ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение высшего профессионального образования
УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет математики и информационных технологий

Д.А. Мальцев

Мультимедиа технологии. Введение в технологии отображения 3D графики.

Методическое пособие для студентов 3 курса
факультета математики и информационных технологий специальностей
«Прикладная математика» и «Информационные системы».

ЧАСТЬ 1

Ульяновск – 2008

УДК 004.9 (075.8)
ББК 32.973.235я73+32.973.2-018.2я73
М21

Печатается по решению Ученого совета
факультета математики и информационных технологий
Ульяновского государственного университета

Рецензенты:
профессор, д.т.н. ***К.В. Кумунжиев***
ст. преподаватель, к.ф.-м.н. ***О.А. Фатьянова***

Мальцев, Д.А.

М 21 **Мультимедиа технологии. Введение в технологии отображения 3D графики :** методическое пособие для студентов 3 курса факультета математики и информационных технологий специальностей «Прикладная математика» и «Информационные системы» / Д.А. Мальцев. – Ч. I. – Ульяновск: УлГУ, 2008. – 59 с.

Методическое пособие содержит рассмотрение интерактивной (нелинейной) мультимедийной информации и библиотек для работы с ней. Особое внимание уделено программированию 3D-графики. Часть пособия посвящена началам аналитической геометрии: преобразования в трехмерном пространстве и их реализация в существующих библиотеках.

Пособие предназначено для студентов вузов прикладных математических специальностей.

УДК 004.9 (075.8)
ББК 32.973.235я73+32.973.2-018.2я73
М21

СОДЕРЖАНИЕ

Введение 5
Определение и классификация 5
Цели и задачи пособия 5
История развития 3D-Ускорителей 6
OpenGL 3.0 12
OpenGL Longs Peak 14
Новая объектная модель 15
Взбираясь на OpenGL Longs Peak 18
OpenGL Mount Evans 18
Реалии нашего времени 19
DirectX 10 22
Инфраструктурные изменения. 23
Изменения в Direct Graphics (D3D). 25
Аппаратные особенности DX10. 26
Математическая база 28
Векторы в трехмерном пространстве 28
Сравнение векторов. 32
Вычисление модуля вектора 33
Нормализация вектора 33
Сложение векторов 33
Вычитание векторов 34
Умножение вектора на скаляр 35
Скалярное произведение векторов 35
Векторное произведение 36
Матрицы 37
Равенство, умножение матрицы на скаляр и сложение матриц 38
Умножение 38
Единичная матрица 39
Инвертирование матриц 40
Транспонирование матриц 40

Матрицы в библиотеке D3DX 41
Матрицы в библиотеке OpenGL 43
Основные преобразования 44
Матрица перемещения 46
Матрицы вращения 47
Масштабирование. 49
Комбинирование преобразований 50
Некоторые функции для преобразования векторов 52
Матричный стек в OpenGL 53
Итоги 54
Список Рекомендуемой литературы 56
Список терминов 57
Anisotropic Filtering 57
Antialiasing 57
API 58
Vertex Buffer Object 58
Z-buffer 59

ВВЕДЕНИЕ

Определение и классификация

Мультимедиа (лат. Multum + Medium) — одновременное использование различных форм представления информации и ее обработки в едином объекте-контейнере. Например, в одном объекте-контейнере может содержаться текстовая, аудиальная, графическая и видео информация, а также, возможно, способ интерактивного взаимодействия с ней. Термин мультимедиа также, зачастую, используется для обозначения носителей информации, позволяющих хранить значительные объемы данных и обеспечивать достаточно быстрый доступ к ним (первыми носителями такого типа были CD-ROM). В таком случае термин мультимедиа означает, что компьютер может использовать такие носители и предоставлять информацию пользователю через все возможные виды данных, такие как аудио, видео, анимация, изображение и другие в дополнение к традиционным способам предоставления информации, таким как текст.

Мультимедиа может быть грубо классифицирована как ***линейная*** и ***нелинейная***. Аналогом линейного способа представления может являться кино. Человек, просматривающий данный документ никаким образом не может повлиять на его вывод. Нелинейный способ представления информации позволяет человеку участвовать в выводе информации, взаимодействуя каким-либо образом со средством отображения мультимедийных данных. Участие человека в данном процессе также называется «*интерактивностью*». Такой способ взаимодействия человека и компьютера наиболее полным образом представлен в категориях компьютерных игр. Нелинейный способ представления мультимедийных данных иногда называется «*гипермедиа*».

Цели и задачи пособия

Целью предмета «Мультимедиа технологии» («Мультимедиа системы») является формирование теоретических знаний в области создания, наполнения и использования мультимедийных данных в компьютерных информационных системах.

Данное методическое пособие первое в линейке книг посвященных теме создания программного обеспечения предоставляющего нелинейную мультимедийную информацию. Задачей методического пособия, является введение в технологию разработки приложений для отображения 3D-графики. Пособие разбито на 2 основных части: исследование существующих аппаратных и программных средств для отображения трехмерной графики и описание необходимой математической базы. Пособие будет полезно как начинающим разработчикам (математика преобразований, общие сведения о существующих технологиях, история развития), так и разработчикам уже знакомым с данной темой (особенности современных API, раскрытые в первой части книги).

ИСТОРИЯ РАЗВИТИЯ 3D-УСКОРИТЕЛЕЙ

Первые видеокарты не были даже не 3D-ускорителями, а не были ускорителями вообще. Они служили лишь как ЦАП[1] – преобразовывали данные, рассчитанные центральным процессором (представляющий собой цифровой код) в аналоговый сигнал, доступный для отображения на мониторе. Тенденция усложнения изображений привела к появлению 2D-ускорителя – видеокарты, имеющий свой собственный, пусть и простейший процессор, бравший на себя часть функций, разгружая центральный процессор. Однако с появлением 3D-графики, их возможностей стало не хватать.

Чтобы построить, скажем, простой фрагмент стены, процессору нужно было выполнить следующие операции: сначала необходимо выделить грани этого объекта, затем наложить текстуры, добавить освещение, а когда таких объектов сотни, их форма сложна, они движутся и перекрываются, отбрасывают тени, то задача становится невероятно сложной. Для помощи процессору в решении этой задачи и были созданы ускорители трёхмерной графики.

Для начала выясним, что собственно нужно, чтобы построить трёхмерное изображение. Необходимо уточнить, что видеоакселератор *не* занимается расчётом того, какую сцену он должен сейчас строить. Определением 3D сцены – объектов, точки наблюдения и т.п. занимается центральный процессор. Как только все необходимые данные собраны – они передаются видеокарте, которая начинает построение сцены.

Построение сцены происходит следующим образом (см. Рис. 1 и Рис. 2):

1. Загрузка в чип ускорителя вершин из памяти акселератора, со своими атрибутами.
2. Далее, каждая из вершин попадает в вершинный процессор. Подробнее мы вернемся к нему чуть позже.
3. Далее происходит установка треугольников - трансформированные и освещенные (т.е. уже обработанные вершинным шейдером или фиксированным T&L блоком)[2] вершины объединяются по три, и происходит подготовка данных для закраски треугольника. Здесь же происходит отсечение невидимых – перекрытых (overdraw) поверхностей.
4. Далее, треугольник разбивается на фрагменты, часть которых признаются невидимыми и отбрасывается в ходе Z-теста[3] на уровне фрагментов (то, что называется Hidden Surface Removal, HSR). Как правило, конечным результатом этого процесса являются видимые (или частично видимые) фрагменты 2x2 пикселя – так называемые «квады», подлежащие закраске. Именно такие фрагменты наиболее удобны для быстрой закраски пикселей.

---

[1] цифро-аналоговый преобразователь

[2] задача обеих технологий – преобразование координат у вершин

[3] определения и описание некоторых технологий даны в разделе «Список терминов»

5. Далее квады отправляются на установку фрагментов. Здесь для каждого из них вычисляется (интерполируется) множество необходимых параметров, таких как текстурные координаты, MIP уровень, векторы, установочные параметры анизотропии и т.д.
6. После установки и интерполяции параметров происходит закраска фрагментов. Существенной частью этого процесса является выборка и фильтрация текстур.
7. После того как значения цвета были рассчитаны, в пиксельном процессоре происходит смешение (блендинг) - если включен соответствующий режим - или просто запись результирующих значений цвета и глубины в буфер кадра. На этом этапе может происходить несколько дополнительных операций.
8. Ну а из буфера кадра происходит вывод изображения на экран, иногда после дополнительных проходов для AA (анти-альязинга).

Рисунок 1. Конвейер для обработки вершин.

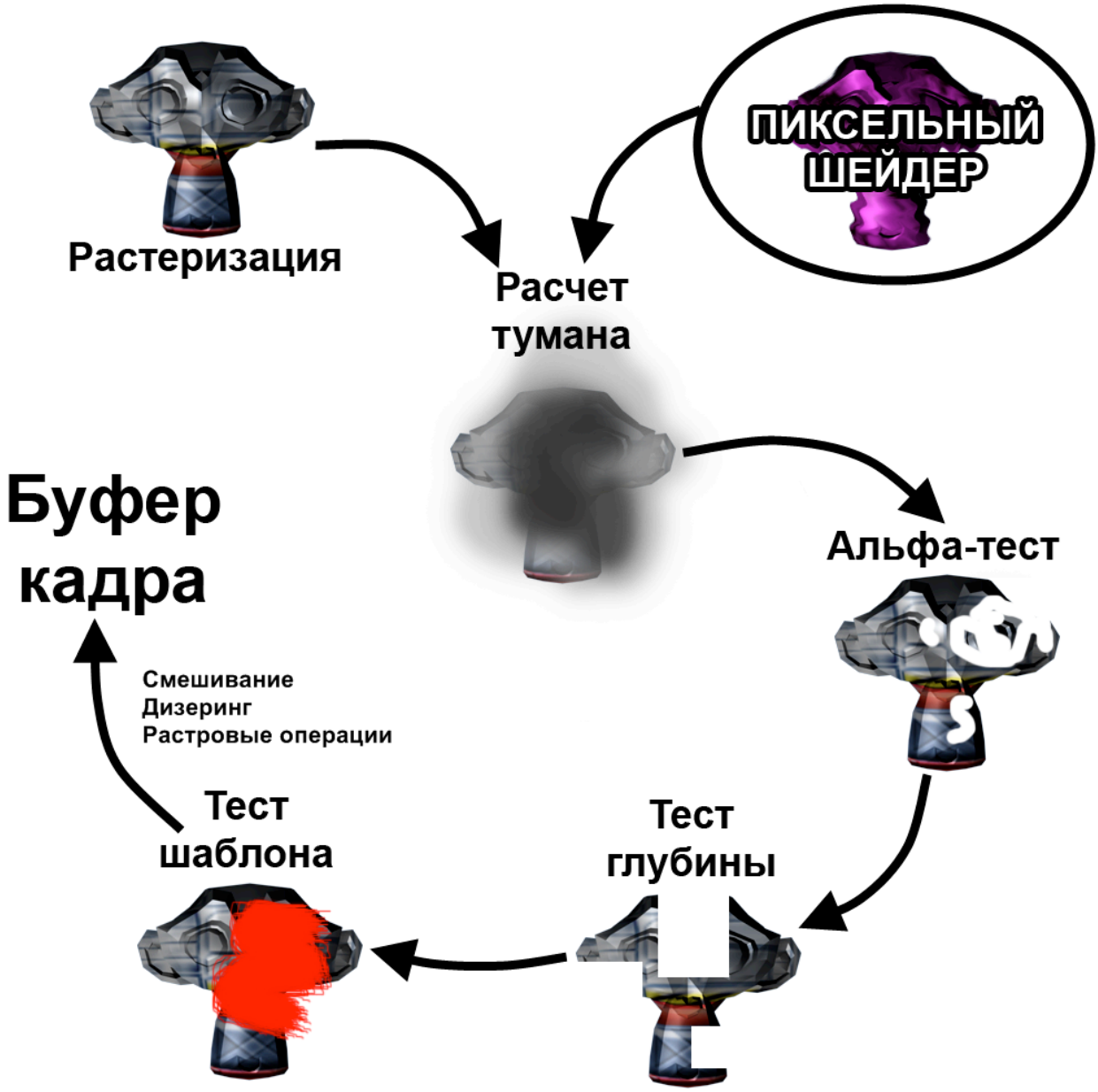

Рисунок 2. Конвейер для обработки фрагментов.

Итак, каждая стадия построения изображения очень ресурсоемка, требует множества расчётов. Вполне логичным выглядит шаг их вынесения из CPU и переправка на специализированный процессор видеокарты. Особенно если учитывать, что графические данные имеют потоковый характер, и вычислительную потребность значительно большую, чем логическую.

Каждый новый виток развития ускорителей представляет собой некое поколение, поэтому для начала введём стандартизацию поколений (понимать поколения можно по-разному – я приведу лишь один вариант):

1. Первое поколение, которое было более-менее распространено – это акселераторы, использующие API Direct3D 5 и Glide. Представителем первых была nVidia Riva128, а вторых – 3Dfx Voodoo. Карты этого поколения брали на себя только последнюю часть построения сцены – текстурирование и закраску. Все предыдущие этапы выполнял CPU.

2. Второе поколение использовало API Direct3D 6, также в это время началось стремительное возрождение API, разработанного SGI – OpenGL. Представителями карт того времени были nVidia RivaTNT и ATI Rage. Это было практически эволюционное развитие карт предыдущего поколения.
3. Третье поколение – Direct3D 7. Именно тогда появились карты, снабженные TCL-блоком (TCL – Transformaton Clipping Lighting), снимавшим с CPU значительную часть нагрузки. Этот блок отвечал за трансформацию, освещение и отсечение. Теперь видеокарта строила сцену самостоятельно – от начала до конца. Представителями этого поколения стали nVidia GeForce256 и ATI Radeon.
4. Четвёртое поколение – очередная революция. Кроме прочих новых возможностей API Direct3D 8.x эти карты принесли с собой самую главную возможность – аппаратные шейдеры. Представляют это поколение nVidia GeForce 3,4 и ATI Radeon 8x00, и младшие версии 9x00.
5. Пятое поколение – это, в основном, развитие шейдерных технологий (версии 2.0-3.0), и попытка ввести AA (Antialiasing) и AF (Anisotropic Filtering) в ряд обязательных к использованию функций. Это поколение, поддерживает API Direct3D версии до 9.0c включительно, представляют ATI Radeon 9500+ и X800+, а также nVidia GeForce FX 5x00, 6x00 и 7x00.
6. Шестое поколение – это поколение DirectX 10.x и OpenGL 3.0. Оно включает в себя серии nVidia GeForce 8x00, 9x00 и GTX2x0, а также AMD ATI Radeon HD 2x00, 3x00 и 4x00. Эти карты поддерживают шейдеры версии 4.0 и выше.

Теперь, определившись с общим устройством конвейера и поколениями видеокарт, мы более подробно рассмотрим вершинный и фрагментный процессоры, а также определимся в отличиях версий соответствующих шейдеров.

Причиной появления шейдеров стало отсутствие какой-либо гибкости у фиксированного TCL блока (его еще называют FFP – Fixed Function Pipeline, конвейер с фиксированной функциональностью). Быстро стало понятно, что ждать момента, когда производители внесут очередную порцию функций в TCL блок видеокарт не лучший выход. Такой подход не устраивал никого. Разработчикам не нравилась мысль, что для того, чтобы внести в, например, игру новый эффект им надо годик подождать выхода нового ускорителя. Производителям тоже не светило ничего хорошего – им бы пришлось постоянно увеличивать как сами чипы, так и драйверы к ним. Это и стало причиной появления шейдеров – программ, способных настраивать ускоритель так, как того требует следующая сцена.

Шейдер - это программа для процессора графической карты (GPU), выполняющая специфические задачи, заменяя соответствующие блоки FFP. Возможность написания полноценных программ появилась с видеоускорителя GeForce 3, но зачатки были реализованы в GeForce256 (при помощи технологии nVidia – Register Combiners[4]).

[4] В дальнейшем она не прижилась и осталась как расширение OpenGL. В DirectX реализована не была.

На данный момент есть три типа таких программ (шейдеров):

- Вершинный шейдер (в OpenGL - вершинная программа) - это программа для вершинного процессора, обрабатывающая вершинные данные.
- Пиксельный шейдер (в OpenGL - фрагментная программа) - это программа для фрагментного процессора, обрабатывающая данные фрагмента, такие как: степень затуманивания, текстурные координаты, глубина и прочее.
- Геометрический шейдер (в OpenGL – пока аналогов нет) – это программа позволяющая добавлять новые вершины при обработке 3D-объекта.

Первые шейдеры состояли всего из нескольких команд, и их нетрудно было написать на низкоуровневом языке ассемблера. Хотя сложность отладки ассемблерного кода поначалу отпугнула от шейдеров многих разработчиков. Но с ростом сложности шейдерных эффектов, насчитывающих иногда десятки и сотни команд, возникла необходимость в более удобном, высокоуровневом языке написания шейдеров. Их появилось сразу два: nVidia Cg (C for graphics) и Microsoft HLSL (High Level Shading Language) - последний является частью стандарта DirectX 9. Cg не получил широкого распространения, однако до сих пор используется некоторыми командами разработчиков, из-за его независимости от 3D API. Затем появился язык высокого уровня для OpenGL - GLSL (OpenGL Shading Language), который в последствии был включен в стандарт OpenGL 2.x.

