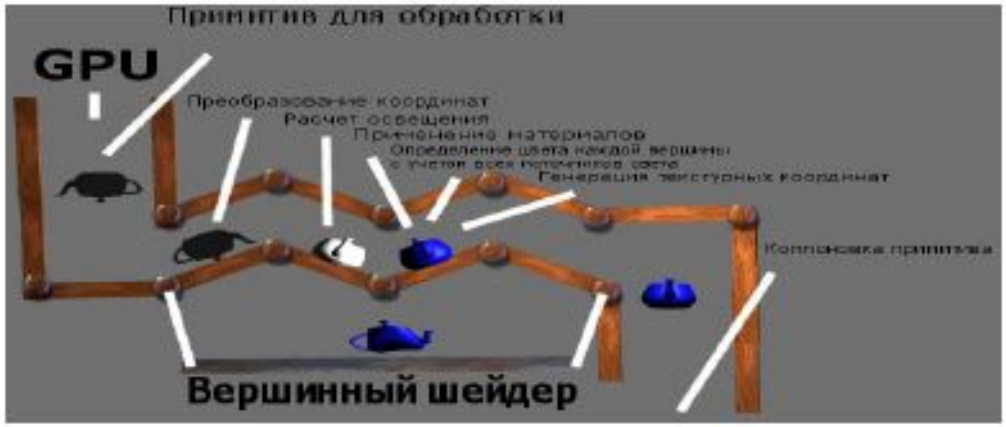
3D-сцена и графический конвейер

Геометрическая стадия (Шейдеры п.п.3-5)

Исходные данные:
Сцена (геометрия, цвет, текстура)
Свет
Камера (Frustum, View Volume).



1. **Wireframe** (Каркасное) моделирование поверхности объектов с учетом видимого объема (Camera, Frustum, View Volume). Формирование списка отображаемых объектов.

2. **Tesselation**. Тесселяция или триангуляция (triangulation): разбиение поверхности на плоские полигональные элементы. Вместо криволинейной поверхности – **полигональная модель**, представленная вершинами (vertex)

3. **Transformation** (трансформация) : перемещение, изменение формы посредством матричных преобразований вершин в пределах видимого объема

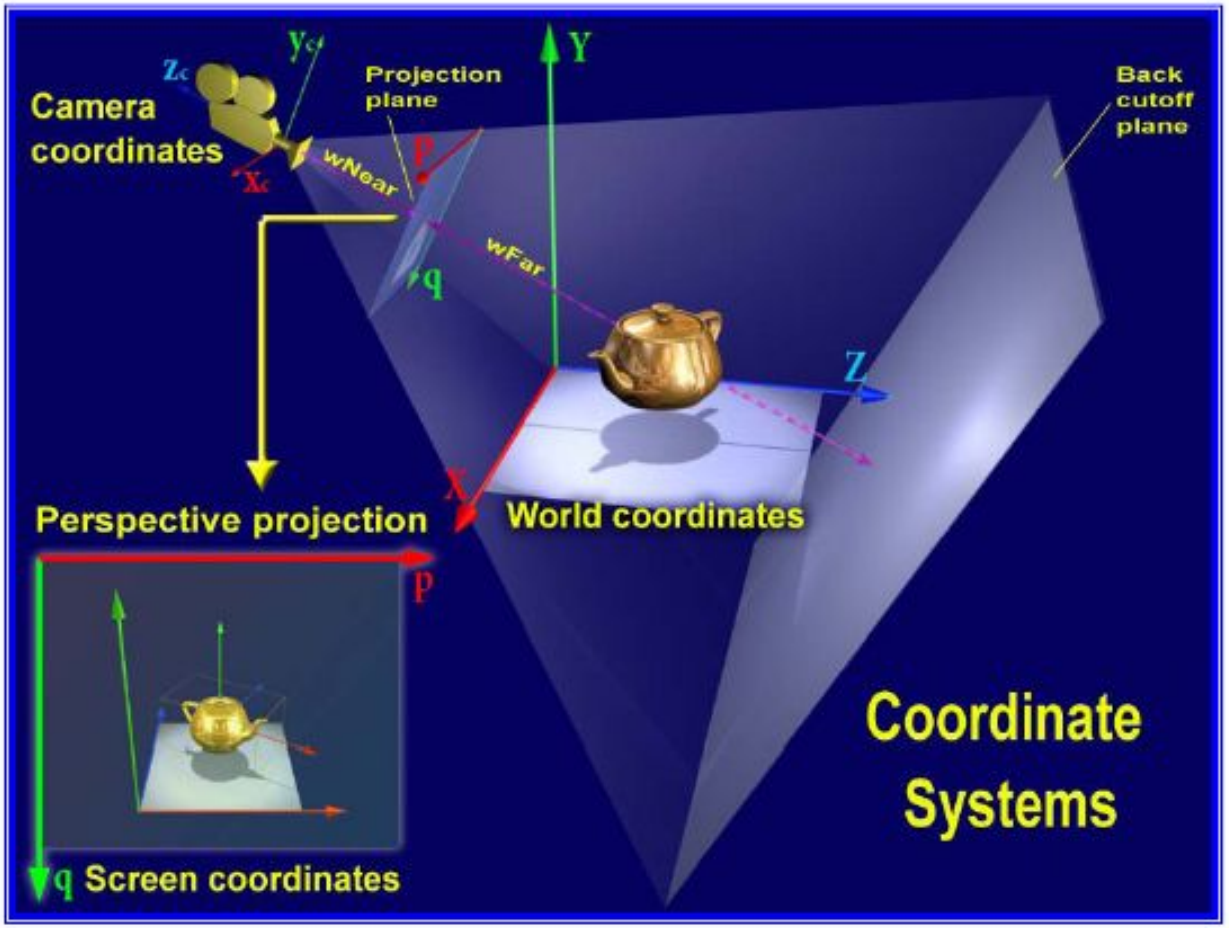
4. **Lighting**. Расчет освещенности и затенения (shading) объектов по вершинам. Методы интерполяции для полигональных поверхностей: Гуро (Gouraud shading), Фонга (Phong shading), применение вершинных шейдеров

5. **Camera-ViewPort**. Проецирование 3D-объекта с сохранением информации о расстоянии (о глубине) каждой из вершин до плоскости проекции

