеров и верстальщиков никакие силы не смогут согнать с редакторов вроде Quark или InDesign и изучать какое-то программирование :) А ведь работу в таких программах очень трудно автоматизировать, многие действия приходится повторять десятки, а то и сотни раз, какие-то операции затруднены особенностями интерфейса и логики программ, что-то невозможно сделать мышью, не вводя точные данные, что-то бывает несовместимо с PDF или имеет какие-то другие ограничения, а что-то и вовсе не поддерживается. Более того, если вы захотите передать кому-либо исходник проекта (например, для редактирования и пересборки), это неизбежно приведет к различным казусам. Выяснится, что на другой системе установлена не та версия редактора, отсутствуют необходимые настройки или другие данные (шрифтов, стандартных изображений и др.). В общем, «радостей» здесь может быть много.

Немногим известно, что существуют мощные системы «программирования» PDF, по возможностям не уступающие WYSIWYG-редакторам. Лишаясь всего лишь возможности предпросмотра результата, вы приобретаете:

● высочающую гибкость при создании документа. Вы можете определить его структуру и иерархию, разбить на смысловые блоки, быстро собирать документ или его части с различными опциями, создавать шаблоны, стили оформления и многое другое. Возможности ограничиваются только языком и вашим талантом проектировщика;

● практически полную автоматизацию. Это и нумерация страниц, и добавление колонтитулов, и подсветка синтаксиса, и предпроцессинг, и специализированная XML-разметка. Компьютер берет на себя всю рутинную работу;

● высокую точность. Можете забыть о мучениях с неточной визуальной подгонкой элементов — это можно сделать, введя пару цифр;

● интеграцию с любыми другими программными компонентами. Например, как вам такое: рендеринг трехмерной графики при помощи OpenGL (или программного растеризатора/трассировщика) и вставка полученных изображений в документ? Кстати, в OpenOffice.org нечто подобное появилось совсем недавно, да и то — с ограничениями;

● портируемость. Для сборки документа не нужно ставить громоздкие пакеты, достаточно иметь транслятор с соответствующего языка. Следовательно, собрать документ можно на любой системе, где такой транслятор реализован;

Одной из лучших таких систем является Reportlab (для языка Python). Изначально предназначенная для генерации отчетов, Reportlab тем не менее отлично справляется и с более сложными задачами. Разработчики Reportlab отмечают, что она хорошо подходит для динамической генерации PDF на веб-серверах, для публикации статистики и содержимого баз данных, а также в качестве системы сборки научных публикаций. В арсенале Reportlab — поддержка Юникода, всех популярных форматов изображений и шрифтов, графических примитивов, векторной графики, таблиц, диаграмм и т.д.

В основе Reportlab лежит концепция холста (canvas). На нем можно последовательно рисовать различные элементы (текст, фигуры, изображения и прочее). Перед отрисовкой элемента задаются цвет контура и заливки, шрифт, параметры трансформации. Для создания сложных документов можно использовать более сложные абстракции. Такие, как, например, абзац (paragraph) — блок текста, к которому применяется форматирование. Абзац в Reportlab поддерживает XML-разметку, чем-то напоминающую HTML. Другой концептуальный слой Reportlab — Platypus (Page Layout and Typography Using Scripts) — предназначен для высокой автоматизации при подготовке многостраничных PDF. Здесь введены понятия документа, шаблона, фреймов и плавающих элементов.

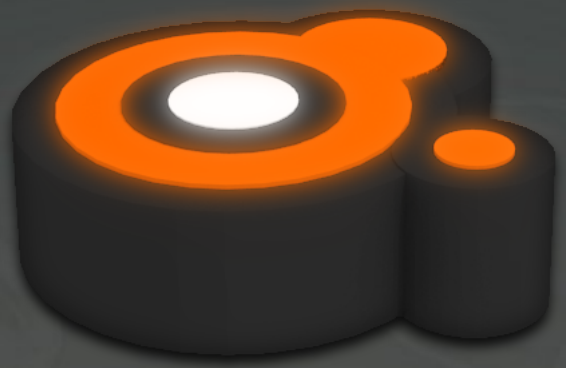
Несмотря на обилие всевозможных управляющих структур и инструментов форматирования, Reportlab все же имеет и свои «узкие места». Это, в первую очередь, поддержка SVG. При помощи сторонних модулей можно сконвертировать SVG в векторный формат Reportlab (и, причем, совершенно прозрачно для программиста), но из-за некоторых ограничений не все элементы SVG будут обработаны корректно. Например, в Reportlab нет градиентной заливки (важный недостаток) — это не позволяет вставлять SVG без предварительной оптимизации. Во-вторых, у Reportlab нет собственной разметки математических выражений. Общепризнанным веб-стандартом для этих целей считается MathML, но использование формул MathML в Reportlab сопряжено с определенными трудностями. При верстке нашего журнала (номер «FPS», который вы сейчас читаете, был сверстан целиком в Reportlab) был применен «конвеер»: MathML > SVGMath > SVGLib > Reportlab (благо, все компоненты этой цепочки тоже написаны на Python). SVGMath принимает текст в разметке MathML и выдает его SVG-вариант, который затем импортируется в Reportlab при помощи SVGLib. Причем последний пришлось модифицировать для поддержки символов Юникода (что за формулы без греческих букв?). Это не считая многочисленных доработок и багфиксов, которые были проделаны в коде самого Reportlab (который, кстати, распространяется по открытой BSD-like лицензии).

Подводя итог, можно сказать, что, несмотря на кажущуюся сложность верстки электронного журнала, ее смело можно проводить и при помощи языков программирования. Я, конечно, далек от того, чтобы призывать всех отказываться от специализированных «верстальных» программ — но задуматься, так ли они необходимы, стоит однозначно.

Gecko
gecko0307@gmail.com

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 clocktower89@mail.ru или gecko0307@mail.ru.



Сборка от 31.12.2010.
python 2.5.2 (r252:60911, Oct 5 2008, 19:24:49) [GCC 4.3.2];
reportlab 2.5; svglib 0.6.3; svgmath 0.3.3; pygments 1.3.1.