Не будем проводить сравнения этих языков, а лишь отметим, что, как и в любой другой области, применение языков высокого уровня значительно сокращает объём необходимых работ по написанию и отладке кода.

OPENGL 3.0

OpenGL – это программный интерфейс к графической аппаратуре. Этот интерфейс состоит приблизительно из 250 отдельных команд (около 200 команд в самой OpenGL и еще 50 в библиотеке утилит), которые используются для указания объектов и операций, которые необходимо выполнить, чтобы получить интерактивное приложение, работающее с трехмерной графикой.

Библиотека OpenGL разработана как обобщенный, независимый интерфейс, который может быть реализован для различного аппаратного обеспечения. По этой причине сама OpenGL не включает функций для создания окон или для захвата пользовательского ввода; для этих операций вы должны использовать средства той операционной системы, в которой вы работаете. По тем же причинам в OpenGL нет высокоуровневых функций для описания моделей трехмерных объектов. Такие команды позволили бы вам описывать относительно сложные фигуры, такие как автомобили, части человеческого тела или молекулы. При использовании библиотеки OpenGL вы должны строить необходимые модели при помощи небольшого набора геометрических примитивов – точек, линий и многоугольников (полигонов).

Тем не менее, библиотека, предоставляющая описанные возможности может быть построена поверх OpenGL. Библиотека утилит OpenGL (OpenGL Utility Library -- GLU) предоставляет множество средств для моделирования, например, квадрические поверхности, кривые и поверхности типа NURBS. GLU – стандартная часть любой реализации OpenGL. Существуют также и более высокоуровневые библиотеки, например, Fahrenheit Scene Graph (FSG), которые построены с использованием OpenGL и распространяются отдельно для многих ее реализаций.

В некоторых реализациях (например, в реализации для системы X Window), OpenGL разработана таким образом, чтобы работать даже в том случае, если компьютер, который отображает графику не то же самый, на котором запущена ваша графическая программа. Это может происходить в случае, если работа происходит в сетевом окружении, состоящем из множества компьютеров, соединенных между собой. В данной ситуации компьютер, на котором функционирует программа, и вызываются команды OpenGL, является клиентом, в то время как компьютер, осуществляющий отображение, является сервером. Формат пересылки команд OpenGL от клиента серверу (или протокол) всегда один и тот же, так что программа может работать по сети даже в том случае, если клиент и сервер – совершенно различные компьютеры. В несетевом случае один и тот же компьютер является и клиентом, и сервером.

На данный момент последней версией OpenGL является версия 3.0. Рассмотрим сначала планы развития третьей версии OGL обещанные консорциумом ARB (Architects Registration Board). А затем текущее состояние дел с версией 3.0.

ATI и nVidia работали сообща более полугода над предложением изменения направления OpenGL. Большая часть времени была потрачена на общение с разработчиками программ с целью выяснить, что они хотят видеть в новом OpenGL. В итоге, предлагаются почти те же идеи, которые предлагались для OpenGL 2.0, с одной лишь разницей – это предлагается сделать для OpenGL 3.0. Представители компаний ATI и nVidia были инициаторами рефакторинга OpenGL, а в данный момент этим вопросом занимается уже вся группа ARB (которая влилась в Khronos, и теперь предложенные ARB решения окончательно принимаются в Khronos Board of Promoters).

На данный момент в задачу ARB входит «издание» двух новых версий OpenGL. Первая, под кодовым именем OpenGL "Longs Peak" (позже должна получить номер версии) и вторая, под кодовым именем OpenGL "Mount Evans".

Приблизительно то, что год назад предлагалось реализовать как OpenGL 3.0, будет реализовано как OpenGL Longs Peak. Тогда предлагалось сделать полную очистку и пересмотр кода, т.е. сделать OpenGL 3.0 несовместимым со старыми версиями. Однако ISVs (Independent software vendor) и некоторые IHVs (Independent hardware vendor) высказались за сохранение совместимости. Поэтому было решено разделить OpenGL на два профиля. Первый, прозванный OpenGL Lean and Mean (OpenGL LM), является ядром API и представляет прямую абстракцию возможностей аппаратного обеспечения, обеспечивает оптимальную производительность. Второй – полный профиль, поддерживает всю существующую функциональность OpenGL, (т. е. следует запомнить – будет полная обратная совместимость со старыми версиями). Он будет выполнен в виде программной прослойки, в которой будут собраны старые и редко используемые возможности, размещённой вне драйвера. Как только произойдёт разделение, наибольшие усилия в будущем будут вкладываться в разработку LM-профиля. Кроме того, будет введена новая объектная модель, которая должна существенно увеличить производительность драйвера.

В то время как OpenGL Longs Peak будет реализовываться для текущих графических чипов (предположительно NV4x, G7x, R5xx), OpenGL Mt. Evans предназначен для новейшего графического оборудования (G8x, R6xx и следующие за ними поколения). Он будет продолжением OpenGL Longs Peak, с новой функциональностью. Среди них – новые виды программируемых шейдеров, централизованная роль буферов данных, поддержка целочисленных операций в OpenGL Shading Language и др.

Почему две версии OpenGL? Решено, что так будет проще для разработчиков. Если вы хотите разрабатывать приложения, совместимые со старым графическим оборудованием, можно использовать OpenGL Longs Peak, но программы также будут работать и на новом оборудовании. Чтобы задействовать все возможности будущего оборудования, необходимо использовать OpenGL Mt. Evans, который, однако, несовместим со старым оборудованием и предыдущими версиями OpenGL. Ситуация чем-то похожа на использование версий Direct3D 9 и 10, с тем лишь отличием, что привязки к операционной системе может и не быть.

OpenGL Longs Peak.

Итак, OpenGL в том виде каким мы его знаем, будет разделён на два профиля. LM-профиль будет представлять собой ядро OpenGL, тонкую абстракцию над графическим оборудованием. В нём будет устранена толстая прослойка между приложением и оборудованием, что уменьшит сложность драйверов и сделает их быстрее. Можно будет писать полноценные приложения, используя только этот профиль. Вся устаревшая функциональность будет вынесена в программную прослойку, на которой будут держаться OpenGL приложения с полным профилем. Поощряться, однако, будет перенос приложений в сторону LM-профиля.

Следует отметить, что это планы годовалой давности, и некоторые из перечисленных ниже вещей могут попасть в Mount Evans, а не в Long Peaks.

Отрисовка геометрии. Какие возможности предлагается вынести в прослойку:

- Непосредственный режим (glBegin, glEnd, glVertex, glTexCoord и т. д.)
- Текущий vertex state
- Массивы вершин, создаваемые не при помощи VBO (Vertex Buffer Object)
- Включение массивов вершин (используя шейдеры, это должно делаться автоматически)
- Списки отображения
- Устаревшие команды, такие как glArrayElement(), glInterleavedArrays(), glRect()

Устранение этого уменьшит количество путей, которыми данные передаются в OpenGL, что упростит реализацию драйвера. VBO будет предпочтительным методом для передачи данных.

Планируется добавить:

- Геометрический инстансинг (instancing)
- Списки отображения только для геометрических данных (не будет GL_COMPILE_AND_EXECUTE, т. к. эта опция сложна для эффективной реализации)
- Массивы вершин, включаемые в зависимости от текущего вершинного шейдера

Объекты состояний (State Objects). Новые объекты будут инкапсулировать в себе множество состояний OpenGL, так, что можно будет переключаться между набором состояний за один вызов. Это будет заменой атрибутов в командах glPush/glPop. Совместно с VBO это должно будет работать очень эффективно. Такие объекты уже существуют в Direct3D 10.

Обработка вершин. Какие возможности предлагается вынести в прослойку: весь процесс фиксированной обработки вершин (трансформация и освещение), встроенные в GLSL переменные (gl_Vertex, gl_Normal и т. д.).

Обработка фрагментов. Какие возможности предлагается вынести в прослойку: текстурное окружение, суммирование цветов, туман. Планируется

добавить: бесшовные кубические карты, массивы текстур, шейдеры, работающие с текстурными фильтрами.

Растеризация примитивов. Какие возможности предлагается вынести в прослойку: ARB imaging subset, т. к. оказалось, что используется эта функциональность очень редко, более того, она может быть легко реализована в фрагментных шейдерах. Планируется добавить: в шейдере можно будет иметь доступ к значениям охвата (coverage values).

Расширение OpenGL Shading Language.

- Полная поддержка целочисленных типов данных и операций над ними – целочисленная операторы и математика, выборка из целочисленных текстуры, целочисленные вершинные атрибуты
- Offline-компиляция шейдеров (уже присутствует в OpenGL ES)
- Неизменяемая позиция вершины (замена ftransform)
- Расширенный контроль над интерполяторами (flat/smooth, center/centroid, perspective correct/not correct). centroid уже введён в GLSL 1.20
- Доступ к ID вершины (функциональность G8x)
- Привязывание текстур к самплерам (а не к текстурным блокам)

Дополнительно. Такая функциональность OpenGL, как color index rendering, accumulation buffer, evaluators, selection и feedback также будет вынесена в прослойку. Кроме того, планируется расширить удобство при работе с новым API. Будет введён callback механизм для отлавливания ошибок, больше кодов ошибок (в стандартном OpenGL их всего 7), команда glInfoLog для всех объектов.

Новая объектная модель

В новом OpenGL API будет существенно переработана объектная модель - менеджмент объектов будет значительно отличаться от того, который существует сейчас. Новая объектная модель должна улучшить производительность драйвера, быть проще в использовании (интуитивнее, меньше вызовов команд), и легкой в реализации для IHVs (что, в свою очередь, предполагает стабильность драйверов и упрощение жизни разработчиков приложений). В общем случае, она наверняка не будет уступать по производительности новому Direct3D 10 (а может, даже будет немного быстрее).

Существующая объектная модель развивалась в течении длительного промежутка времени и имеет несколько непоследовательный дизайн. Вначале были введены списки отображения в OpenGL 1.0, а затем – текстурные объекты в OpenGL 1.1. К сожалению, текстурные объекты были добавлены неоптимальным путём. Комбинация glBindTexture и glActiveTexture делает сложным отладку кода. Далее были добавлены новые типы объектов с иногда непоследовательной семантикой. Более того, существующая объектная модель оптимизирована для быстрого создания объекта, а не для производительной работы с ним в run-time (время выполнения кода, основной цикл выполнения кода, очень часто встречается в книгах как «рантайм»). В большинстве случаев

объект создаётся однажды и используется далее в работе, поэтому логичнее сделать так, чтобы объект был лёгок именно для рендеринга.

В текущей объектной модели приложения поставляют в API беззнаковое целое «имя», которое может генерироваться либо самим приложением, либо запросом к API (glGenLists, glGenTextures, glGenBuffers и т. д.). Когда это имя используется, реализация осуществляет выборку из хэш-таблицы, чтобы найти объект по этому имени. Далее объект привязывается (bind) для редактирования или использования. Объекты могут разделяться между контекстами.

Проблемы с использованием этой модели следующие:

- Стоимость выборки из хэш-таблицы может добавлять 3-5% ко времени работы драйвера.
- Процесс привязывания может быть причиной ошибок, из-за косвенного обращения (привязали объект в одной части программы, а его непреднамеренное редактирование может произойти совсем в другом месте).
- Драйверу сложно провести оптимизацию, т. к. он не может предсказать, что вы будете делать с объектом.
- Некоторые пробелы в поведении, например новое использование существующего имени.

Какие цели преследуются при разработке новой модели:

- Достижение максимальной производительности в run-time.
- Устранение незавершённых (incomplete) объектов (известны две проблемы – неполный набор MIP уровней в текстуре, и неполный набор буферов в FBO).
- Разделение между контекстами на пообъектной основе (а не все объекты сразу)
- Частичное устранение существующей семантики привязывания.

Что касается первого пункта, то здесь собираются пойти тем же путём, который используется в Direct3D 10. Т. е. максимальная производительность может быть достигнута только тогда, когда overhead в OpenGL драйвере минимизирован. Существует несколько путей:

- Количество проверок в реализации OpenGL, которые осуществляются в момент рисования (draw call), будет уменьшено. Сейчас, например, множество проверок могут происходить (и происходят) в момент рисования, замедляя скорость рендеринга.
- Количество вызовов API для смены установок конвейера будет уменьшено, улучшая тем самым производительность. Например, количество операций привязывания или копирования значений юниформов будет снижено.
- Новая модель уменьшит время, которое драйвер OpenGL затрачивает на поиск объекта по его имени.

Вместо используемых сейчас целочисленных имён, планируется создать новый тип, т. н. дескриптор объекта, размером как указатель в системе, на которой OpenGL запущен. Т. е:

```
typedef void *GLobject;
```

Это позволит (по выбору реализации) сохранять в дескрипторе указатель на структуру с внутренними данными, что позволит OpenGL быстро осуществлять к ним доступ (ранее существовали отдельные реализации OpenGL, в которых пытались использовать указатели под видом типа GLuint, но в документации по OpenGL писалось, что такая реализация не поощряется). Конечно, такой подход менее безопасен и может вызвать «падение» приложения. Предполагается ввести в OpenGL отладочный режим, в котором возможные ошибки при работе с объектами будут отлавливаться. Кроме того планируется, что для каждого объекта в OpenGL через typedef будет определён свой тип данных, что позволит проводить строгую проверку типов в compile-time. Это будет хорошим средством отладки, т. к. выдаваемые компилятором предупреждения будут индикаторами ошибок в программе.

Дескрипторы будут всегда генерироваться реализацией OpenGL. Такой дескриптор должен будет передаваться в каждую команду OpenGL, которая будет работать с объектом, который этот дескриптор описывает. Семантика «привязать для редактирования» и «привязать для использования» будет устранена. Нет ничего хорошего в том, чтобы устанавливать объект на конвейер только для того, чтобы отредактировать какое-то его свойство. Например:

```
glBindTexture(GL_TEXTURE_2D, texture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, mag);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, min);
```

В таких случаях будет передаваться сам дескриптор texture, а привязывание (binding) будет осуществляться только тогда, когда объект действительно требуется для рендеринга. К слову, такой механизм уже был введён в шейдерные объекты OpenGL 2.0 – дескриптор генерируется командами glCreateShader/glCreateProgram, далее дескриптор передаётся как аргумент функции, а устанавливается программный объект как часть конвейера только командой glUseProgram. Теперь этот механизм будет обобщён и распространён на все команды API.

Создание любого объекта в OpenGL станет атомарной операцией. Все атрибуты, которые необходимы для создания объекта на сервере OpenGL, будут передаваться во время создания. После создания реализация возвращает дескриптор объекта. Атрибуты будут храниться в специальных шаблонах (template), которые будут расширяемыми, в случае появления новых расширений, добавляющих к объекту новые атрибуты. Как только шаблон создан, он будет хранить атрибуты по умолчанию. Атрибуты в шаблонах будут изменяемыми.

Атрибуты, передаваемые при создании объекта, будут неизменяемыми всё время существования объекта. Это значит, что эти свойства объекта впоследствии не смогут быть изменены. Например, текстура будет иметь такие неизменяемые параметры, как формат и размерность. Можно будет менять лишь сами текстурные данные. В отличие от этого, существующие вызовы glTexImage1D/2D/3D изменяют как текстурные данные, так и внутренний формат и размеры текстуры. Отделение формы и размеров текстуры от её

данных поможет, например, драйверу лучше оптимизировать работу с памятью. Также устраняются проблемы с разделяемыми между контекстами объектами. Сейчас например, в документации неясно определено, как будет вести себя реализация OpenGL, если размерности текстуры, используемые в одном контексте, изменяются в другом контексте. Вы об этом не задумывались? А бывает оказывается и такое, и это головная боль для разработчиков драйверов.

Взбираясь на OpenGL Longs Peak

Не так давно появилась новая информация о важных решениях, которые приняло ARB относительно OpenGL Longs Peak:

1) Создание объектов будет асинхронной операцией. Это значит, что при вызове команды создания объекта она вернёт дескриптор ещё до того, как реальный объект будет создан реализацией OpenGL. «Крутость» этого подхода в том, что такой дескриптор – действительный и его можно использовать немедленно. Это открывает возможности параллельной работы приложения и реализации OpenGL, что является хорошей возможностью. Например, можно получить дескриптор текстуры, и далее выполнять какую-то другую работу. К тому времени, когда эта текстура понадобится для отображения, она уже будет создана на стороне сервера.