6. **Triangle setup**. Подготовка (компоновка) треугольников объекта: генерация текстурных координат; сортировка вершин; отбор и отбрасывание нелицевых граней (culling)

Место преобразований координат в графическом конвейере

- Преобразование Model → Wold определяются **матрицей вида** Model
- Положение и направление камеры определяют преобразование Wold → Camera, **матрицу** View
- Способ проецирования 3D-сцены на проекционную плоскость камеры определяется **матрицей проецирования** Projection
- В окне отображается проекция 3D-сцены на проекционную плоскость камеры в **экранных координатах (p, q)**;



3D-сцена и графический конвейер

Стадия рендеринга

(rendering -преобразование, представление)

1. **HSR (Hidden Surface Removal)** – Удаление скрытых, для текущей точки наблюдения, поверхностей. Алгоритмы: z-сортировка; z-буферизация; построчное сканирование

2. **Texture mapping.** Текстурирование – первый этап растровой графики. Тексели-элементы текстуры формата 2^m x 2^n. Соответствие пикселов и текселов устанавливается по результатам проецирования.
Приемы: **MIP-mapping** (текстуры с различным разрешением); **perspective corrected texture mapping** (коррекция перспективы); **Filtering** (LF, BLF, TLF, Anisotropic); **bump mapping** (наложение мелкомасштабного рельефа); мультитекстурирование (конвейерное) – 2 и более блоков текстурирования; **пиксельные шейдеры**

3. **α-blending and fogging** (моделирование полупрозрачности, коррекция цвета: α- смешивание и затуманивание)

4. **Anti-aliasing** (Коррекция зазубренности границ: edge AA (краевой) и full screen AA (полный ~ FSAA)
Приемы: super sampling (супер- и мультисэмплинг); tile based архитектура; техника аккумулятора.

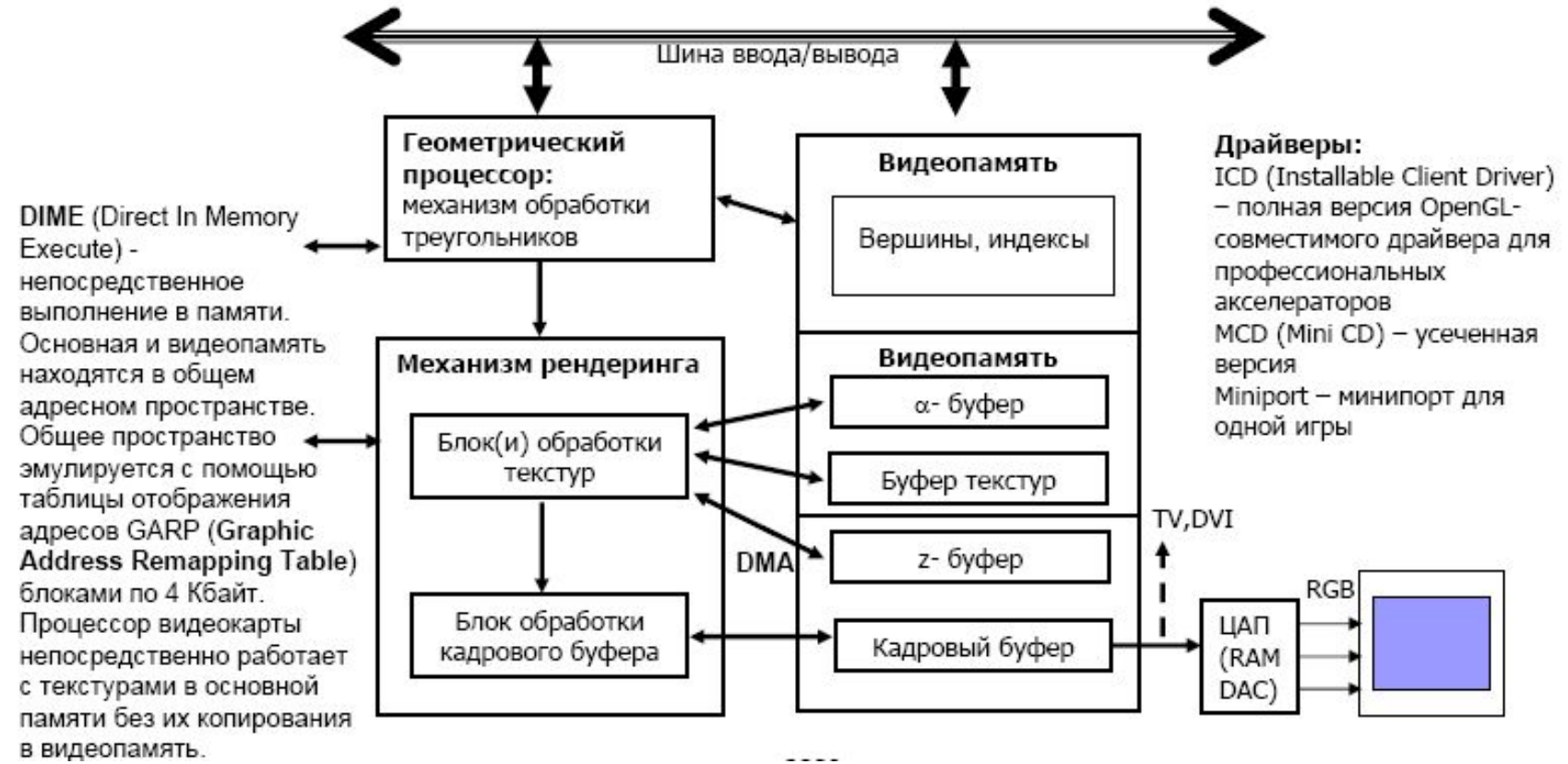
5. **Dithering**. Интерполяция недостающих цветов (для индексированногоцвета)

6. **Frame buffer**. Формирование кадрового буфера для формирования выходного аналогового сигнала. Приемы: double buffering двойная буферизация ~ формирование 2-го начинается до того как закончится передача в ЦАП (RAM DAC) первого

7. **Post-processing.** Пост-обработка: двумерные эффекты над целым кадром.