2) Множество привязываемых программных объектов. В OpenGL 2.1 на графический конвейер может устанавливаться только один программный объект. Это удобная модель, если присутствует только два программируемых этапа – вершинный и фрагментный, но она становится неудобной, если количество программируемых этапов увеличивается, т. к. количество возможных комбинаций этапов увеличивается и следовательно увеличивается количество необходимых программных объектов. В OpenGL Longs Peak станет возможно привязывать сразу несколько программных объектов для рендеринга. Каждый программный объект сможет хранить как один шейдер для одного программируемого этапа, так и несколько шейдеров для более чем одного программируемого этапа. Уже сейчас ясно, что добавлением в конвейер только геометрического шейдера дело не ограничится – мы увидим новые виды шейдеров, поэтому такая схема жизненно необходима.

3) Группы юниформных переменных можно будет объединять в блоки. Т. о. в шейдер можно будет передать сразу множество данных одним вызовом API. Также память для юниформов будет находиться в буфере, создаваемом приложением, и она сможет разделяться между множеством программных объектов. Подобная функциональность сейчас присутствует в OpenGL в виде расширения GL_EXT_bindable_uniform.

OpenGL Mount Evans

Разработкой этой версии OpenGL занимается отдельная группа в составе ARB - Next Gen TSG. Она будет продолжением Longs Peak и как уже было сказано, будет нацелена на использование совместно с новейшим графическим оборудованием. То, что можно прочитать из новостей ARB Next Gen TSG Update, свидетельствует об одном – в ядро будет включена большая часть

функциональности G8x и R6xx. Разработка API ведётся на базе предложенных компанией NVidia расширений.

В 2006 году в OpenGL 3.0 (тогда, напомню, было одно направление, а не два) планировалось добавить т. н. texture shaders, sample shaders и primitive shaders. Первые два вида шейдеров станут доступны в будущих поколениях графических ускорителей, поэтому эта функциональность должна попасть только в Mount Evans (для Direct3D её планируется добавить в DirectX 10.1).

Что должны заменить Texture Shaders:

1) Фильтры текстур
2) Режимы обёртки (wrap)
3) Режимы сравнения (для depth-текстур)
4) sRGB преобразования

Что должны заменить Sample Shaders:

1) Тест глубины
2) Тест трафарета
3) Операции смешивания (blend)

Из-за всего этого будут внесены значительные дополнения в OpenGL Shading Language, сообщается что группа Shading Language TSG очень серьёзно занята этой проблемой.

Все новые возможности будут введены в виде новой объектной модели, представленной в OpenGL Longs Peak. Из-за зависимостей спецификация возможностей Mount Evans будет завершена через 2-3 месяца после того, как Object Model TSG завершит свою работу над Longs Peak.

Реалии нашего времени

11 августа 2008 года на всеобщее обозрение выложили спецификацию долгожданного OpenGL 3.0, и разочаровали почти всё сообщество пользователей GL.

Всякие мечты о новом API, объектной модели, компиляции шейдеров в бинарный код и переработанном GLSL оказались разбиты. В общем, повторилась ситуация, знакомая по OpenGL 2.0.

Вместо всего этого, в очередной раз несколько расширений было добавлено в core, обновлён GLSL (до версии 1.3). Новый раздел в спецификации – deprecation model. Многие вещи были объявлены устаревшими, и с помощью расширения ARB_create_context можно создать контекст, в котором они не будут работать, и будут выдавать ошибки.

Список урезанного функционала:

- Режим индексного цвета (палитра), включая форматы текстур.
- Отрисовка с помощью glBegin/glEnd и из массивов, расположенных в системной памяти.
- Все встроенные вершинные атрибуты и функции по их установке. Теперь поддерживаются только пользовательские аттрибуты.
- Весь GL state кроме gl_DepthRange и стек матриц трансформаций, а также функции для настройки FFP.
- Смена толщины линий.

- Примитивы QUAD, QUAD_STRIP и POLYGON.
- Режим отображения задних и передних граней не может быть разным.
- Запись пикселей и растровые операции.
- Форматы ALPHA LUMINANCE, LUMINANCE ALPHA, и INTENSITY заменены R и RG текстурами.
- Альфатест.
- Буфер аккумулятора.
- Evaluators.
- Режим выбора.
- Списки отображения.
- Стек атрибутов.

Это многое из того, что убрали. А добавили в ядро следующие расширения:

- EXT_gpu_shader4 – возможности Shader Model 4.0. Нативная поддержка int и uint. Получения размера текстуры, обращение к конкретным текселям текстуры, или по обычным координатам, но с целочисленным сдвигом. gl_VertexID на вход вершинного шейдера. Выбор режима интерполяции varying'ов. Geometry shaders сюда не входят.
- NV_conditional_render – выполнения отрисовки в зависимости от результата query, который не нужно получать от видеокарты.
- Точное управление регионов для мапинга буферов в клиентскую память.
- ARB_color_buffer_float, NV_depth_buffer_float, ARB_texture_float, EXT_packed_float и EXT_texture_shared_exponent – различные float форматы текстур, а также 32-битный float буфер глубины.
- EXT_framebuffer_object – сколько лет прошло, и его одобрили.
- EXT_framebuffer_multisample и EXT_framebuffer_blit – в поддержку предыдущего, рендер в РТ с мультисемплингом, а также ресолв в обычную текстуру.
- NV_half_float, ARB_half_float_pixel – 16-ти битные форматы с плавающей точкой для вершинных атрибутов и фреймбуфера.
- EXT_texture_integer – целочисленные текстуры.
- EXT_texture_array – массивы текстур.
- EXT_packed_depth_stencil – формат render buffer с interleaved глубиной и значением стенциля (читать D24S8)
- EXT_draw_buffers2 – раздельный режим блендинга для разных таргетов при MRT.
- EXT_texture_compression_rgtc – поддержка сжатых форматов для одноканальных и двухканальных текстур.
- EXT_transform_feedback – Stream Out из VS.

- EXT_framebuffer_sRGB - sRGB текстуры появились в 2.1, а теперь в них можно рисовать.

В GLSL 1.3 добавили функционал из EXT_gpu_shader4, текстурную выборку с пользовательскими градиентами, функции для работы с целочисленными текстурами. Функции текстурных выборок переименовали: теперь не нужно писать тип текстуры (1D, 2D, 3D, Cube, *Shadow), просто texture и компилятор сам догадается, какая у нас текстура. Встроенных атрибутов больше нет, как нет и GL_state. ftransform() убрали, попросили ставить флажок invariant.

Поддержка всего этого возможна только на видеокартах уровня ДХ10, но почему-то многие основные возможности этих видеокарт оказались за бортом. Нет геометрических шейдеров, нет instancing, нет константных буферов, нельзя читать отдельные семплы multi-sampled текстуры в GLSL. Это всё предложено отдельными расширениями от ARB, кроме чтения из MS текстуры, но и оно может быть добавлено расширением. Вот только ждать этих расширений от некоторых вендоров возможно придётся довольно долго.

В итоге, вся надежда на коренные изменения опять переложена на следующую версию. Пока о ней сказано лишь то, что deprecated функции каким-то образом будут убраны полностью.

DIRECTX 10

DirectX – это набор мультимедийных драйверов, которые зачастую в обход стандартных сервисов ОС позволяют напрямую обращаться к конечному оборудованию, используя все его аппаратные возможности. Более того, некоторые Win32 сервисы используют DirectX ядро для своих собственных нужд. Набор мультимедийных API и драйверов высокого уровня – это будет наилучшим определением. Высокого, потому что в цепочке между самим DirectX и конечным оборудованием стоят ещё и драйвера производителя конкретного оборудования. Всегда следует помнить обобщенную схему взаимодействия конечного приложения с аппаратным обеспечением: Приложение - DirectX - Драйвер - Оборудование. DirectX состоит из нескольких компонент. Каждая из компонент обеспечивает функциональность в определенном направлении, как-то:

- DirectGraphics - 2D/3D графика.
- DirectShow - Потоковое видео.
- DirectInput - Различные устройства ввода.
- DirectSound и DirectMusic - Звук.
- DirectPlay - Стандартный интерфейс для создания сетевого взаимодействия на базе различных сетевых протоколов и провайдеров (под провайдером, конечно, понимается не та фирма у которой приобретаешь доступ в интернет).

Примечание: Среди выше перечисленных компонент не упомянут DirectSetup - небольшой API для работы с установкой DX на конечном компьютере.

Большая часть функциональности девятой и предыдущих версий DX строится на основе COM объектов, доступ к которым мы получаем через интерфейсы. Если коротко - COM модель представляет собой более жесткое определение объектно-ориентированной модели. С COM объектами мы работаем только через интерфейсы. Каждая компонента DirectX будь то, к примеру, DirectInput или DirectSound содержит некоторое количество интерфейсов, использованием функций которых мы получаем доступ к возможностям объекта. Грубо говоря, интерфейс представляет собой класс, содержащий в себе указатели на функции, с помощью которых и происходит взаимодействие с объектом.

Хронология развития DirectX[5]. Приблизительно в 1995 году после выхода ОС Windows 95 Microsoft выпускает Game SDK – первую версию DirectX. Основной упор делается на растровую двумерную графику. Кроме этого работа со звуком, сетью и устройствами ввода. 1996 год вместе с выходом Microsoft Developers Kit ознаменовывается появлением уже DirectX 2. Первые шаги в составе DirectX делает Direct3D. Всё тот же 96 - DirectX 3 можно назвать самым

[5] http://en.wikipedia.org/wiki/DirectX

универсальным из всех имеющихся DirectX'ов на планете, его функциональность доступна даже на NT 4.0. Пятая версия (MS пропустила выпуск 4-й версии) выходит в 98 году и радует обновленной DirectInput компонентой, которая по сути стала самостоятельной, а не оболочкой над Win32 функциями. Всё тот же 98 - шестая версия Direct3D (DirectX 6) уже поддерживает мультитекстурирование, stencil и w-buffer'ы. Конец 1999 года - по-своему революционная версия DirectX 7. Поддержка T&L (аппаратные трансформации и освещение) и совместимые видео карты. Размышления над фиксированным конвейером. Конец 2000 года - на сцену выступает DirectX 8. Программируемый конвейер, шейдеры, как средство программирования и очередной шаг от компьютерной графики к реальности. DirectDraw был исключен из API. Декабрь 2002 ознаменовался выходом DirectX 9. Основное число изменений коснулось именно работы с графикой. С этой версии разработчикам доступен HLSL (High-Level Shader Language) - высокоуровневый язык работы с шейдерами. Программно реализована поддержка второй и третьей версии пиксельных и вершинных шейдеров. 30 ноября 2006 года, вышел DirectX 10. О нем и пойдет разговор.

Инфраструктурные изменения.

Нужно отметить, что большинство из них относятся не только к DX10, но и вообще ко всем версиям DX на Windows Vista (Виста).

Новая driver model. В Висте совершенно новая driver model (WDDM - Windows Display Driver Model, в девичестве LDDM - Longhorn Display Driver Model), это самое большое изменение в модели видеодрайвера со времен, когда вообще появилось аппаратное ускорение графики.

Очень поверхностно о том, как было раньше, в XDDM (XP Display Driver Model). В XDDM каждый вызов DX добавлял указатель в так называемый “command buffer”, в независимом от видеокарты формате. Когда run-time решал, что буфер уже достаточно велик, вызывалась функция драйвера в kernel mode, ей передавался этот буфер, и драйверу нужно его разобрать и каким-то одному ему известным способом передать видеокарте. Никаких функций драйвера в user mode не было как таковых. К сожалению (нет, к счастью!), оказалось, что видеокарты очень динамично развиваются, и формат command buffer усложнялся и усложнялся. Появились сложные шейдеры, которые нужно компилировать в микрокод видеокарты, появились настройки под разные игры и так далее. В результате, в kernel mode оказался десяти мегабайтный с гаком кусок кода, который в случае любой ошибки убивает систему. Подавляющее большинство BSOD в XP – драйвер дисплея, самый сложный и опасный драйвер в системе. Вторая часть проблем про разделение видеокарты между несколькими процессами. В XDDM у ОС нет ни способа установить приоритет, ни управлять видеопамятью, ни вообще выполнять какой-то разумный порядок вызовов DX от разных процессов. Чуть что - Device Lost (потеря устройства вывода), и живой девайс остается у одного процесса.

В новой driver model есть разделение между user mode и kernel mode частью драйвера. Все вызовы DX напрямую идут в user-mode driver, который подготавливает сразу зависимый от оборудования буфер, который иногда

скидывает его в kernel, где он идет в видеокарту. Идея в том, что всю тяжелую работу можно выполнять в user-mode части, а в kernel фактически только переслать собранный буфер в DMA-трансфер видеокарте. Во-первых, если user-mode драйвер упадет, ничего страшного не случится - закроется конкретное приложение, но не вылетит ОС. Во-вторых, у драйвера больше контроля, когда и сколько вызовов в kernel делать, причем эти вызовы не тяжелые. В-третьих, run-time становится совсем тонкий - нет никаких command buffers, просто напрямую вызываются функции драйвера.

Кроме того, между user-mode и kernel-mode частями есть Microsoft GPU Scheduler, который может выбирать, какие собранные буфера отправлять видеокарте, то есть разделять GPU на много процессов.

Video memory virtualization. В новой модели драйвера есть виртуальная видеопамять - то есть, если ее не хватает, ресурсы будут переноситься в системную. Так как контроль за выделениями видеопамяти теперь у ОС, а не у драйвера, это можно делать гораздо эффективнее, чем POOL_MANAGED в XDDM. Сейчас это работает без аппаратной поддержки - GPU Scheduler перед передачей DMA-пакета карточке загружает все нужные текстуры в видеопамять. Он достаточно умный и умеет подгружать их заранее, пока GPU занят другим и свободна шина. Если приложение уйдет в полноэкранный режим - все остальное из видеопамяти выбросят по мере необходимости, если в windowed - то будет пытаться распределять память по текущим процессам. Собственно, совершенно аналогично обычной памяти.

Важный вопрос - контроль над производительностью со стороны приложения, когда хочется гарантировать наличие ресурса в видеопамяти. Ответ сейчас такой – полноэкранный режим и контроль над размером выделений видеопамяти.

То есть, больше не бывает Device Lost. Если станет активным другое приложение - ОС в случае необходимости выгрузит ресурсы и восстановит их обратно. Все еще есть “Device Removed”, но это уже экзотические случаи вроде “выдернули видеокарту” или “поставили новую версию драйвера”, о таких сценариях обычно не надо беспокоиться.

No caps! В DX10 больше нет caps'ов (проверок на совместимость текущего устройства вывода с используемыми технологиями в приложении, например, проверка поддержки 3-х шейдеров видеокартой), как таковых. Гарантируется наличие всей функциональности, за редкими исключениями (например, наличия блендинга для некоторых текстурных форматов, да и это поправили в DX10.1), то есть если карта поддерживает DX10, то она обязана держать последнюю версию шейдеров в полном объеме, поддерживать все форматы текстур, все возможные режимы фильтрации, трафарета и всего-всего-всего. Более того, для DX10 написали спецификацию правил растеризации треугольников и прочего, то есть теперь ожидается и то, что картинка на разных видеокартах на одинаковом коде всегда будет одинаковой и совпадать с эталонным программным растеризатором. Где не так - ошибка производителя карточки.

Lean & mean runtime. Много в API и driver model изменилось для того, чтобы уменьшить DIP cost (стоимость вызова отрисовки полигона, произошло от имени функции DrawIndexedPrimitives) на CPU. На XP он очень серьезный, тяжелый код может проводить около десятка миллисекунд (а то и больше), в вызовах DX. В них мало времени занимает сам рантайм, и много драйвер. Основное, что изменилось - сама модель драйвера, в которой рантайм фактически ничего не делает, а сразу предоставляет исполнение драйверу.

На стороне API появились State Objects, которые можно прекомпилировать при создании и потом быстро устанавливать на видеокарте и Constant Buffers как средство более эффективного управления обновлениями констант шейдеров. Разработчики из Crytek пишут, что действительно видели серьезное уменьшение DIP cost в Crysis на DX10.

Все эти инфраструктурные изменения и являются основной технической причиной того, что DX10 нет на XP. Переносить новую модель драйвера на XP не представляется возможным - слишком много изменений в ядре ОС, переносить все особенности DX10 на старую модель драйвера невозможно (виртуализация и управление, к примеру, принципиально не сможет работать). Есть все еще компромиссы - переносить только аппаратные особенности DX10 на старую модель драйвера, это прилично работы для MS и очень много работы для IHV - писать поддержку новых особенностей для новой и старой моделей драверов, и поведение API будет разным на разных ОС. Или предоставить прослойку, которая будет предоставлять DX9-функциональность с DX10-API, но это может и так написать разработчик, и остается открытым вопрос с представлением caps'ов с DX10 API. Все упирается в нехватку ресурсов для разработки, технически это все сложно и тяжело, но теоретически возможно.

Изменения в Direct Graphics (D3D).

В DX10 очень много изменений в 3D API, наверное, это одни из самых больших изменений за всю историю D3D. Большинство из них направлено на то, чтобы сделать интерфейс общим и уменьшить количество работы драйверу и рантайму.

Например, буфер вершин, буфер индексов и буфер констант - это один и тот же ID3D10Buffer, так как это набор байтов в памяти, и API к ним один и тот же. Однако при создании буфера все равно нужно указывать, как он будет использоваться, так что все это остается безопасно для применения.

Для ресурсов введены отдельные объекты для подключения к потоку отрисовки, так называемые resource views. Т.е. сначала создается текстура как объект в памяти, а потом создается ее resource view как входной параметр для шейдера или как render target, и уже с этим view вызывается PSSetShaderResources (вместо SetTexture) и OMSetRenderTargets (вместо SetRenderTarget). Разумеется, у одного ресурса view может быть несколько.

Все Set*State заменены на State Objects. Стейты разделены по нескольким группам: Rasterizer State (fill mode, cull mode, depth bias, multisample, scissor и т.д.), Blend State (alpha blend, color write mask, blend op и т.д.), Depth State (depth func, stencil func и все вокруг) и разумеется SamplerState (tex filtering, clamping и все такое). Стейты для каждой группы ставятся целиком, а не каждый по

отдельности, как в D3D9. Для каждой группы можно создать State Object, которому при создании указывается полный набор стейтов этой группы, и установить можно только его. Создание State Object - дорогая и медленная операция, и должна вызываться редко. Мотивация, опять же, та же самая - такой API позволяет драйверу сгенерировать набор команд видеокарте заранее (при создании State Object) и не генерировать его каждый раз во время рендера при вызовах Set*State.

Set*ShaderConstant заменены на Constant Buffers - группы констант, устанавливающихся за раз. Т.е. создается буфер на n констант, как и обычный буфер, и подключать его к шейдеру начиная с какого-то слота. В D3D9 рекомендовали устанавливать константы не по одной, а блоками подряд для производительности. В DX10 более формализованный способ делать то же самое. То есть, разделяем константы на несколько групп по частоте обновления - per-object, per-material, per-pass, per-scene, для каждой заводим свой Constant Buffer, и обновляем их по мере необходимости. Таким образом, разделение на блоки обновлений констант, необходимое для производительности, происходит автоматически, и дает драйверу высокоуровневую картину, а следовательно больше возможностей для оптимизации.

Основная идея всех перечисленных нововведений - как можно больше выносить в создание-инициализацию, как можно больше вычислять и заранее проверять, чтобы при рендере делать как можно меньше, и таким образом уменьшать DIP cost.

Кроме этого, из изменений стоит отметить, что шейдеры больше нельзя писать на ассемблере, нужно пользоваться HLSL. Хотя ассемблер для shader model 4.x есть и можно смотреть результат компиляции шейдеров в него, больше нет возможности получить бинарный код шейдера из текста ассемблера.

Чтобы было легче портировать код шейдеров, компилятор умеет компилировать HLSL-шейдеры старых версий (SM2.0, SM 3.0) в SM4.0. В новом HLSL, добавили атрибуты для подсказок компилятору - unroll для циклов и выбор dynamic vs static branching (динамический или статический переход) для условных переходов.

Аппаратные особенности DX10.

Конечно же, первым пунктом аппаратных особенностей DX10 идут Geometry Shaders (геометрические шейдеры). Geometry Shader - это дополнительный шейдер между Vertex Shader и Pixel Shader, который может генерировать примитивы. На вход ему подается примитив с информацией о соседях, на выход - можно сгенерировать несколько (не фиксированное число) новых примитивов. Основная идея - наконец генерировать геометрию на GPU. Как пример, построение shadow volumes (одна из техник построения теней) на GPU. Основное препятствие использования GS - их скорость на современном

оборудовании. Именно поэтому все статьи про GS в GPU Gems (см. список литературы) - от nVidia, а не от разработчиков[6].

Нельзя не упомянуть о Stream Out. Это возможность записывать результат работы Vertex Shader/Geometry Shader в память. Например, кэшировать обработку геометрии или вообще геометрию, созданную GS. Можно считать итеративные эффекты, типа моделирования ткани или водной поверхности. То есть теперь можно напрямую трансформировать и записывать геометрию на GPU, а не только рисовать пиксели в Render Target.

Близкая особенность - возможность читать в шейдере из буфера памяти по индексу, то есть иметь достаточно большую разделяемую память только для чтения. nVidia предлагает хранить там константы анимации для instancing'a.

Эволюционные изменения в шейдерах - в четвертой шейдерной модели добавили целочисленные инструкции и битовые операции (наконец-то можно считать в честном fixed point и передавать булевы флажки), убрали ограничение на количество инструкций (но, очень длинный шейдер может упереться в ограничение по времени выполнения пакета на GPU).

Отдельная группа особенностей посвящена уменьшению количества вызовов отрисовок и переключений состояний. Появились массивы текстур, то есть контейнер одинаковых по размеру и формату текстур, из которого шейдер может выбирать по индексу (в DX10.1 - можно и cubemap arrays). В шейдер приходят primitive/instance id, в зависимости от instance id можно использовать другой набор текстур/координат. То есть, в теории, можно за один вызов генерировать большое количество геометрии с разными параметрами, текстурами и вообще материалами. На практике, больше всего мешает стоимость dynamic branch и проблем, с ним связанных (вычисление градиентов текстурных координат). А остальное - вполне можно и нужно использовать.

Multi-sampling antialiasing features. Небольшая особенность, ради одной которой можно переходить на DX10. Теперь в шейдере можно читать каждый MSAA-сэмпл отдельно, то есть писать свой собственный AA-фильтр и вообще использовать MSAA RT как текстуру.

Официальная поддержка depth textures (текстур, которые хранят буфер глубины). Можно сказать, чтобы при сэмплинге сравнивал со значением и делал фильтрацию соседей, можно достать чистый depth value. Можно даже stencil value достать.

Как итог, все стало более последовательно, логично и с меньшим количеством исключений из правил. Можно больше работы делать на GPU, не привлекая CPU, появились новые задачи, которые вообще можно решать на GPU, исправлены многие старые проблемы, внесены в API и легализованы полезные недокументированные особенности IHV.

---

[6] Причина очевидна: задача nVidia – показать возможности современных GPU на примере технологических демо-программ, разработчики же пишут алгоритмы применяемые в реальных приложениях: играх, CAD-системах и т.п.

МАТЕМАТИЧЕСКАЯ БАЗА

Задача данной главы – предоставить минимальную математическую базу, необходимую при разработке 3D приложений. Настоятельно рекомендуются к самостоятельному изучению основы линейной алгебры и основы аналитической геометрии.

Векторы в трехмерном пространстве

Геометрическим представлением вектора является направленный отрезок прямой линии, что показано на рис. 3. У каждого вектора есть два свойства: длина (также называемая модулем или нормой вектора) и направление. Благодаря этому векторы очень удобны для моделирования физических величин, которые характеризуются модулем и направлением. С другой стороны, в трехмерной компьютерной графике векторы часто используются только для моделирования направления. Например, нам часто требуется указать направление распространения световых лучей, ориентацию грани или направление камеры, глядящей на трехмерный мир. Векторы обеспечивают удобный механизм задания направления в трехмерном пространстве.

Рисунок 3

Поскольку местоположение не является характеристикой вектора, два вектора с одинаковой длиной и указывающие в одном и том же направлении считаются равными, даже если они расположены в различных местах. Обратите внимание, что два таких вектора будут параллельны друг другу. Например, на рисунке векторы u и v равны.

На этом рис. 3 видно, что обсуждение векторов может вестись без упоминания системы координат, поскольку всю значимую информацию, — длину и направление, — вектор содержит в себе. Добавление системы координат не добавляет информации в вектор; скорее можно говорить, что вектор, значения которого являются его неотъемлемой частью, просто описан

относительно конкретной системы координат. И если мы изменим систему координат, мы только опишем тот же самый вектор относительно другой системы.

Отметив этот важный момент, перейдем к изучению того, как векторы описываются в левосторонней трехмерной декартовой системе координат. На рис. 4 показаны левосторонняя и правосторонняя системы координат. Различие между ними — положительное направление оси Z. В левосторонней системе координат положительное направление оси Z погружается в страницу. В правосторонней системе координат положительное направление оси Z направлено от страницы.

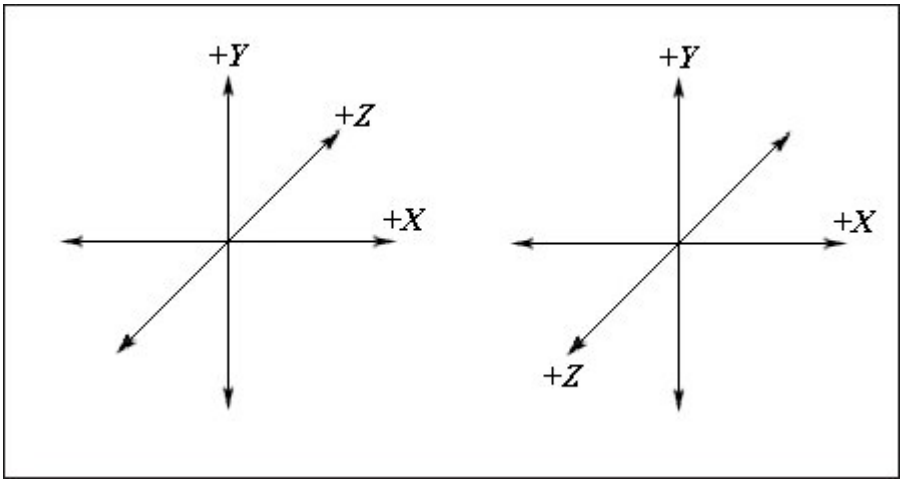

Рисунок 4

Поскольку местоположение вектора не изменяет его свойств, мы можем перенести векторы таким образом, чтобы начало каждого из них совпадало с началом координат выбранной координатной системы. Когда начало вектора совпадает с началом координат, говорят, что вектор находится в стандартной позиции. Таким образом, если вектор находится в стандартной позиции, мы можем описать его, указав только координаты конечной точки. Мы будем называть эти координаты компонентами вектора. На рис. 5 показаны векторы, изображенные на рис. 3, которые были перемещены в стандартные позиции.

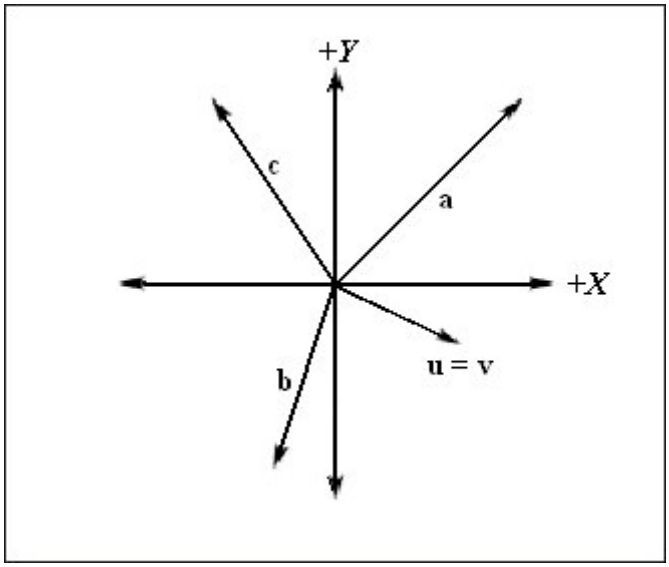

Рисунок 5

Примечание: Поскольку мы описываем находящийся в стандартной позиции вектор, указывая его конечную точку, как если бы мы описывали отдельную точку, легко перепутать точку и вектор. Чтобы подчеркнуть различия между этими двумя понятиями, мы вновь приведем определение каждого из них. Точка описывает только местоположение в системе координат, в то время как вектор описывает величину и направление.

Будем пользоваться для обозначения векторов строчными буквами, но иногда будем применять и полужирные заглавные буквы. Вот пример двух-, трех- и четырехмерных векторов соответственно:

$$u = (u_x, u_y), N = (N_x, N_y, N_z), c = (c_x, c_y, c_z, c_w)$$

Теперь введем четыре специальных трехмерных вектора, которые показаны на рис. 4. Первый из них называется нулевым вектором, и значения всех его компонент равны нулю; мы будем обозначать такой вектор нулем: 0 = (0, 0, 0). Следующие три специальных вектора называются единичными базовыми векторами (базовыми ортами) трехмерной системы координат. Эти векторы, направленные вдоль осей X, Y и Z координатной системы, будем называть i, j и k соответственно. Модуль этих векторов равен единице, а определение выглядит следующим образом: i = (1, 0, 0), j = (0, 1, 0), k = (0, 0, 1).

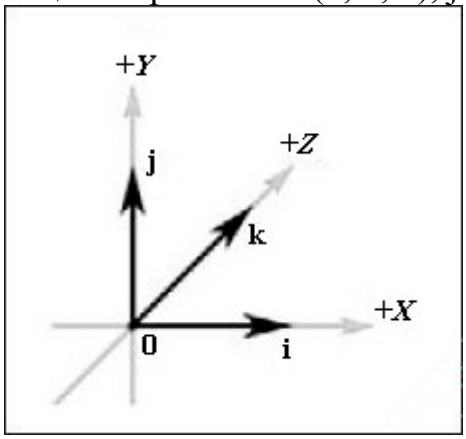

Рисунок 6

Примечание: Вектор, длина которого равна единице, называется единичным вектором или ортом.

Эти вектора изображены на рис. 6.

В библиотеке D3DX для представления векторов в трехмерном пространстве можем воспользоваться классом D3DXVECTOR3. Его определение выглядит следующим образом:

```
typedef struct D3DXVECTOR3 : public D3DVECTOR
{
public:
    D3DXVECTOR3() {};
    D3DXVECTOR3( CONST FLOAT * );
    D3DXVECTOR3( CONST D3DVECTOR& );
    D3DXVECTOR3( FLOAT x, FLOAT y, FLOAT z );

    // приведение типа
    operator FLOAT* ();
    operator CONST FLOAT* () const;

    // операторы присваивания
    D3DXVECTOR3& operator += ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator -= ( CONST D3DXVECTOR3& );
    D3DXVECTOR3& operator *= ( FLOAT );
    D3DXVECTOR3& operator /= ( FLOAT );

    // унарные операторы
    D3DXVECTOR3 operator + () const;
    D3DXVECTOR3 operator - () const;

    // бинарные операторы
    D3DXVECTOR3 operator + ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator - ( CONST D3DXVECTOR3& ) const;
    D3DXVECTOR3 operator * ( FLOAT ) const;
    D3DXVECTOR3 operator / ( FLOAT ) const;
    friend D3DXVECTOR3 operator * ( FLOAT,
           CONST struct D3DXVECTOR3& );
    BOOL operator == ( CONST D3DXVECTOR3& ) const;
    BOOL operator != ( CONST D3DXVECTOR3& ) const;
} D3DXVECTOR3, *LPD3DXVECTOR3;
```

Обратите внимание, что D3DXVECTOR3 наследует компоненты от D3DVECTOR, определение которого выглядит следующим образом:

```
typedef struct _D3DVECTOR {
    float x;
    float y;
    float z;
} D3DVECTOR;
```

Так же, как и у скалярных величин, у векторов есть собственная арифметика, что видно из наличия описаний математических операций в определении класса D3DXVECTOR3. Возможно, сейчас вы не знаете, что делают эти методы.

В OpenGL отдельной структуры для векторов не существует. Практически всю математическую базу для работы с векторами приходится

писать самостоятельно. Сам же вектор может быть представлен в виде массива из 2-х, 3-х или 4-х элементов типа float или указывается раздельными параметрами в функции.

Примечание: Хотя основной интерес представляют векторы в трехмерном пространстве, занимаясь программированием трехмерной графики часто сталкиваются с векторами в двухмерном и четырехмерном пространствах. Библиотека D3DX предоставляет классы D3DXVECTOR2 и D3DXVECTOR4, предназначенные для представления векторов в двухмерном и четырехмерном пространствах соответственно. Векторы в пространствах с другим количеством измерений обладают теми же свойствами, что и векторы в трехмерном пространстве, а именно — длиной и направлением, отличается только количество измерений. Кроме того, математические операции с векторами, за исключением векторного произведения, которое определено только для трехмерной системы координат, могут быть обобщены для векторов любой размерности. Таким образом, за исключением векторного произведения, все операции для векторов в трехмерном пространстве, распространяются и на векторы в двухмерном, четырехмерном и даже n-мерном пространствах.

Сравнение векторов.

В геометрии два вектора считаются равными, если они указывают в одном и том же направлении и имеют одинаковую длину. В алгебре говорят, что векторы равны, если у них одинаковое количество измерений и их соответствующие компоненты равны. Например, $(u_x, u_y, u_z) = (v_x, v_y, v_z)$ если $u_x = v_x$, $u_y = v_y$ и $u_z = v_z$.