Основы аппаратной реализации графического конвейера: геометрический и пиксельный шейдеры

Обобщенная структура графического процессора (GPU)



Шейдер (англ. Shader) — это программа, используемая в трёхмерной графике, предназначенная для одной из ступеней графического конвейера, для определения финишных параметров объекта или изображения. Шейдер отвечает за функцию описания поглощения и рассеяния света, наложения текстуры, а также формирует отражение и преломление, затенение, смещение поверхности и определяет эффекты пост-обработки.
Программируемые шейдеры гибки и эффективны. Сложные с виду поверхности могут быть визуализированы при помощи простых геометрических форм. Например, шейдеры могут быть использованы для рисования поверхности из трёхмерной керамической плитки на абсолютно плоской поверхности.

Типы шейдеров: вершинные, геометрические и фрагментные (пиксельные).

вершинный шейдер (Vertex Shader) - управляет данными, отвечающими за вершины многогранников. Такими данными, являются координаты вершины в пространстве, текстурные координаты, тангенс-вектор, вектор бинормали, вектор нормали. При помощи вершинного шейдера происходит видовое и перспективное преобразования вершин, генерации текстурных координат, расчета освещения и т. д.

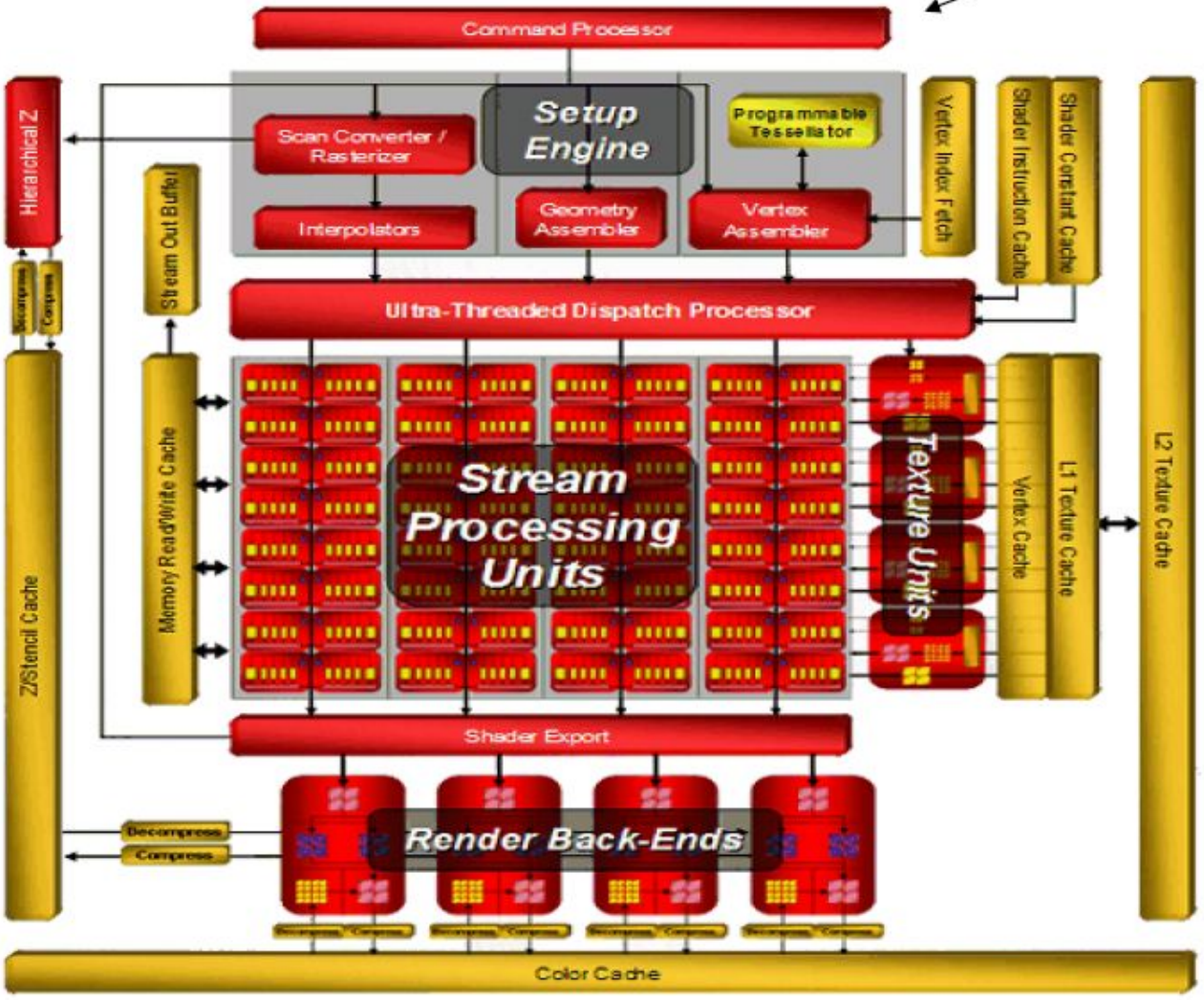
геометрический шейдер (Geometry Shader) - в отличие от вершинного шейдера, способен обработать не только одну вершину, но и целый примитив. Это может быть отрезок (две вершины) и треугольник(три вершины), а при наличии информации о смежных вершинах (adjacency) может быть обработано до шести вершин для треугольного примитива. Кроме того геометрический шейдер способен самостоятельно формировать примитивы, не используя главный процессор. Впервые геометрический шейдер начал использоваться на видеокартах Nvidia 8-й серии.

фрагментный - пиксельный шейдер (Pixel Shader) - работают с фрагментами изображения. Фрагментом изображения в данном случае является пиксел, а сам шейдер определяет набор атрибутов, таких как цвет, глубина, текстурные координаты. Фрагментный шейдер используется на последнем этапе графического конвейера для формирования фрагмента изображения.