В коде можно проверить равны ли два вектора, используя перегруженный оператор равенства:

```
D3DXVECTOR u(1.0f, 0.0f, 1.0f);
D3DXVECTOR v(0.0f, 1.0f, 0.0f);
if( u == v ) return true;
```

Аналогичным образом, можно убедиться, что два вектора не равны, используя перегруженный оператор неравенства:

```
if( u != v ) return true;
```

Примечание: Сравнивая числа с плавающей точкой следует быть очень аккуратным, поскольку из-за погрешностей округления, два числа с плавающей точкой, которые должны быть равными, могут слегка отличаться. По этой причине предпочтительно проверять приблизительное равенство чисел с плавающей точкой. Для этого мы определяют константу EPSILON, содержащую очень маленькое значение, которое будет служить «буфером». Будем говорить, что два числа приблизительно равны, если разница между ними меньше EPSILON. Другими словами, EPSILON дает некий допуск для ошибок округления чисел с плавающей точкой. Приведенная ниже функция показывает, как EPSILON может использоваться при проверке равенства двух чисел с плавающей точкой:

```
const float EPSILON = 0.001f;
bool Equals(float lhs, float rhs)
{
   // если lhs == rhs разность должна быть равна нулю
```

```
    return fabs(lhs - rhs) < EPSILON ? true : false;
}
```

Об этом не надо беспокоиться, работая с классом D3DXVECTOR, поскольку перегруженные операции сравнения все делают автоматически, но очень важно знать об этой особенности сравнения чисел с плавающей точкой при программировании в OpenGL.

Вычисление модуля вектора

В геометрии модулем вектора называется длина направленного отрезка линии. В алгебре, зная компоненты вектора можно вычислить его модуль по следующей формуле:

$$|u| = \sqrt{u_x^2 + u_y^2 + u_z^2}$$

Вертикальные линии в |u| обозначают модуль u. Работая с библиотекой D3DX, для вычисления модуля вектора применяют следующую функцию:

```
FLOAT D3DXVec3Length(     // Возвращает модуль
    CONST D3DXVECTOR3* pV // Вектор, чью длину мы вычисляем
);
…
D3DXVECTOR3 v(1.0f, 2.0f, 3.0f);
float magnitude = D3DXVec3Length(&v); // = sqrt(14)
```

Нормализация вектора

В результате нормализации получается вектор, направление которого совпадает с исходным, а модуль равен единице (единичный вектор). Чтобы нормализовать произвольный вектор, достаточно разделить каждый компонент вектора на модуль вектора, как показано ниже:

$$\hat{u} = \frac{u}{|u|} = \left(\frac{u_x}{|u|}, \frac{u_y}{|u|}, \frac{u_z}{|u|}\right)$$

Единичный вектор отмечается размещением над его обозначением символа ^(крышки): û. В библиотеке D3DX для нормализации векторов применяется следующая функция:

```
D3DXVECTOR3 *D3DXVec3Normalize(
    D3DXVECTOR3* pOut,    // Результат
    CONST D3DXVECTOR3* pV // Нормализуемый вектор
);
```

Примечание: Эта функция возвращает указатель на результат, который может быть передан в качестве параметра другой функции. В большинстве случаев, за исключением явно указанных, математические функции библиотеки D3DX возвращают указатель на результат.

Сложение векторов

Можно сложить два вектора, сложив их соответствующие компоненты; обратите внимание, что размерность складываемых векторов должна быть одинаковой:

$$u + v = (u_x + v_x, u_y + v_y, u_z + v_z)$$

Геометрическая интерпретация сложения векторов показана на рис. 7.

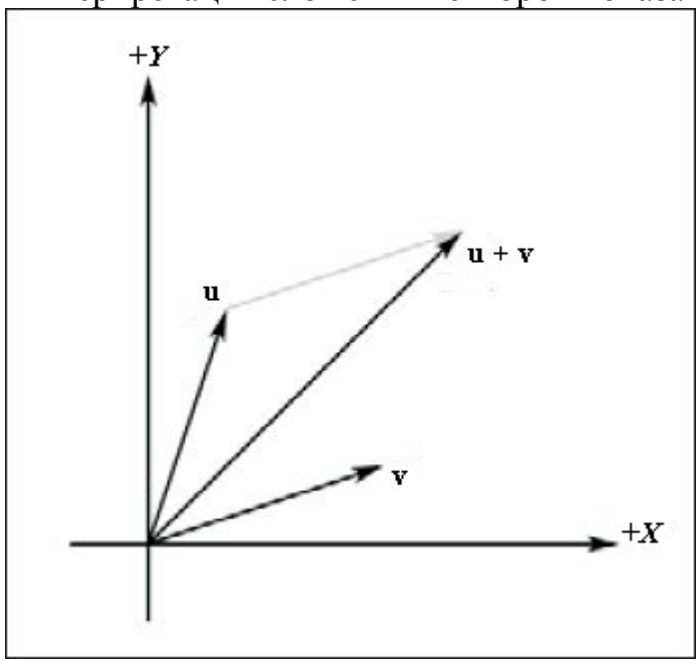

Рисунок 7

Обратите внимание, как выполняется параллельный перенос вектора v таким образом, чтобы его начало совпало с концом вектора u; суммой будет вектор, начало которого совпадает с началом вектора u, а конец совпадает с концом перенесенного вектора v.

В коде для сложения двух векторов будем применять перегруженый оператор сложения:

```
D3DXVECTOR3 u(2.0f, 0.0f, 1.0f);
D3DXVECTOR3 v(0.0f, -1.0f, 5.0f);
// (2.0 + 0.0, 0.0 + (-1.0), 1.0 + 5.0)
D3DXVECTOR3 sum = u + v; // = (2.0f, -1.0f, 6.0f)
```

Вычитание векторов

Аналогично сложению, вычитание векторов осуществляется путем вычитания их отдельных компонент. Опять же оба вектора должны иметь одинаковую размерность.

$$u - v = u + (-v) = (u_x - v_x, u_y - v_y, u_z - v_z)$$

Геометрическая интерпретация вычитания векторов показана на рис. 8.

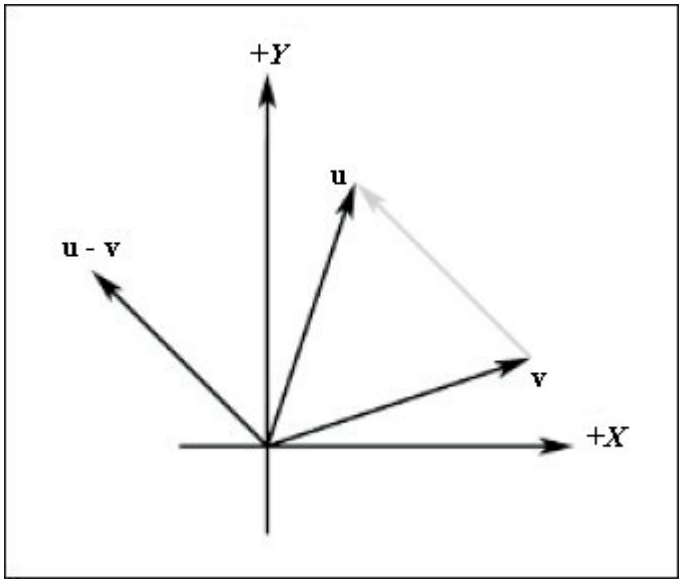

Рисунок 8

В коде для вычитания двух векторов применяют перегруженый оператор вычитания:

```
D3DXVECTOR3 u(2.0f, 0.0f, 1.0f);
D3DXVECTOR3 v(0.0f, -1.0f, 5.0f);
D3DXVECTOR3 difference = u - v; // = (2.0f, 1.0f, -4.0f)
```

Как видно на рис. 8, операция вычитания векторов возвращает вектор, начало которого совпадает с концом вектора v, а конец — с концом вектора u. Если мы интерпретируем компоненты u и v как координаты точек, то результатом вычитания будет вектор, направленный от одной точки к другой. Это очень удобная операция, поскольку часто будет необходимо найти вектор, описывающий направление от одной точки к другой.

Умножение вектора на скаляр

Как видно из названия раздела, можно умножать вектор на скаляр, в результате чего происходит масштабирование вектора. Если масштабный множитель положителен, направление вектора не меняется. Если же множитель отрицателен, то направление вектора изменяется на противоположное (инвертируется).

$$ku = (ku_x, ku_y, ku_z)$$

Класс D3DXVECTOR3 предоставляет оператор умножения вектора на скаляр:

```
D3DXVECTOR3 u(1.0f, 1.0f, -1.0f);
D3DXVECTOR3 scaledVec = u * 10.0f;
// = (10.0f, 10.0f, -10.0f)
```

Скалярное произведение векторов

Скалярное произведение векторов — это первая из двух определенных в векторной алгебре операций умножения. Вычисляется такое произведение следующим образом:

$$u \cdot v = u_x v_x + u_y v_y + u_z v_z = s$$

У приведенной выше формулы нет очевидной геометрической интерпретации. Используя теорему косинусов, мы получим отношение

$$u \cdot v = |u||v|\cos(j)$$

говорящее, что скалярное произведение двух векторов равно произведению косинуса угла между векторами на модули векторов[7]. Следовательно, если u и v — единичные векторы, их скалярное произведение равно косинусу угла между ними.

Вот некоторые полезные свойства скалярного произведения:

- Если u·v = 0, значит u ^ v, символ ^ обозначает «ортогональный» или «перпендикулярный».
- Если u·v > 0, значит угол j между двумя векторами меньше 90 градусов.
- Если u·v < 0, значит угол j между двумя векторами больше 90 градусов.

Для вычисления скалярного произведения двух векторов в библиотеке D3DX предназначена следующая функция:

```
FLOAT D3DXVec3Dot( // Возвращает результат.
    CONST D3DXVECTOR3* pV1, // Левый операнд.
    CONST D3DXVECTOR3* pV2  // Правый операнд.
);

D3DXVECTOR3 u(1.0f, -1.0f, 0.0f);
D3DXVECTOR3 v(3.0f, 2.0f, 1.0f);
// 1.0*3.0 + -1.0*2.0 + 0.0*1.0
// = 3.0 + -2.0
float dot = D3DXVec3Dot(&u, &v); // = 1.0
```

Векторное произведение

Второй формой операции умножения, определенной в векторной алгебре, является векторное произведение. В отличие от скалярного произведения, результатом которого является число, результатом векторного произведения будет вектор. Векторным произведением двух векторов u и v будет другой вектор p, являющийся взаимно перпендикулярным для векторов u и v. Это означает, что вектор p перпендикулярен вектору u и одновременно вектор p перпендикулярен вектору v.

Вычисляется векторное произведение по следующей формуле:

$$p = u \times v = [(u_y v_z - u_z v_y), (u_z v_x - u_x v_z), (u_x v_y - u_y v_x)]$$

В компонентной форме вычисление выглядит так:

$$p_x = (u_y v_z - u_z v_y);\ p_y = (u_z v_x - u_x v_z);\ p_z = (u_x v_y - u_y v_x)$$

[7] Теорема косинусов определяет зависимость между сторонами и углами треугольника. Она утверждает, что во всяком треугольнике квадрат длины стороны равен сумме квадратов двух других сторон без удвоенного произведения длин этих сторон на косинус угла между ними. Если угол прямой, то теорема косинусов переходит в теорему Пифагора, т.к. косинус прямого угла равен 0.

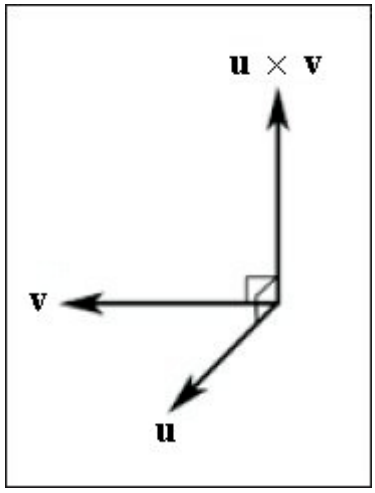

Рисунок 9

На рис. 9 вектор p = u × v перпендикулярен как вектору u, так и вектору v. Для вычисления векторного произведения двух векторов в библиотеке D3DX предназначена следующая функция:

```
D3DXVECTOR3 *D3DXVec3Cross(
    D3DXVECTOR3* pOut,      // Результат
    CONST D3DXVECTOR3* pV1, // Левый операнд
    CONST D3DXVECTOR3* pV2  // Правый операнд
);
```

Как явствует из рисунка, вектор –p также взаимно перпендикулярен векторам u и v. Какой из векторов, p или –p будет возвращен в качестве результата векторного произведения определяется порядком операндов. Другими словами, u × v = –(v × u). Это заначит, что операция векторного произведения не является коммутативной. Определить, какой вектор будет возвращен в качестве результата, можно с помощью правила левой руки. (Используем правило левой руки, поскольку работаем с левосторонней системой координат обычно в DirectX. Если бы была правосторонняя система координат, как в OpenGL, пришлось бы воспользоваться правилом правой руки.) Если расположить пальцы левой руки вдоль первого вектора, а ладонь руки — вдоль второго, отогнутый на 90 градусов большой палец укажет направление результирующего вектора.

Матрицы

В этом разделе сосредоточимся на математике матриц. Их использование в трехмерной компьютерной графике будет рассмотрено ниже.

Матрицей m × n называется прямоугольный массив чисел, состоящий из m строк и n столбцов. Количество строк и столбцов определяет размер матрицы. Отдельный элемент матрицы идентифицируется путем указания его строки и столбца в состоящем из двух элементов спсике индексов; первый индекс определяет строку, а второй — столбец. Ниже в качестве примера приведены матрицы M размером 3 × 3, B размером 2 × 4 и C размером 3 × 2:

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}; B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \end{bmatrix}; C = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{bmatrix}$$

В большинстве случаев для обозначения матриц будем использовать заглавные буквы. Иногда матрицы состоят из единственной строки или единственного столбца. Чтобы отличать такие матрицы, дадим им специальные имена: вектор-строка (row vector) и вектор-столбец (column vector). Вот примеры таких векторов:

$$v = [v_1, v_2, v_3, v_4]; u = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix}$$

Для элементов вектора-строки и вектора-столбца необходим только один индекс. Иногда для идентификации элемента строки или столбца в качестве индекса будем использовать буквы.

Равенство, умножение матрицы на скаляр и сложение матриц

Для пояснения рассматриваемых терминов в данном разделе будут использованы следующие четыре матрицы:

$$A = \begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix}; B = \begin{bmatrix} 6 & 2 \\ 5 & -8 \end{bmatrix}; C = \begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix}; D = \begin{bmatrix} 1 & 2 & -1 & 3 \\ -6 & 3 & 0 & 0 \end{bmatrix}$$

Две матрицы считаются равными, если они имеют одинаковую размерность и их соответствующие элементы равны. Например, A = C, поскольку матрицы A и C имеют одинаковую размерность и их соответствующие элементы равны. Мы говорим, что A ≠ B и A ≠ D поскольку у этих матриц либо разная размерность, либо не равны соответствующие элементы.

Мы можем умножить матрицу на скаляр для чего нам необходимо умножить каждый элемент матрицы на данный скаляр. Например, умножив D на скаляр k получим:

$$kD = \begin{bmatrix} k(1) & k(2) & k(-1) & k(3) \\ k(-6) & k(3) & k(0) & k(0) \end{bmatrix}$$

Если k = 2, получим:

$$kD = 2D = \begin{bmatrix} 2 & 4 & -2 & 6 \\ -12 & 6 & 0 & 0 \end{bmatrix}$$

Можно сложить две матрицы, но только в том случае, если у них одинаковая размерность. Сумма вычисляется путем сложения соответствующих элементов матриц. Например:

$$A + B = \begin{bmatrix} 1 & 5 \\ -2 & 3 \end{bmatrix} + \begin{bmatrix} 6 & 2 \\ 5 & -8 \end{bmatrix} = \begin{bmatrix} 1+6 & 5+2 \\ -2+5 & 3+(-8) \end{bmatrix} = \begin{bmatrix} 7 & 7 \\ 3 & -5 \end{bmatrix}$$

Аналогично сложению можно выполнять вычитание двух матриц, имеющих одинаковую размерность. Вычитание матриц иллюстрирует следующий пример:

$$A - B = A + (-B) = \begin{bmatrix} 1-6 & 5-2 \\ -2-5 & 3+8 \end{bmatrix} = \begin{bmatrix} -5 & 3 \\ -7 & 11 \end{bmatrix}$$

Умножение

Умножение матриц это наиболее важная операция, которая постоянно используется в трехмерной компьютерной графике. Именно умножение матриц позволяет осуществлять преобразование векторов и комбинировать несколько преобразований в одно.

Чтобы получить произведение матриц AB необходимо чтобы количество столбцов матрицы A было равно количеству строк матрицы B. Если условие выполняется, произведение матриц определено. Рассмотрим представленные ниже матрицы A и B, с размерностью 2 × 3 и 3 × 3 соответственно:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}; B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

Произведение AB определено поскольку количество столбцов матрицы A равно количеству строк матрицы B. Обратите внимание, что произведение BA, получаемое в результате перестановки множителей, не определено, потому что количество столбцов матрицы B не равно количеству строк матрицы A. Это говорит о том, что в общем случае операция умножения матриц не коммутативна (то есть AB ≠ BA). Мы говорим «в общем случае не коммутативна» по той причине, что существует ряд частных случаев в которых операция умножения матриц ведет себя как коммутативная.

После того, как мы узнали в каких случаях произведение матриц определено, можно дать определение операции умножения матриц: если A — это матрица m × n, а B — матрица n × p, то их произведением будет матрица C, размером m × p, в которой элемент cij находится как скалярное произведение i-го вектора-строки матрицы A и j-го вектора-столбца матрицы B:

$$c_{ij} = a_i \cdot b_j$$

В этой формуле a_i обозначает i-ый вектор-строку в матрице A, а b_j — j-ый вектор-столбец матрицы B.

Для примера вычислим произведение:

$$AB = \begin{bmatrix} 4 & 1 \\ -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}$$

Произведение определено, поскольку количество столбцов матрицы A равно количеству строк матрицы B. Кроме того, обратите внимание, что размер полученной в результате матрицы — 2 × 2. Согласно формуле (4) получаем:

$$AB = \begin{bmatrix} 4 & 1 \\ -2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 \\ a_2 \cdot b_1 & a_2 \cdot b_2 \end{bmatrix} = \begin{bmatrix} (4\ \ 1) \cdot (1\ \ 2) & (4\ \ 1) \cdot (3\ \ 1) \\ (-2\ \ 1) \cdot (1\ \ 2) & (-2\ \ 1) \cdot (3\ \ 1) \end{bmatrix}$$
$$= \begin{bmatrix} 6 & 13 \\ 0 & -5 \end{bmatrix}$$

Единичная матрица

Существует особая матрица, называемая единичной матрицей (identity matrix). Это квадратная матрица все элементы которой равны нулю, за исключением тех, что расположены на главной диагонали — эти элементы равны единице. Ниже приведены примеры единичных матриц размером 2 × 2, 3 × 3 и 4 × 4:

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Единичная матрица действует как мультипликативное тождество:

$$MI = IM = M$$

Следовательно, операция умножения на единичную матрицу не изменяет исходную матрицу. Более того, умножение на единичную матрицу это один из случаев, когда операция умножения матриц является коммутативной. Можно думать о единичной матрице как о числе 1 для матриц.

Инвертирование матриц

В математике матриц нет аналога операции деления, но зато есть мультипликативная операция инвертирования. Приведенный ниже список обобщает важные особенности инвертирования:

- Инвертировать можно только квадратные матрицы, так что когда говорят об инвертировании матрицы, подразумевается, что имеют дело с квадратной матрицей.
- В результате инвертирования матрицы M размером n × n получается матрица размером n × n, которую будем обозначать M^{-1}.
- Не всякую квадратную матрицу можно инвертировать.
- Если перемножить исходную и инвертированную матрицы, получится единичная матрица: $MM^{-1} = M^{-1}M = I$. Обратите внимание, что в случае перемножения исходной и инвертированной матриц операция умножения матриц коммутативна.

Инверсия матриц применяется для нахождения искомой матрицы в уравнениях. Для примера возьмем выражение p' = pR и предположим, что нам известны p' и R, а требуется найти p. Сначала вычислим R^{-1} (подразумевается, что эта матрица существует). Получив R^{-1} можно вычислить p по следующему алгоритму:

$$p'^{R^{-1}} = p(RR^{-1})$$
$$p'^{R^{-1}} = pI$$
$$p'^{R^{-1}} = p$$

Описание способа вычисления инвертированной матрицы выходит за рамки этого пособия, но можно его найти в любом учебнике линейной алгебры.

Завершая раздел об инвертировании матриц стоит упомянуть одно полезное свойство, касающееся инвертирования произведения:

$$(AB)^{-1} = B^{-1}A^{-1}$$

Здесь подразумеваеся, что матрицы A и B могут быть инвертированы и что обе они — квадратные матрицы одинакового размера.

Транспонирование матриц

Транспонирование матрицы осуществляется путем перестановки ее строк и столбцов. Следовательно, результатом транспонирования матрицы m × n будет матрица n × m. Результат транспонирования матрицы M будем обозначать M^T.

Транспонируем следующие две матрицы:

$$A = \begin{bmatrix} 2 & -1 & 8 \\ 3 & 6 & -4 \end{bmatrix}; B = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Транспонирование матрицы осуществляется путем перестановки ее строк и столбцов. Следовательно:

$$A^T = \begin{bmatrix} 2 & 3 \\ -1 & 6 \\ 8 & -4 \end{bmatrix}; \; B^T = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

Матрицы в библиотеке D3DX

Программируя приложения Direct3D чаще всего будем использовать матрицы 4 × 4 и векторы-строки 1 × 4. Обратите внимание, что использование матриц двух указанных размеров подразумевает, что определены результаты следующих операций умножения матриц:

- Умножение вектора-строки на матрицу. То есть, если v — это вектор-строка 1 × 4, а T — это матрица 4 × 4, произведение vT определено и представляет собой вектор-строку 1 × 4.
- Умножение матрицы на матрицу. То есть, если T — это матрица 4 × 4 и R — это матрица 4 × 4, произведения TR и RT определены и оба являются матрицами 4 × 4. Обратите внимание, что произведение TR не обязательно равно RT, поскольку операция умножения матриц не коммутативна.

Для представления вектора-строки 1 × 4 в библиотеке D3DX, будем использовать классы векторов D3DXVECTOR3 и D3DXVECTOR4. Конечно, в классе D3DXVECTOR3 только три компоненты, а не четыре. Однако обычно подразумевается что четвертая компонента равна нулю или единице (более подробно это будет обсуждаться в следующем разделе).

Для представления матриц 4 × 4 в библиотеке D3DX, используем класс D3DXMATRIX, определение которого выглядит следующим образом:

```
typedef struct D3DXMATRIX : public D3DMATRIX
{
    public:
        D3DXMATRIX() {};
        D3DXMATRIX(CONST FLOAT*);
        D3DXMATRIX(CONST D3DMATRIX&);
        D3DXMATRIX(FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
                   FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
                   FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
                   FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44);

        // получение элемента
        FLOAT& operator () (UINT Row, UINT Col);
        FLOAT operator () (UINT Row, UINT Col) const;

        // приведение типа
        operator FLOAT* ();
        operator CONST FLOAT* () const;

        // операторы присваивания
        D3DXMATRIX& operator *= (CONST D3DXMATRIX&);
        D3DXMATRIX& operator += (CONST D3DXMATRIX&);
```

```
        D3DXMATRIX& operator -= (CONST D3DXMATRIX&);
        D3DXMATRIX& operator *= (FLOAT);
        D3DXMATRIX& operator /= (FLOAT);

        // унарные операторы
        D3DXMATRIX operator + () const;
        D3DXMATRIX operator - () const;

        // бинарные операторы
        D3DXMATRIX operator * (CONST D3DXMATRIX&) const;
        D3DXMATRIX operator + (CONST D3DXMATRIX&) const;
        D3DXMATRIX operator - (CONST D3DXMATRIX&) const;
        D3DXMATRIX operator * (FLOAT) const;
        D3DXMATRIX operator / (FLOAT) const;

        friend D3DXMATRIX operator * (FLOAT, CONST D3DXMATRIX&);

        BOOL operator == (CONST D3DXMATRIX&) const;
        BOOL operator != (CONST D3DXMATRIX&) const;
} D3DXMATRIX, *LPD3DXMATRIX;
```

Класс D3DXMATRIX наследует элементы данных от простой структуры D3DMATRIX, определенной следующим образом:

```
typedef struct _D3DMATRIX {
    union {
        struct {
            float _11, _12, _13, _14;
            float _21, _22, _23, _24;
            float _31, _32, _33, _34;
            float _41, _42, _43, _44;
        };
        float m[4][4];
    };
} D3DMATRIX;
```

Обратите внимание, что в классе D3DXMATRIX есть десятки полезных операторов для проверки равенства, сложения и вычитания матриц, умножения матрицы на скаляр, преобразования типов и — самое главное — перемножения двух объектов типа D3DXMATRIX. Поскольку умножение матриц так важно, приведем пример кода, использующего этот оператор:

```
D3DXMATRIX A(E);      // инициализация A
D3DXMATRIX B(E);      // инициализация B
D3DXMATRIX C = A * B; // C = AB
```

Другим важным оператором класса D3DXMATRIX являются скобки, позволяющие легко получить доступ к отдельным элементам матрицы. Обратите внимание, что при использовании скобок нумерация элементов матрицы начинается с нуля, подобно нумерации элементов массива в языке C.

Например, чтобы обратиться к верхнему левому элементу матрицы, следует написать:

```
D3DXMATRIX M;
// Присвоить первому элементу матрицы значение 5.0f.
M(0, 0) = 5.0f;
```

Кроме того, библиотека D3DX предоставляет набор полезных функций, позволяющих инициализировать единичную матрицу D3DXMATRIX, транспонировать матрицу D3DXMATRIX и инвертировать матрицу D3DXMATRIX:

```
D3DXMATRIX *D3DXMatrixIdentity(
    D3DXMATRIX *pout   // Матрица, инициализируемая как единичная
  );

D3DXMATRIX M;
D3DXMatrixIdentity(&M);    // M = единичная матрица

D3DXMATRIX *D3DXMatrixTranspose(
    D3DXMATRIX *pOut,       // Результат транспонирования матрицы
    CONST D3DXMATRIX *pM   // Транспонируемая матрица
  );

D3DXMATRIX A(...);            // инициализация A
D3DXMATRIX B;
D3DXMatrixTranspose(&B, &A);  // B = транспонированная(A)

D3DXMATRIX *D3DXMatrixInverse(
    D3DXMATRIX *pOut,    // возвращает результат инвертирования pM
    FLOAT *pDeterminant,// детерминант, если необходим, иначе 0
    CONST D3DXMATRIX *pM // инвертируемая матрица
  );
```

Функция инвертирования возвращает NULL, если переданная ей матрица не может быть инвертирована. Кроме того, в этой книге мы игнорируем второй параметр и всегда передаем в нем 0.

```
D3DXMATRIX A(...);         // инициализация A
D3DXMATRIX B;
D3DXMatrixInverse(&B, 0, &A); // B = инвертированная(A)
```

Примечание: Математические операции над матрицами в OpenGL придется писать самостоятельно. Либо использовать готовые функции для преобразований векторов.

Матрицы в библиотеке OpenGL

Матрица в OpenGL представляет собой вектор, содержащий 16 величин (m_1,m_2,…,m_{16}), задающих матрицу M следующим образом:

$$M = \begin{bmatrix} m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \\ m_4 & m_8 & m_{12} & m_{16} \end{bmatrix}$$

Если вы хотите загрузить в качестве текущей некоторую специфическую матрицу (например, чтобы задать какую-либо особенную проекцию или преобразование), используйте glLoadMatrix{fd}v(). Похожим образом используйте glMultMatrix{fd}v() для умножения текущей матрицы на матрицу, переданную в качестве аргумента. Для установления текущей матрицы в единичную, используйте команду glLoadIdentity().

Все перемножения матриц в OpenGL производятся следующим образом. Обозначим текущую матрицу как C, а матрицу, заданную командой glMultMatrix*() или любой командой преобразования как M. После умножения результирующей матрицей всегда является CM. Поскольку в общем случае произведение матриц не коммутативно, порядок перемножения матриц имеет большое значение.

Матрицы в OpenGL и DirectX являются транспонированным отображением одна в другую. Т.е. матрица $M_{DX} = M_{OGL}^{T}$.

Основные преобразования

Создавая использующие Direct3D программы, для представления преобразований будем применять матрицы 4 × 4. Идея заключается в следующем: инициализируем элементы матрицы X размером 4 × 4 таким образом, чтобы они описывали требуемое преобразование. Затем мы помещаем координаты точки или компоненты вектора в столбцы вектора-строки v размером 1 × 4. Результатом произведения vX будет новый преобразованный вектор v'. Например, если матрица X представляет перемещение на 10 единиц вдоль оси X, и v = [2, 6, –3, 1], произведение vX = v' = [12, 6, –3, 1].

Следует пояснить несколько моментов. Используем матрицы размера 4 × 4 по той причине, что они позволяют представить все необходимые нам преобразования. На первый взгляд матрицы размером 3 × 3 кажутся более подходящими для трехмерной графики. Однако, с их помощью нельзя представить ряд преобразований, которые могут нам потребоваться, таких как перемещение, перспективная проекция и отражение. Помните, что мы работаем с произведением вектора на матрицу и при выполнении преобразований ограничены правилами умножения матриц. Дополнение матрицы до размера 4 × 4 позволяет нам с помощью матрицы описать большинство преобразований и при этом произведение вектора на матрицу будет определено.

Координаты точки или компоненты вектора будем хранить в столбцах вектора-строки размером 1 × 4. Но наши точки и векторы — трехмерные! Зачем же использовать вектор-строку 1 × 4? Необходимо дополнить трехмерные точки/векторы до четырехмерного вектора-строки 1 × 4 чтобы был определен результат умножения вектора на матрицу; произведение вектора-строки 1 × 3 и матрицы 4 × 4 не определено.

Так какое же значение использовать для четвертой компоненты, которую, кстати, мы будем обозначать w? Когда вектор-строка 1 × 4 используется для

представления точки, значение w будет равно 1. Это позволяет корректно выполнять перемещение точки. Поскольку вектор не зависит от местоположения, операция перемещения векторов не определена и результат попытки переместить вектор не имеет смысла. Чтобы предотвратить перемещение векторов, помещая компоненты вектора в вектор-строку 1 × 4, присваиваем компоненте w значение 0. Например, точка p = (p_1, p_2, p_3), помещенная в вектор-строку 1 × 4 будет выглядеть как [p_1, p_2, p_3, 1], а вектор v = (v_1, v_2, v_3), помещенный в вектор-строку 1 × 4 будет выглядеть как [v_1, v_2, v_3, 0].

Примечание: Дополненный четырехмерный вектор называется однородным вектором (homogenous vector) и, поскольку однородный вектор может описывать и точки и векторы, мы будем использовать термин «вектор», подразумевая, что он может относиться как к точке, так и к вектору.

Иногда в результате преобразований компонента w вектора будет меняться таким образом, что w ≠ 0 и w ≠ 1. Рассмотрите следующий пример:

$$p = [p_1, p_2, p_3, 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = [p_1, p_2, p_3, p_3] = p'$$

для $p_3 \neq 0$ и $p_3 \neq 1$.

Обратите внимание, что $w = p_3$. Когда w ≠ 0 и w ≠ 1, мы говорим, что у нас есть вектор в однородном пространстве (homogeneous space), в противоположность векторам в трехмерном пространстве. Мы можем отобразить вектор в однородном пространстве обратно на трехмерное пространство, разделив каждую компоненту вектора на значение компоненты w. Например, для отображения вектора (x, y, z, w) в однородном пространстве в трехмерный вектор x, выполним следующие операции:

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, \frac{w}{w}\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right) = x$$

Переход к однородному пространству и обратное отображение векторов в трехмерное пространство используются в программировании трехмерной графики для выполнения перспективной проекции.

Примечание: Когда записываем точку (x, y, z) в виде (x, y, z, 1) мы фактически описываем наше трехмерное пространство как плоскость в четырехмерном пространстве, а именно четырехмерную плоскость w = 1. (Обратите внимание, что плоскость в четырехмерном пространстве является трехмерным пространством, точно так же как плоскость в трехмерном пространстве является двухмерным пространством). Таким образом, присваивая w какое-нибудь другое значение, мы перемещаемся с полскости w = 1. Чтобы вернуться на эту плоскость, которая соответствует нашему трехмерному пространству, мы выполняем обратную проекцию путем деления каждой компоненты на w.

Матрица перемещения

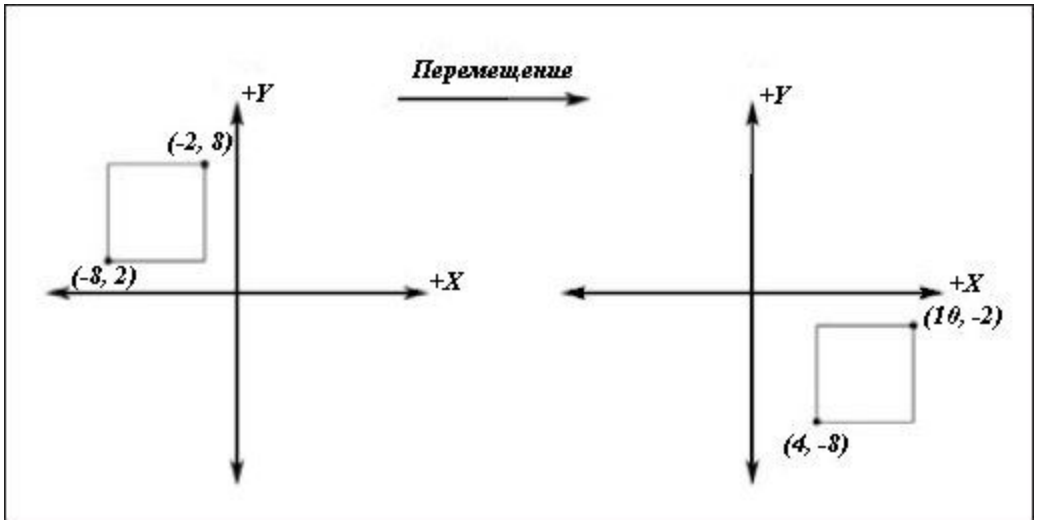

Рисунок 10

На рис. 10 изображено перемещение на 12 единиц по оси X и на –10 единиц по оси Y.