Архитектура графических процессоров с шейдерами 4 версии

30% эффективности

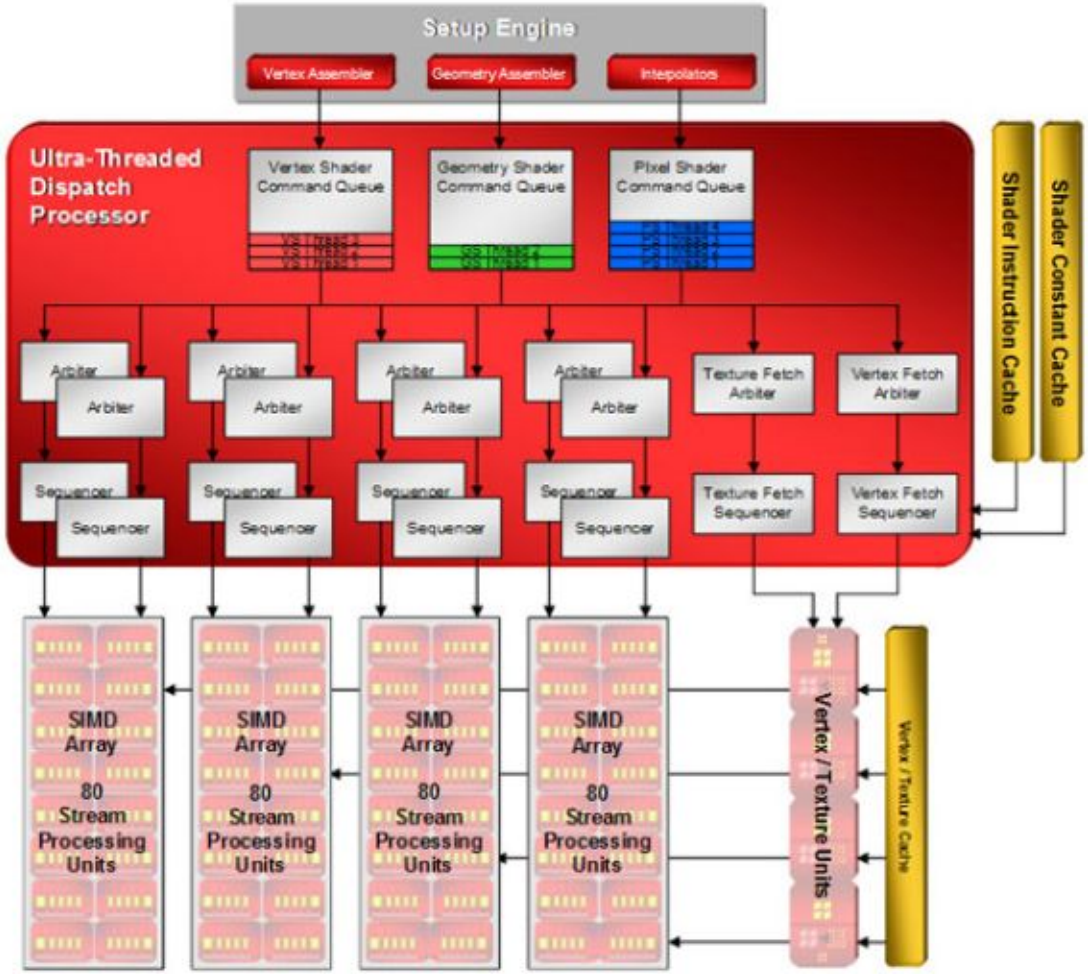
Архитектура
Radeon
HD 2900



Setup Engine:
Состоит из вершинного и геометрического ассемблеров и интерполятора, готовит данные для обработки потоковыми процессорами. Он может исполнять три вида функций:

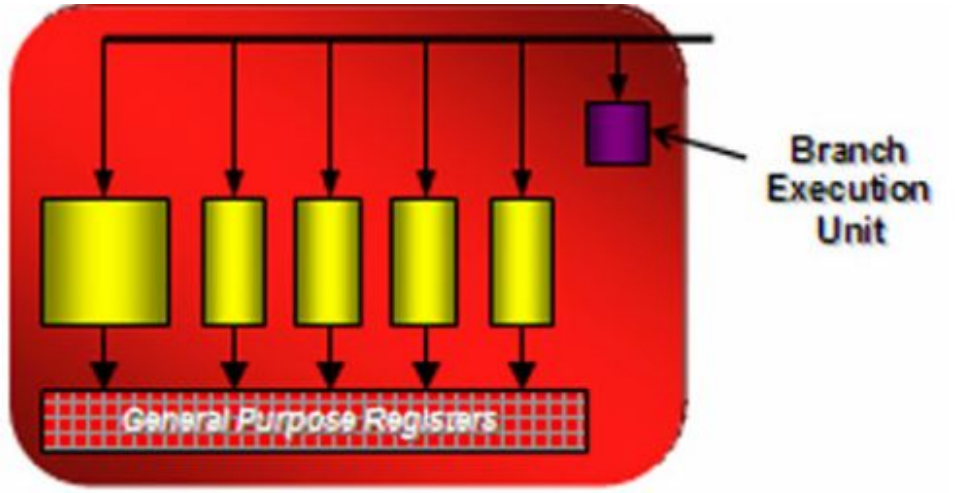
•сборку вершин (vertex assembly) и тесселяцию,
•геометрическую обработку (для нововведения DirectX 10 – Geometry Shaders)
•выборку и интерполяцию для пиксельных шейдеров.

Затем данные передаются диспетчеру.



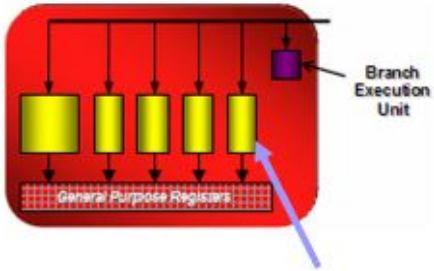
Ultra-Threaded Dispatch Processor. Он поддерживает раздельные очереди команд для каждого типа шейдеров. Инструкции и данные поступают в диспетчер и распределяются по очередям. Затем данные из очередей передаются на арбитраж. На каждый SIMD-массив приходится по два арбитра, что даёт возможность обрабатывать по 2 потока данных на массив, то есть до 8 потоков одновременно. Для текстур и вершин – свои арбитры.

Для вершинных и пиксельных шейдеров предусмотрены собственные кэши, геометрические шейдеры используют кэш вершинных. При этом **объём вершинного кэша увеличен в 8 раз** по сравнению с архитектурой Radeon X1950. Наличие кэша инструкций позволяет исполнять шейдеры неограниченной длины с неограниченным числом констант.



Потоковый процессор

Диспетчер следит за тем, чтобы исполняющие блоки не простаивали, и при возникновении ситуации, когда обрабатываемые в конвейере данные нуждаются в выборке из памяти или ожидают результатов обработки в другом конвейере, возвращает эти данные в текущем состоянии в очередь и запускает следующие из очереди в обработку. После выполнения условий, необходимых для продолжения обработки, она возобновляется с места остановки.



Ядро содержит 320 так называемых «единиц потокового вычисления» (stream processing units). Фактически процессоров в ядре 64, а не 320, они сгруппированы по 16 в 4 SIMD-массива (SIMD – Single Instruction on Multiple Data). Такое решение позволяет с большой скоростью проводить однотипные вычисления над большим количеством примитивов, что, собственно, мы и наблюдаем в современной графике (постобработка, шейдеры освещения, тени и т.п.). Каждый процессор способен выполнять 5 инструкций, при этом **4 из них – MUL/ADD (Multiply-Add),** и ещё **1** сложную трансцендентную инструкцию (**SIN, COS, LOG и т.п**.). Все инструкции с плавающей точкой выполняются с 32-битной точностью. В составе каждого процессора имеется блок исполнения ветвлений, управляющий выборкой данных и отсылкой их на исполнение, что уменьшает простой конвейеров и упрощает работу диспетчера. Каждый процессор оборудован регистром общего назначения, хранящим исходные данные, временные значения обработки и выходные значения после обработки. **Процессоры спроектированы под VLIW-архитектуру (Very Large Instruction Word),** каждое слово инструкции может содержать до 6 независимых операций, 5 из них математических и 1 управляющая (Flow Control).

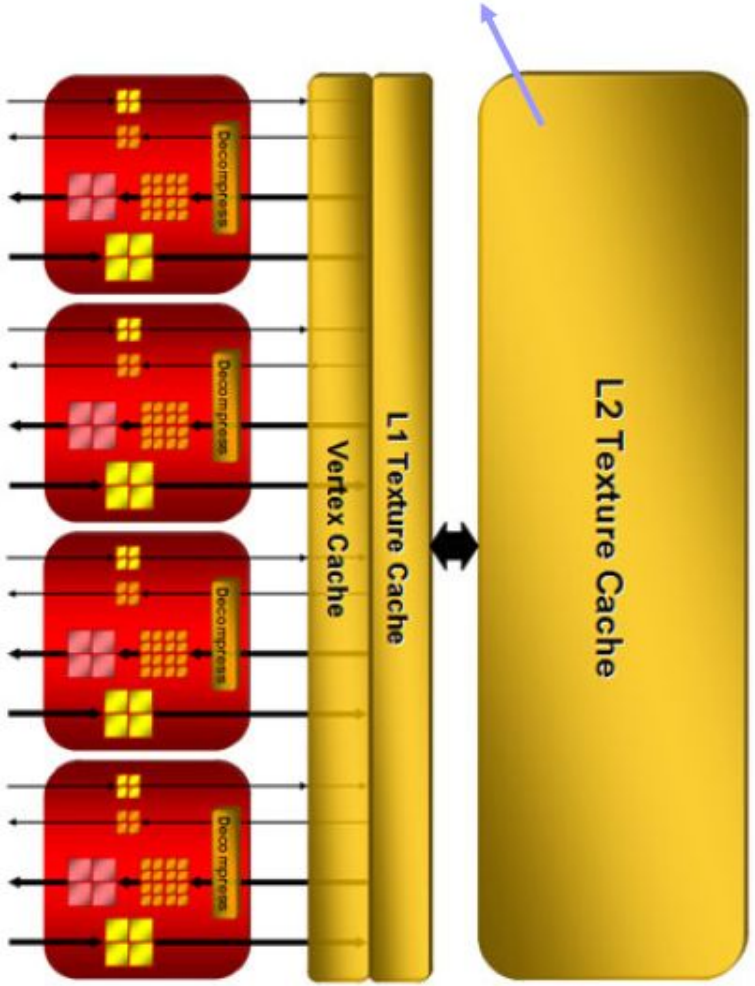
Вычислительная мощь SIMD-массивов ядра составляет до 475 ГигаFLOPs по оценке AMD, причём в случае использования массива Crossfire заявляется до 950 миллиардов операций с плавающей точкой в секунду.

Текстурные блоки (TMU).

Radeon HD 2900 оборудован четырьмя такими блоками. Каждый из них содержит 8 процессоров, осуществляющих адресацию текстур, 20 текстурных семплеров, осуществляющих по 1 выборке текстуры за такт, 4 блока фильтрации FP(FloatingPoint)-текстур. Для текстурных блоков сохранена способность архитектуры X1000, называемая Fetch4: возможность выборки 4 нефильтрованных значений вместо 1 фильтрованного.
В результате Radeon HD 2900 способен за 1 такт адресовать 32 текселя, осуществить выборку 80 значений текстурных координат и билинейную фильтрацию одного 64-битного значения цветовых координат (64-битный HDR). Билинейная фильтрация 128-битных текстур осуществляется за 2 такта. Это в 7 раз быстрее, чем у Radeon X1000.

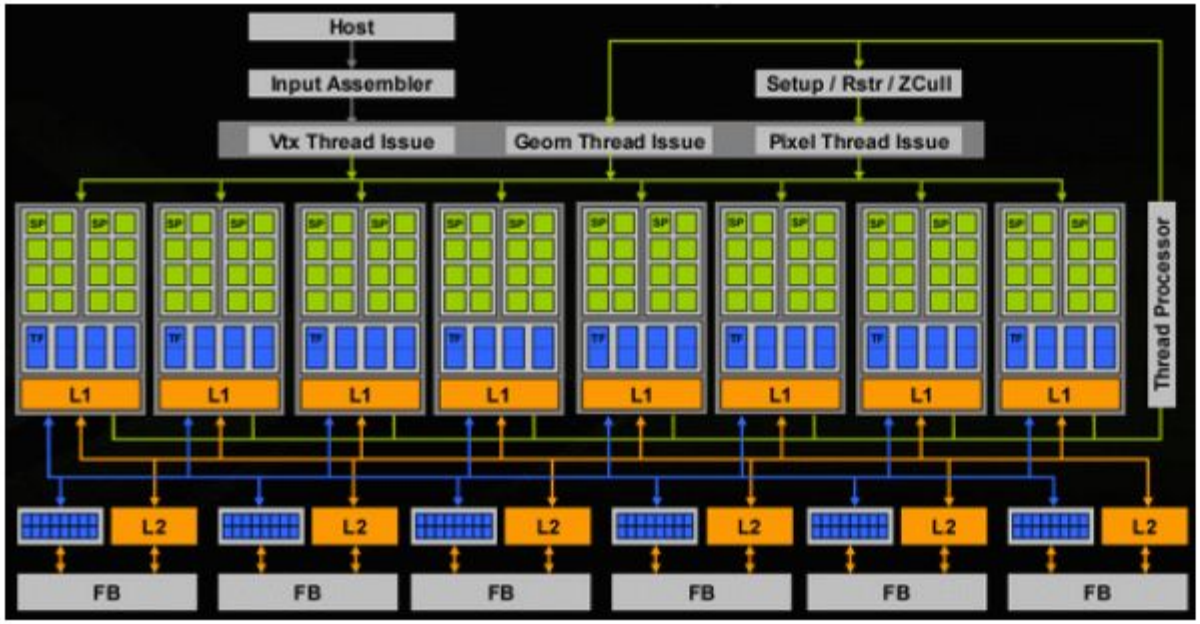
Текстурные блоки разделяют 256 кбайт кэша второго уровня, при этом без ограничений могут получать доступ к кэшу 1 уровня и вершинному кэшу.