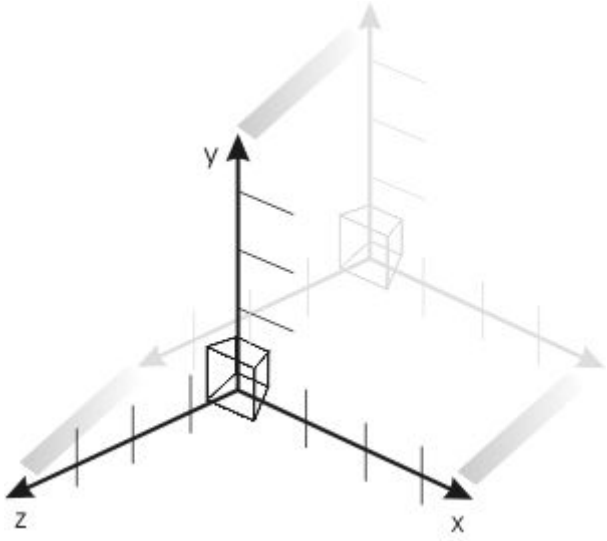

Рисунок 11

Мы можем переместить вектор (x, y, z, 1) на px единиц по оси X, py единиц по оси Y и pz единиц по оси Z умножив его на следующую матрицу:

$$T(p) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p_x & p_y & p_z & 1 \end{bmatrix}$$

Для создания матрицы перемещения в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixTranslation(
     D3DXMATRIX* pOut, // Результат
     FLOAT x,     // Количество единиц для перемещения по оси X
     FLOAT y,     // Количество единиц для перемещения по оси Y
     FLOAT z      // Количество единиц для перемещения по оси Z
```

```
);
```

В OpenGL функция

```
void glTranslate{fd} (TYPE x, TYPE y, TYPE z);
```

Умножает текущую матрицу на матрицу, передвигающую (переносящую) объект на расстояния x, y, z, переданные в качестве аргументов команды, по соответствующим осям (или перемещает локальную координатную систему на те же расстояния).

Инверсия матрицы перемещения получается путем простой смены знака компонент вектора перемещения p.

$$T^{-1} = T(-p) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -p_x & -p_y & -p_z & 1 \end{bmatrix}$$

Матрицы вращения

Поворот на 30 градусов против часовой стрелки вокруг оси Z

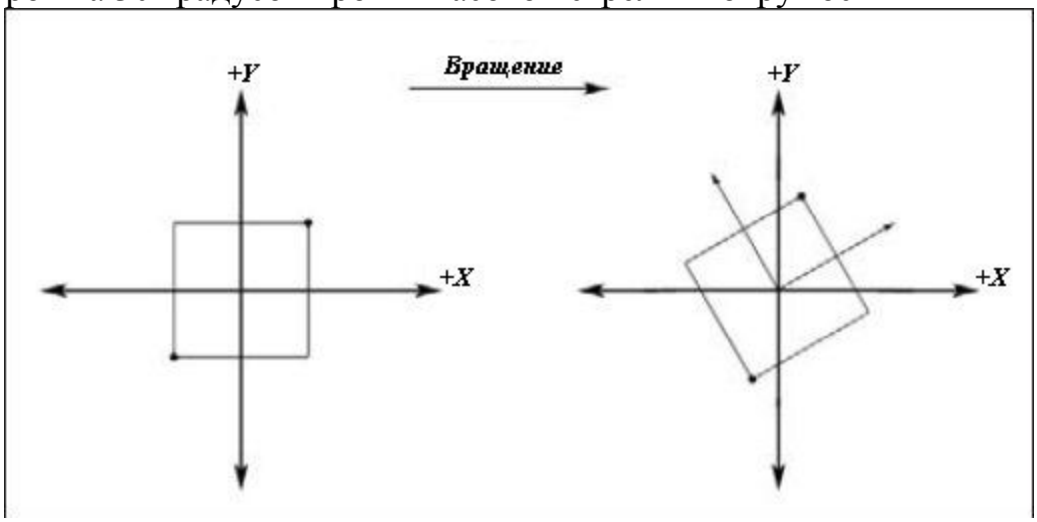

Рисунок 12

Используя приведенные ниже матрицы мы можем повернуть вектор на φ радиан вокруг осей X, Y или Z. Обратите внимание, что если смотреть вдоль оси вращения по направлению к началу координат, то углы измеряются по часовой стрелке.

$$X(\vartheta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\vartheta & \sin\vartheta & 0 \\ 0 & -\sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Для создания матрицы вращения вокруг оси X в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixRotationX(
     D3DXMATRIX* pOut, // Результат
     FLOAT Angle       // Угол поворота в радианах
```

```
);
```

$$Y(\vartheta) = \begin{bmatrix} \cos\vartheta & 0 & -\sin\vartheta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\vartheta & 0 & \cos\vartheta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Для создания матрицы вращения вокруг оси Y в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixRotationY(
     D3DXMATRIX* pOut, // Результат
     FLOAT Angle       // Угол поворота в радианах
);
```

$$Z(\vartheta) = \begin{bmatrix} \cos\vartheta & \sin\vartheta & 0 & 0 \\ -\sin\vartheta & \cos\vartheta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Для создания матрицы вращения вокруг оси Z в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixRotationZ(
     D3DXMATRIX* pOut, // Результат
     FLOAT Angle       // Угол поворота в радианах
);
```

Инверсией матрицы вращения R является результат транспонирования этой матрицы: $R^T = R^{-1}$. Такие матрицы называются ортогональными.

Для поворота в OpenGL используется функция

```
void glRotate{fd} (TYPE angle, TYPE x, TYPE y, TYPE z);
```

которая умножает текущую матрицу на матрицу, которая поворачивает объект (или локальную координатную систему) в направлении против часовой стрелки вокруг луча из начала координат, проходящего через точку (x, y, z). Параметр angle задает угол поворота в градусах.

Результат выполнения glRotatef(45.0, 0.0,0.0,1.0), то есть поворот на 45 градусов вокруг оси Z, показан на рис. 13.

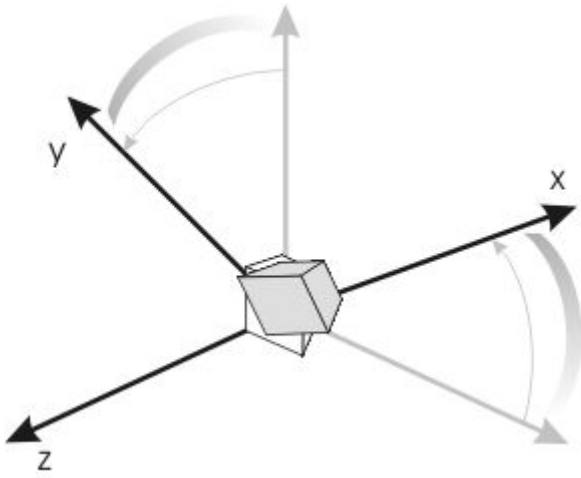

Рисунок 13

Стоит обратить внимание, на то, что из-за разницы в системе координат (OpenGL – правосторонняя, DirectX - левосторонняя) направление поворота различно. А также на то, что угол в DX задается в радианах, а в OGL – в градусах.

Масштабирование.

Масштабирование с коэффициентом 1/2 по оси X и коэффициентом 2 по оси Y

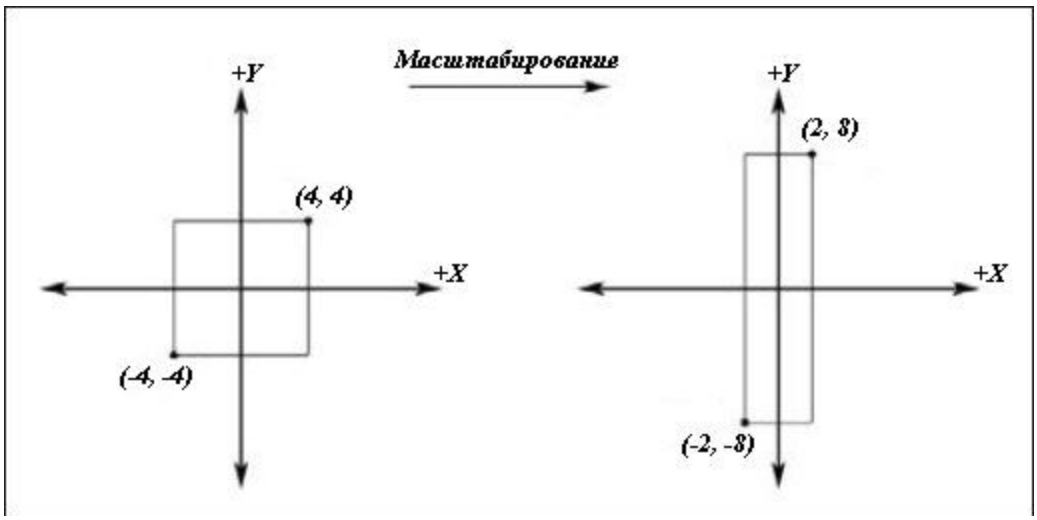

Рисунок 14

Мы можем масштабировать вектор с коэффициентом q_x по оси X, коэффициентом q_y по оси Y и коэффициентом q_z по оси Z, умножив его на следующую матрицу:

$$S(q) = \begin{bmatrix} q_x & 0 & 0 & 0 \\ 0 & q_y & 0 & 0 \\ 0 & 0 & q_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Для создания матрицы масштабирования в библиотеке D3DX используется следующая функция:

```
D3DXMATRIX *D3DXMatrixScaling(
     D3DXMATRIX* pOut, // Результат
     FLOAT sx,         // Коэффициент масштабирования по оси X
     FLOAT sy,         // Коэффициент масштабирования по оси Y
     FLOAT sz          // Коэффициент масштабирования по оси Z
);
```

Чтобы инвертировать матрицу масштабирования надо взять обратную дробь для каждого коэффициента масштабирования:

$$S^{-1} = S\left(\frac{1}{q_x}, \frac{1}{q_y}, \frac{1}{q_z}\right) = \begin{bmatrix} \frac{1}{q_x} & 0 & 0 & 0 \\ 0 & \frac{1}{q_y} & 0 & 0 \\ 0 & 0 & \frac{1}{q_z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Масштабирование в OpenGL производится при помощи функции

```
void glScale{fd} (TYPE x, TYPE y, TYPE z);
```

которая умножает текущую матрицу на матрицу, которая растягивает, сжимает или отражает объект вдоль координатный осей. Каждая x -, y- и z- координата каждой точки объекта будет умножена на соответствующий аргумент x, y или z команды glScale*(). При рассмотрении преобразования с точки зрения локальной координатной системы, оси этой системы растягиваются, сжимаются или отражаются с учетом факторов x, y и z, и ассоциированный с этот системой объект меняется вместе с ней.

На рисунке 15 показан эффект команды glScalef(-2.0,0.5,1.0);

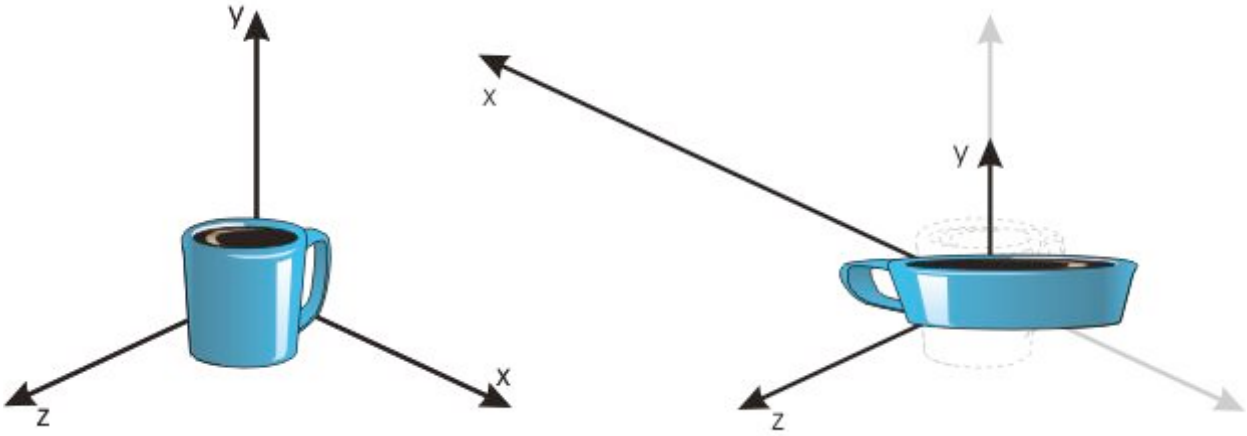

Рисунок 15

Комбинирование преобразований

Часто будем применять к векторам целую последовательность преобразований. Например, можем масштабировать вектор, затем повернуть его и потом переместить в требуемую позицию.

В качестве примера мы рассмотрим вектор p = [5, 0, 0, 1], который масштабируем по всем осям с коэффициентом 1/5, затем повернем его на π/4 радиан вокруг оси Y и, наконец, переместим на 1 единицу по оси X, 2 единицы по оси Y и –3 единицы по оси Z.

Обратите внимание, что мы должны выполнить масштабирование, поворот вокруг оси Y и перемещение. Мы инициализируем наши матрицы преобразований S, R_y, T для масштабирования, поворота и перемещения соответственно, следующим образом:

$$S\left(\frac{1}{5},\frac{1}{5},\frac{1}{5}\right)=\begin{bmatrix}\frac{1}{5} & 0 & 0 & 0\\ 0 & \frac{1}{5} & 0 & 0\\ 0 & 0 & \frac{1}{5} & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$R_y\left(\frac{\pi}{4}\right)=\begin{bmatrix}.707 & 0 & -.707 & 0\\ 0 & 1 & 0 & 0\\ .707 & 0 & .707 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

$$T(1,2,-3)=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 1 & 2 & -3 & 1\end{bmatrix}$$

Применив последовательность преобразований в заданном порядке — масштабирование, вращение, перемещение — получим:

$pS = [1,0,0,1] = p'$

$p'^{R_y} = [.707,0,-.707,1] = p''$

$p''T = [1.707,2,-3.707,1]$

Ключевым преимуществом использования матриц является возможность использования умножения матриц для комбинирования нескольких преобразований в одной матрице. Вернемся к примеру, рассматриваемому в начале данного раздела. Давайте с помощью умножения матриц скомбинируем все три матрицы преобразований в одну, которая будет представлять все три преобразования сразу. Обратите внимание, что порядок, в котором мы умножаем матрицы должен соответствовать порядку применения отдельных преобразований.

$$SR_yT = \begin{bmatrix} \frac{1}{5} & 0 & 0 & 0 \\ 0 & \frac{1}{5} & 0 & 0 \\ 0 & 0 & \frac{1}{5} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} .707 & 0 & -.707 & 0 \\ 0 & 1 & 0 & 0 \\ .707 & 0 & .707 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 2 & -3 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} .1414 & 0 & -.1414 & 0 \\ 0 & 1 & 0 & 0 \\ .1414 & 0 & .1414 & 0 \\ 1 & 2 & -3 & 1 \end{bmatrix} = Q$$

В результате получаем pQ = [1.707, 2, –3.707, 1].

Возможность комбинировать преобразования значительно увеличивает производительность. Представьте, что вам необходимо применить одну и ту же последовательность преобразований масштабирования, вращения и перемещения к большому набору векторов (это обычная задача в трехмерной компьютерной графике). Вместо того, чтобы применять к каждому вектору последовательность преобразований, как мы делали это в первом примере, можно скомбинировать все три преобразования в одной матрице. После этого остается только умножить каждый вектор на единственную матрицу, содержащую комбинацию всех трех преобразований. Благодаря этому количество выполняемых операций умножения вектора на матрицу значительно сокращается.

В OpenGL именно поэтому функции трансформаций не заменяют, а помножают на текущую матрицу.

Некоторые функции для преобразования векторов

Библиотека D3DX предоставляет две функции для преобразования точек и векторов соответственно. Функция D3DXVec3TransformCoord используется для преобразования точек и предполагает, что четвертая компонента вектора равна 1. Функция D3DXVec3TransformNormal используется для преобразования векторов и предполагает, что четвертая компонента вектора равна 0.

```
D3DXVECTOR3 *D3DXVec3TransformCoord(
     D3DXVECTOR3* pOut,     // Результат
     CONST D3DXVECTOR3* pV, // Преобразуемая точка
     CONST D3DXMATRIX* pM   // Матрица преобразования
);

D3DXMATRIX T(...);   // инициализация матрицы преобразований
D3DXVECTOR3 p(...);  // инициализация точки
D3DXVec3TransformCoord(&p, &p, &T); // преобразование точки

D3DXVECTOR3 *D3DXVec3TransformNormal(
     D3DXVECTOR3 *pOut,     // Результат
     CONST D3DXVECTOR3 *pV, // Преобразуемый вектор
     CONST D3DXMATRIX *pM   // Матрица преобразования
```

```
);

D3DXMATRIX T(...);   // инициализация матрицы преобразований
D3DXVECTOR3 v(...);  // инициализация вектора
D3DXVec3TransformNormal(&v, &v, &T); // преобразование вектора
```

Примечание: Библиотека D3DX также предоставляет функции D3DXVec3TransformCoordArray и D3DXVec3TransformNormalArray для преобразования массива точек и массива векторов соответственно.

Матричный стек в OpenGL

Так как в OpenGL нет понятия матрицы как отдельного объекта, а есть понятие только текущей матрицы. То в OGL используется механизм матричного стека.

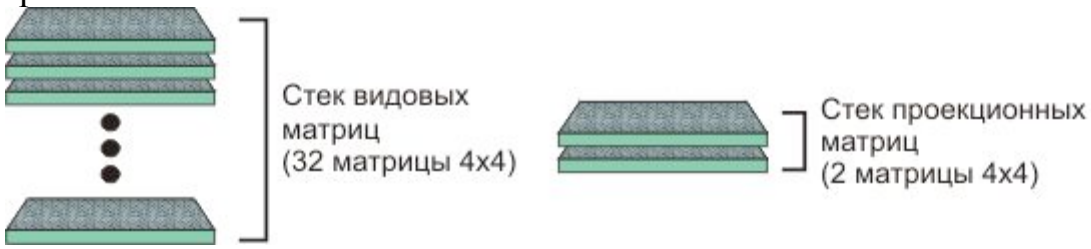

Рисунок 16

Матричный стек полезен для построения иерархических моделей, в которых сложные объекты конструируются при помощи простых. Предположим, например, что вы рисуете автомобиль, у которого 4 колеса и каждое колесо прикреплено к автомобилю пятью болтами. У вас имеется функция, рисующая колесо и функция, рисующая болт, так как все колеса и болты выглядят одинаково, то каждая из этих функций рисует колесо или болт в четко определенном месте, скажем в начале координат, и с определенной ориентацией, например, с центральной осью объекта, совпадающей с отрицательным направлением оси z. Когда вы рисуете машину, включая колеса и болты, вы захотите вызвать функцию, рисующую колесо 4 раза с применением различных трансформаций для правильного позиционирования каждого колеса. При рисовании каждого колеса вы захотите нарисовать болт пять раз, каждый раз перенося болт в нужное место относительно колеса.

На секунду предположим, что все, что вы хотите сделать – это нарисовать корпус машины и колеса. В этом случае алгоритм процесса может быть описан следующим образом:

- Нарисовать корпус машины.
- Запомнить, где мы находимся, и выполнить перенос к переднему левому колесу.
- Нарисовать колесо и отбросить последний перенос, чтобы вернуться в начало координат относительно корпуса машины.
- Запомнить, где мы находимся, и выполнить перенос к левому заднему колесу...

Похожим образом для каждого колеса, нам следует нарисовать его, запомнить, где мы, и последовательно выполнять переносы к позиции каждого болта, отбрасывая преобразования после того, как каждый болт нарисован.

Поскольку преобразования сохраняются в матрицах, матричный стек предоставляет идеальный механизм для подобного рода запоминаний, переносов и отбрасываний. Все ранее описанные матричные операции (glLoadMatrix(), glMultMatrix(), glLoadIdentity() и команды, создающие специфические матрицы) работают с текущей матрицей, то есть с верхней матрицей стека. С помощью команд управления стеком вы можете управлять тем, какая матрица находится на вершине стека: glPushMatrix() копирует текущую матрицу и добавляет копию на вершину матричного стека, glPopMatrix() уничтожает верхнюю матрицу в стеке (см. рисунок ниже). (Помните, что текущей матрицей всегда является матрица на вершине). Говоря проще, glPushMatrix() означает «запомнить, где мы находимся», а glPopMatrix() – «вернуться туда, где мы были».

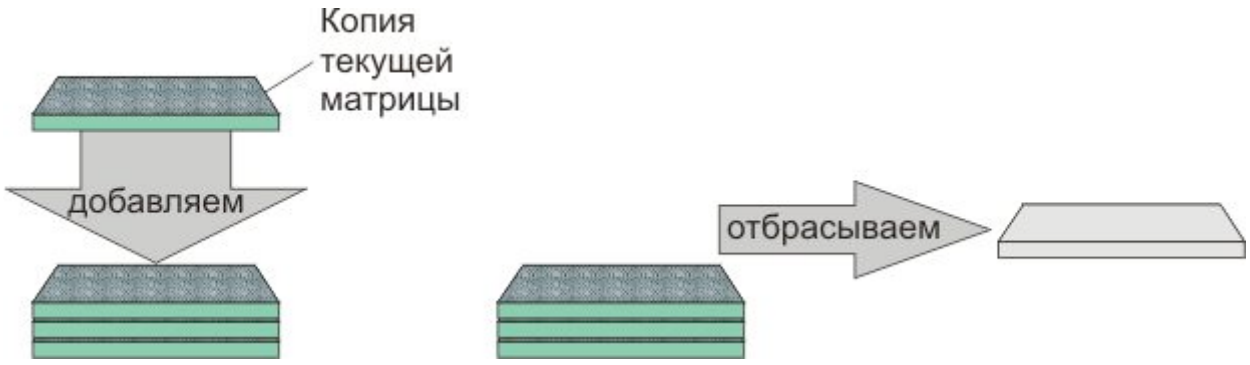

Рисунок 17

```
void glPushMatrix (void);
```

Опускает все имеющиеся в текущем стеке матрицы на один уровень. То, какой стек является текущим, задается с помощью вызова glMatrixMode(). Верхняя матрица при этом копируется, таким образом, ее содержимое продублировано в верхней и второй сверху матрице стека. Если добавлено слишком много матриц, будет сгенерирована ошибка.

```
void glPopMatrix (void);
```

Выкидывает верхнюю матрицу из стека, тем самым, уничтожая ее содержимое. Верхней (и, как следствие, текущей) становится матрица, которая занимала второе сверху место в стеке. Текущий стек задается командой glMatrixMode(). Если стек содержит только одну матрицу, вызов glPopMatrix() сгенерирует ошибку.

Итоги

Векторы используются для моделирования физических величин, которые характеризуются величиной и направлением. Геометрическим представлением вектора является направленный отрезок прямой. Когда вектор находится в

стандартной позиции его начало совпадает с началом координат. Вектор в стандартной позиции описывается путем указания координат конца вектора.

Мы можем использовать матрицы 4 × 4 для представления преобразований и однородные векторы 1 × 4 для описания точек и векторов. В результате умножения вектора-строки 1 × 4 на матрицу преобразования 4 × 4 получается новый преобразованный вектор-строка 1 × 4. Можно скомбинировать несколько матриц преобразований в одну перемножив их друг на друга.

Для представления векторов и точек мы используем однородные четырехмерные векторы. Для вектора значение компоненты w равно 0, а для точки значение компоненты w равно 1. Если $w \neq 0$ и $w \neq 1$, то у нас есть вектор (x, y, z, w) в однородном пространстве, который может быть отображен обратно на трехмерное пространство путем деления каждой его компоненты на w;

$$\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, \frac{w}{w}\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

Плоскости делят трехмерное пространство на две части: положительное полупространство перед плоскостью и отрицательное полупространство за ней. Плоскости применяются для проверки местоположения точек относительно них (другими словами, чтобы проверить в каком полупространстве относительно данной плоскости находится точка).

Лучи описываются путем указания начальной точки и вектора направления. Лучи полезны для моделирования различных физических явлений, таких как луч света или полет снаряда (например, пули или ракеты) по прямолинейной траектории.

СПИСОК РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

1. ARB, Dave Shreiner «OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, Version 2», 5-е изд. Addison-Wesley Professional, 2005.
2. DirectX SDK Documentation, 2008 / http://msdn.microsoft.com/en-us/library/aa139765.aspx
3. OpenGL 3.0 Specification, 2008 / http://www.opengl.org/registry/doc/glspec30.20080811.pdf
4. OpenGL Shading Language Specification 1.30, 2008 / http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.30.08.withchanges.pdf
5. GPU Gems / http://developer.download.nvidia.com/books/HTML/gpugems/gpugems_copyrightpg.html
6. GPU Gems 2 / http://http.developer.nvidia.com/GPUGems2/gpugems2_part01.html
7. Эдвард Энджел «Интерактивная компьютерная графика. Вводный курс на базе OpenGL» 2-е изд. Вильямс, 2001
8. Д. Гончаров, Т. Салихов «DirectX 7.0 для программистов». Питер, 2001.
9. Ю. Тихомиров «Программирование трехмерной графики». BHV - Санкт – Петербург, 1998.

СПИСОК ТЕРМИНОВ

Anisotropic Filtering[8]

При текстурировании для конкретного пикселя на мониторе из текстуры делаем выборку при помощи окружности (point sampling, Биллинейная фильтрация, Трилинейная фильтрация).

Таким образом эти фильтрации дают хороший результат когда текстурируемый полигон перпендикулярен взгляду камеры. Если же на него смотрят под углом то возникают искажения. AF заключается в том что мы учитываем нашу трехмерность и используем эллипс вместо окружности.

Antialiasing[9]

Устранение контурных неровностей, дефектов изображения. Способ обработки (интерполяции) пикселей для получения более четких краев (границ) изображения (объекта). Наиболее часто используемая техника для создания плавного перехода от цвета линии или края к цвету фона. В некоторых случаях результатом является смазывание (blurring) контуров объектов. Т.к. монитор представляет из себя сетку пикселей, линии отличные от вертикальных и горизонтальных получаются ступенчатыми. Чтобы сгладить этот эффект применяются различные техники сглаживания. Две основные: Supersampling и Multisampling.

Идея этих двух техник в том, что изображение рисуется в буфер кадра, в увеличенном в N раз разрешении. Таким образом, для каждого пикселя в результирующем изображении соответствует N суб-пиксельных значений цвета (сэмплов). После того, как вывод в буфер окончен, он "сжимается" с усреднением цветовых значений сэмплов. Таким образом, получается сглаженная сцена.

Supersampling(SSAA) является самой простой в реализации техникой и полностью следует исходной идее. Он также является самым ресурсоемким методом реализации AA.

Multisampling(MSAA) немного изменяет подход к тому, как рассчитывается цвет отдельных сэмплов одного пикселя. В SSAA расчёт цвета во фрагментном шейдере исполняется для каждого сэмпла в отдельности. В MSAA фрагментный шейдер исполняется только один раз. Таким образом, видимые изменения происходят только на границах объектов, но не внутри них. Получается более быстрый, но в некоторых случаях менее качественный AA. Это видно в отсутствии сглаживания на треугольниках, у которых некоторые фрагменты внутри отброшены тестом альфы или фрагментным шейдером (discard\clip).

Сэмплы в буфере могут быть связаны с разными физическими положениями внутри пикселя, не смотря на то, что в буфере они, скорее всего,

[8] http://www.gamedev.ru/terms/AnisotropicFiltering
[9] http://www.gamedev.ru/terms/Antialiasing

расположены в регулярном порядке. Можно отметить следующие способы расположения:

Orientied Grid Super Sampling - сэмплы располагаются на обычной, регулярной сетке.

Rotated Grid Super Sampling - сэмплы располагаются на повёрнутой сетке. Этом метод дает особенно хороший результат на близких к горизонтальным и вертикальным линиях.

Современные видеокарты имеют более свободную привязку положений к сэмплам, к примеру, карты АТИ в некоторых режимах качества могут располагать позиции даже за пределами самого пикселя.

API[10]

Интерфейс программирования приложений (англ. Application Programming Interface, API; по-русски чаще произносят [апи]) — набор методов (функций), который программист может использовать для доступа к функциональности программного компонента (программы, модуля, библиотеки). API является важной абстракцией, описывающей функциональность «в чистом виде», безотносительно того, как реализована эта функциональность.

Программные компоненты взаимодействуют друг с другом посредством API. При этом обычно компоненты образуют иерархию — высокоуровневые компоненты используют API низкоуровневых компонент, а те, в свою очередь, используют API ещё более низкоуровневых компонент.

Vertex Buffer Object

ARB_vertex_buffer_object - это расширение OpenGL, позволяющее работать с видеопамятью в части чтения/записи/рендеринга массивов вертексов (вершин) и индексов. Поддерживается практически на всех 3D-ускорителях (все GeForce, Radeon, Quadro, FireGL, Riva TNT, большая часть интегрированных чипсетов Intel, часть карт S3 и 3DLabs), кроме самых старых (таких, как 3dfx Voodoo).

Это расширение позволяет, в частности, размещать геометрию в видеопамяти для последующего (многократного) рендеринга или же организовывать потоковую передачу данных видеокарте (пересылка данных + рендеринг их в каждом кадре).

VBO позволяет избежать передачи больших массивов данных через шину видеокарты в каждом кадре, так как данные хранятся локально на видеокарте. Кроме того, видеопамять на современных видеокартах обычно быстрее системной (system) памяти.

На современных видеокартах VBO обеспечивает очень большое увеличение производительности (наблюдалось, например, ускорение в 1000 раз на GeForce 6800, на статической геометрии). Использовать VBO рекомендуется практически всегда, проще назвать причины, когда не следует его

[10] http://ru.wikipedia.org/wiki/Интерфейс_программирования_приложений

использовать: это либо очень маленькие массивы данных (неэффективно), либо очень большие (когда они не помещаются в свободную видеопамять).

Z-buffer[11]

Один из методов отсечения невидимой геометрии, который широко используется для построения изображений в ускорителях 3д графики. Данный метод создаёт дополнительный буфер, с данными для каждого фрагмента изображения.

Буфер глубины может иметь различную разрядность данных:

- *16 бит*. Устаревший формат с невысокой точностью. На современных видеокартах используется редко, например для рендера в текстуру глубины на картах АТИ Радеон 9***, которые не поддерживают получения данных буфера глубины более высокой разрядности.
- *24 бита*. Можно сказать стандартный тип данных буфера глубины, в основном используется он.
- *32 бита*. Формат ещё более высокой точности, но совершенно не распространён, не стоит рассчитывать на его поддержку.

Проверка видимости этим методом на современных видеокартах может проводиться в двух местах:

- До выполнения фрагментного шейдера (early Z test), если фрагментный шейдер не изменяет глубину фрагмента (впрочем, вполне могут быть и другие условия, по причине которых early Z test проводиться не будет). Дополнительно к этому, современные видеокарты могут отбрасывать сразу целые блоки фрагментов, что ещё более повышает производительность.
- После выполнения фрагментного шейдера.

Проверка производиться с использованием значения глубины текущего фрагмента в постпроективном пространстве и предыдущего значения, хранящегося в буфере глубины. Сравнение происходит с использованием функции, выбранной пользователем. Доступны следующие функции:

```
Действие     | Название в DirectX   |  Название в OpenGL
-----------------------------------------------------
false        |  D3DCMP_NEVER        |  GL_NEVER
Znew <  Zold |  D3DCMP_LESS         |  GL_LESS
Znew == Zold |  D3DCMP_EQUAL        |  GL_EQUAL
Znew <= Zold |  D3DCMP_LESSEQUAL    |  GL_LEQUAL
Znew >  Zold |  D3DCMP_GREATER      |  GL_GREATER
Znew != Zold |  D3DCMP_NOTEQUAL     |  GL_NOTEQUAL
Znew >= Zold |  D3DCMP_GREATEREQUAL |  GL_GEQUAL
true         |  D3DCMP_ALWAYS       |  GL_ALWAYS
```

Где Znew значение глубины фрагмента и Zold предыдущее значение глубины. Функция сравнения устанавливается функциями `IDirect3DDevice::SetRenderState(D3DRS_ZFUNC, func)` в DirectX, и `glDepthFunc(func)` в OpenGL. Также для работы следует включить тест

[11] http://www.gamedev.ru/terms/ZBuffer

глубины функциями `IDirect3DDevice::SetRenderState(D3DRS_ZENABLE, true)` в DirectX, и `glEnable(GL_DEPTH_TEST)` в OpenGL. После того как тест пройден, новое значение записывается в буфер глубины.

Значение глубины точки можно рассчитать при помощи следующей формулы:

$$z' = \frac{zfar + znear}{zfar - znear} + \frac{1 * (-2 * zfar * znear)}{z * (zfar - znear)}$$

Полученное значение z' затем нормализуется в диапазон [0; 1], где 0 - у ближней плоскости отсечения, и 1 - у дальней.

Значения глубины точки не находятся в линейной зависимости от расстояния до камеры. Значения у ближней плоскости расположены очень плотно, а с увеличением дальности к дальней плоскости отсечения всё более редко. По этой причине нужно следить за расстояниями, на которых вы ставите плоскости отсечения: ближнюю плоскость не стоит ставить слишком близко! В противном случае можно получить множество артефактов изображения вдали, в особенности z-fightning, когда небольшое изменение ракурса камеры может изменить видимость близко расположенных треугольников.