ATI вводит новый формат данных для 32-битных HDR – RGBE 9:9:9:5.
Поддерживается хранение и использование текстур сверхвысокого разрешения – 8192x8192.



Архитектура NVIDIA GeForce 8800 GTX

2.1 SIMD ядра

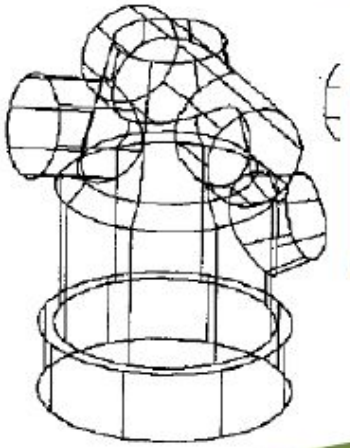


NVIDIA GeForce 8800 GTX **имеет: 128 скалярных потоковых процессоров (scalar processors), которые сгруппированы в 8 блоков (мультипроцессоров) по 16 процессоров, каждый блок должен быть оснащен 4 текстурными модулями и общей кэш-памятью L1.**

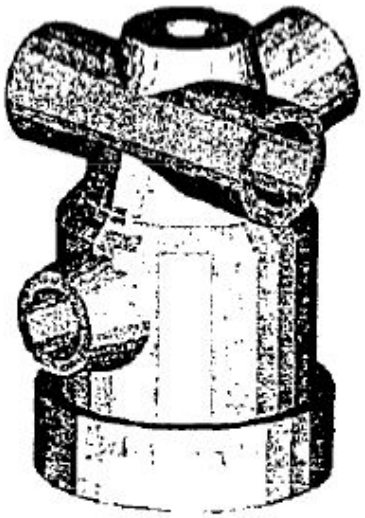
Vtx (Geom|Pixel) Tread Issue – исток для вершинных (геометрических / пиксельных шейдеров) Каждый блок делится на два шейдерных процессора (каждый из которых состоит из 8 потоковых процессоров), имеющих SIMD архитектуру. И все восемь блоков имеют доступ к любому из шести L2 кэшей и к любому из шести массивов регистров общего назначения (РОН). Таким образом, данные обработанные на одном шейдерном процессоре могут быть использованы другим шэйдерным процессором.

Графический процесс: типовая последовательность применения алгоритмов

Геометрическая модель
Модель освещения
Модель анимации



Модель → Алгоритм синтеза → Изображение



Получение проекции
Заливка

При выборе алгоритма надо выбрать между скоростью и качеством

Скорость

Растеризация

Качество

Трассировка лучей/фотонов
Излучательность

В общем случае растеризация – преобразование векторной информации в растровый формат

Свойства: скорость и поддержка на уровне графических процессоров

- Широко распространен
- Аппаратная поддержка
- OpenGL, DirectX реализуют именно этот подход
- Ориентация на скорость визуализации

- Определяет только проекцию, поэтому может быть совмещен с различными подходами по вычислению цвета

OpenGL

OpenGL – это кросс-платформенная программная библиотека функций для создания интерактивных 2D и 3D приложений.

Является отраслевым стандартом с 1992.

Основой стандарта стала библиотека IRIS GL, разработанная фирмой Silicon Graphics Inc.

Основная функция: интерактивная визуализация трехмерных функций.

OpenGL проста для изучения

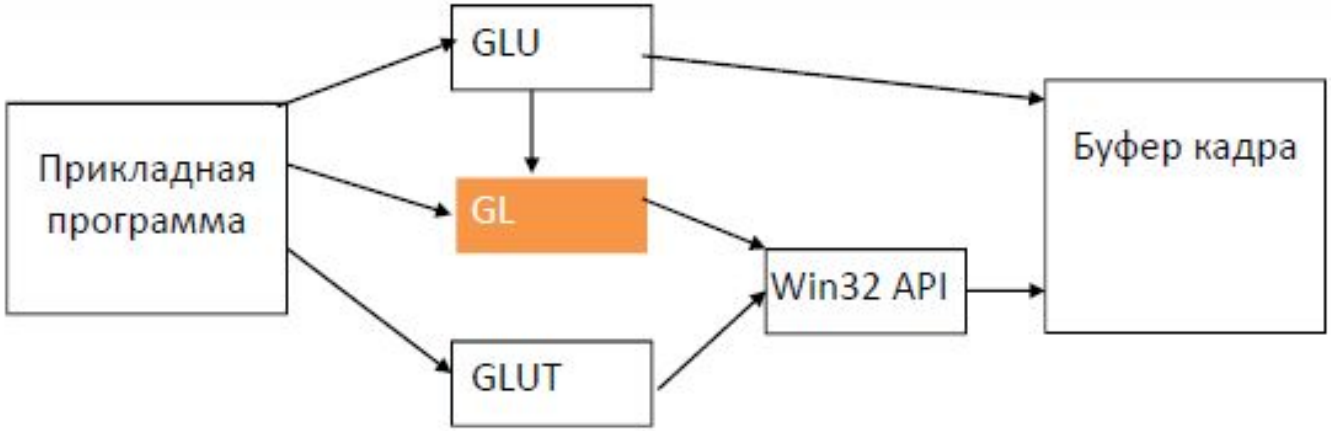
Аналогичные библиотеки: DirectX, Java 3D

OpenGL

– Стабильность (с 1992 г
– Переносимость: Независимость от оконной и операционной системы
– Легкость применения: простой интерфейс, открытость

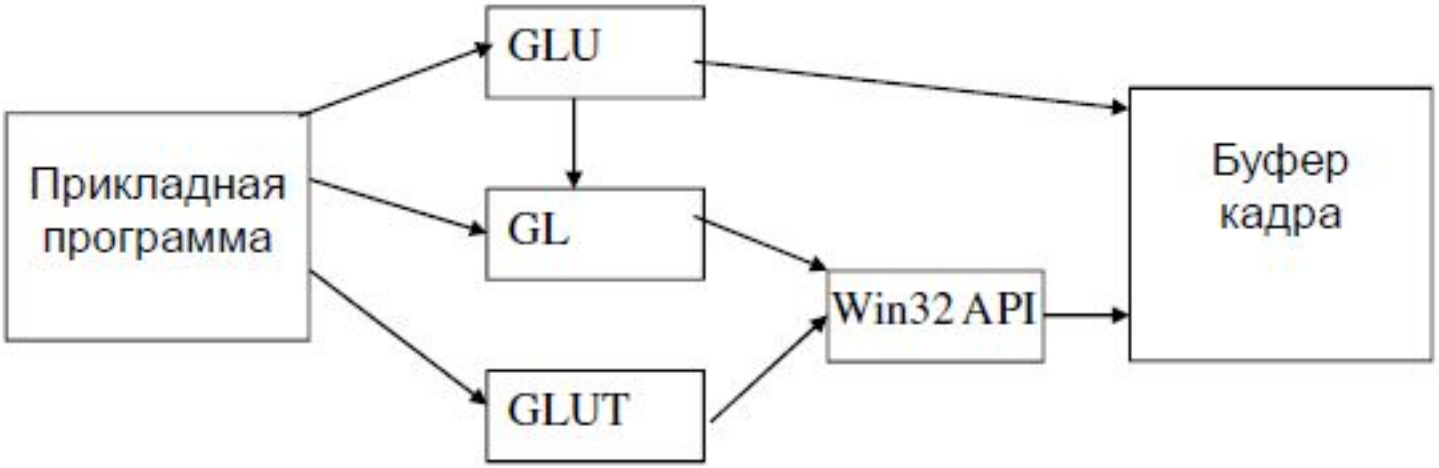
OpenGL – прослойка между программой и драйвером видеокарты

- Взаимодействует с пользовательскими программами, драйвером видеокарты и с WIN API



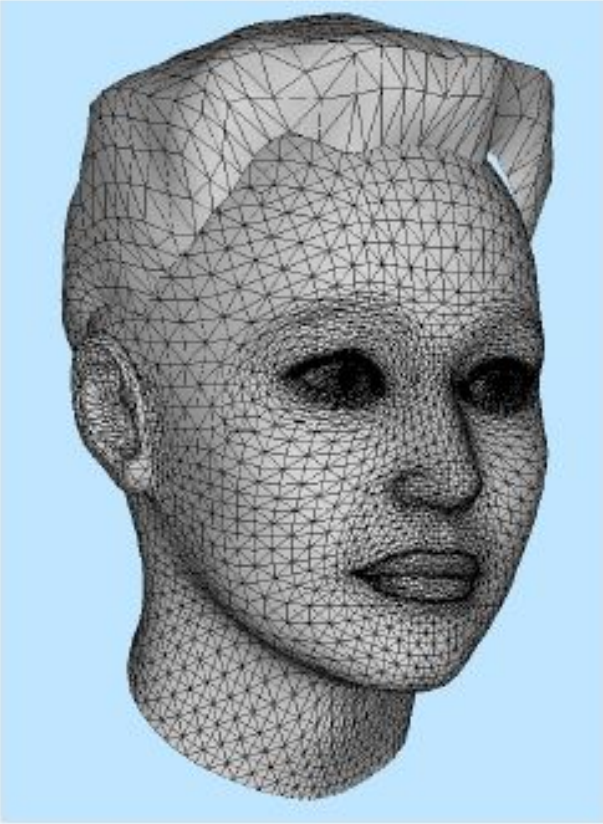
OpenGL состоит из набора библиотек

- AGL, GLX, WGL
  - Связь между OpenGL и оконной системой
- GLU (OpenGL Utility Library)
  - Часть OpenGL
  - NURBS, tessellators, quadric shapes, etc
- GLUT (OpenGL Utility Toolkit)
  - Переносимый оконный API
  - Неофициальная часть OpenGL



OpenGL работает с разными типами моделей, в основном с полигональными

OpenGL работает с моделями, заданными в граничном полигональном представлении

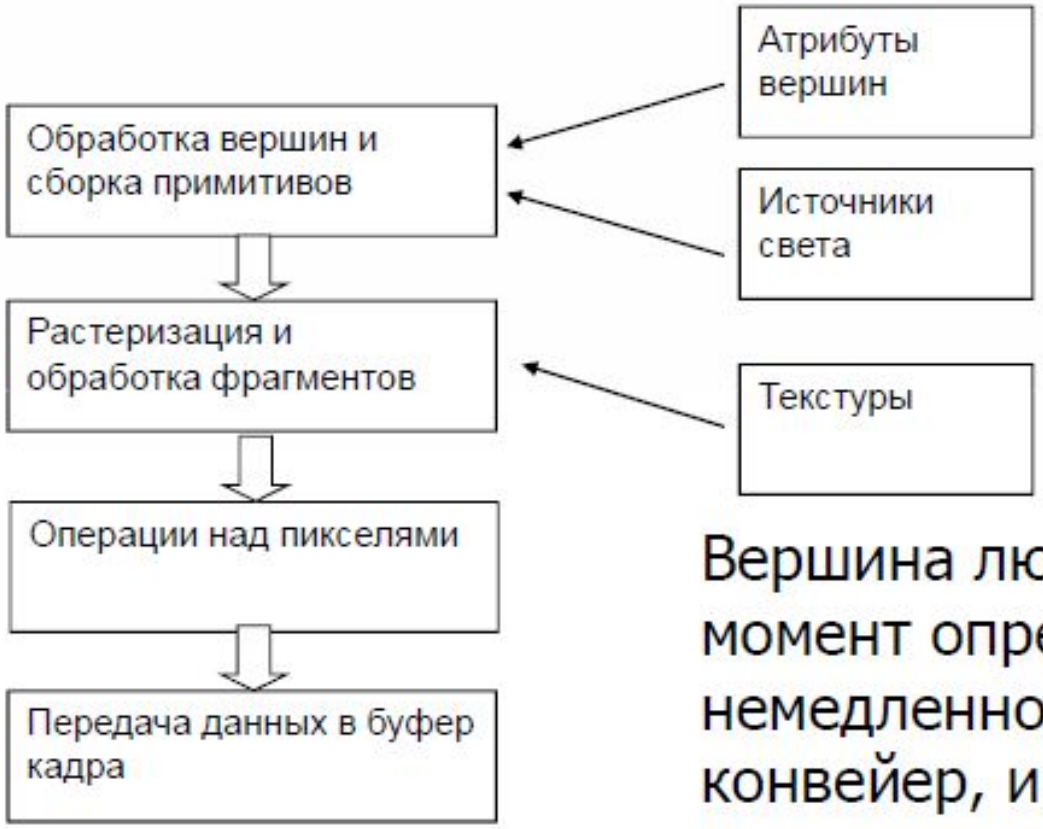


Поверхность приближается набором полигональных граней (face, polygon)

Границы граней описываются ребрами (edge)

Часть отрезка, формирующего ребро, заканчивается вершинами (vertex)

Программная архитектура OpenGL может быть представлена в виде конвейера



Вершина любого объекта в момент определения немедленно передается в конвейер, и проходит все его ступени

3D координаты -> экранные

Два типа функций (команд): передача данных и изменения состояния

Команды передача данных

- Объекты на экране рисуются путем последовательной передачи в конвейер вершин примитивов, которые составляют объект

Команды изменения состояния

- Настройка обработки данных на каждом этапе конвейера

OpenGL может по-разному объединять вершины в полигоны

GL_POINTS

GL_LINES

GL_LINE_STRIP

GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_QUAD_STRIPE

GL_POLIGON

GL_QUADS

С каждой вершиной ассоциировано несколько атрибутов

Каждая вершина кроме положения в пространстве может иметь несколько других атрибутов

- Материал
- Цвет
- Нормаль
- Текстурные координаты

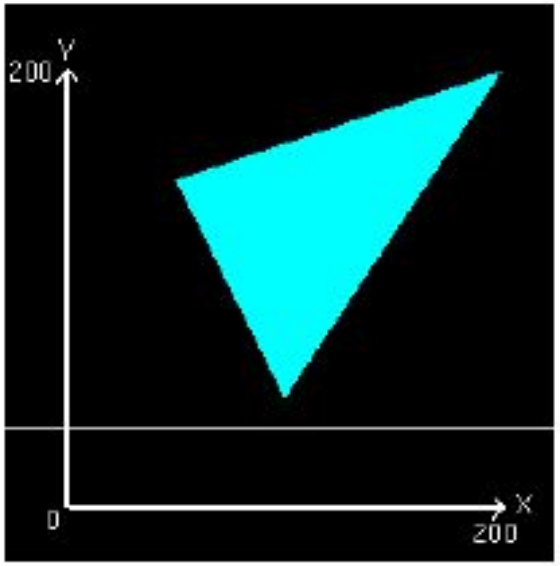
Внимание: всегда используется ТЕКУЩИЙ набор атрибутов

- OpenGL как конечный автомат

Пример кода для вывода треугольника

Цветной треугольник

- glBegin(GL_TRIANGLES);
  - glColor3f(0.0f,1.0f,0.0f);
  - glVertex2f(150.0f, 50 .0f);
  - glVertex2f(50.0f, 150 .0f);
  - glVertex2f(200 .0f, 200 .0f);
- glEnd();



Таким образом можно задать любой объект!

Теперь задача в том, чтобы правильно показать этот объект на экране

В OpenGL каждая вершина задается в локальной системе координат

- Каждая вершина объекта задается в локальных координатах модели

- Необходимо определить набор геометрических преобразований, таких, что каждая вершина преобразуется в точку на плоскости экрана

- Три последовательных преобразования:
  - модельное преобразование
  - видовое преобразование
  - проективное преобразование

В конвейере OpenGL применяются исключительно линейные и проективные преобразования

В графическом конвейере OpenGL используются линейные и проективные геометрические преобразования

Преобразования описываются матрицами 4x4

Операции производятся над векторами в однородных координатах

Виртуальная камера – нужно задать для получения изображения

- Определяет положение наблюдателя в пространстве
- Параметры
  - Положение
  - Направление взгляда
  - Направление «вверх»
  - Параметры проекции
- Положение, направление взгляда и направление «вверх» задаются матрицей видового преобразования

- Что если мы хотим переместить наблюдателя?

  Варианты:
  - Изменить матрицу проекции чтобы включить в нее информации о камере
  - **Применить дополнительное преобразование, «подгоняющее» объекты под стандартную камеру**

- Стандартная камера в OpenGL:
  - Наблюдатель в (0, 0, 0)
  - Смотрит по направлению (0, 0, -1)
  - Верх (0, 1, 0)

Модельно-видовое преобразование = модельное + видовое

- OpenGL не имеет раздельных матриц для видового и модельного преобразования

- Поэтому нужно задавать сразу произведение:

$$M = M_{view} \cdot M_{mdl}$$

Матрицы преобразований

- Выбираем матрицу преобразований для изменения:

```
void glMatrixMode(Glenum mode);
    mode={GL_MODELVIEW|GL_PROJECTION}
```

- Две основные операции над матрицами

```
void glLoadIdentity();
```

$$M = E$$

```
void glMultMatrixd(GLdouble c[16]);
```

$$M = M \cdot \begin{bmatrix} c[0] & c[4] & c[8] & c[12] \\ c[1] & c[5] & c[9] & c[13] \\ c[2] & c[6] & c[10] & c[14] \\ c[3] & c[7] & c[11] & c[15] \end{bmatrix}$$

Матрицы преобразований (2)

```
void glTranslated(GLdouble x,
                  GLdouble y,
                  GLdouble z);

void glScaled(GLdouble x,
              GLdouble y,
              GLdouble z);

void glRotated(GLdouble angle,
               GLdouble ax,
               GLdouble ay,
               GLdouble az);

void gluPerspective(GLdouble fov,
                    GLdouble aspect,
                    GLdouble znear,
                    GLdouble zfar);
```

Видовое преобразование

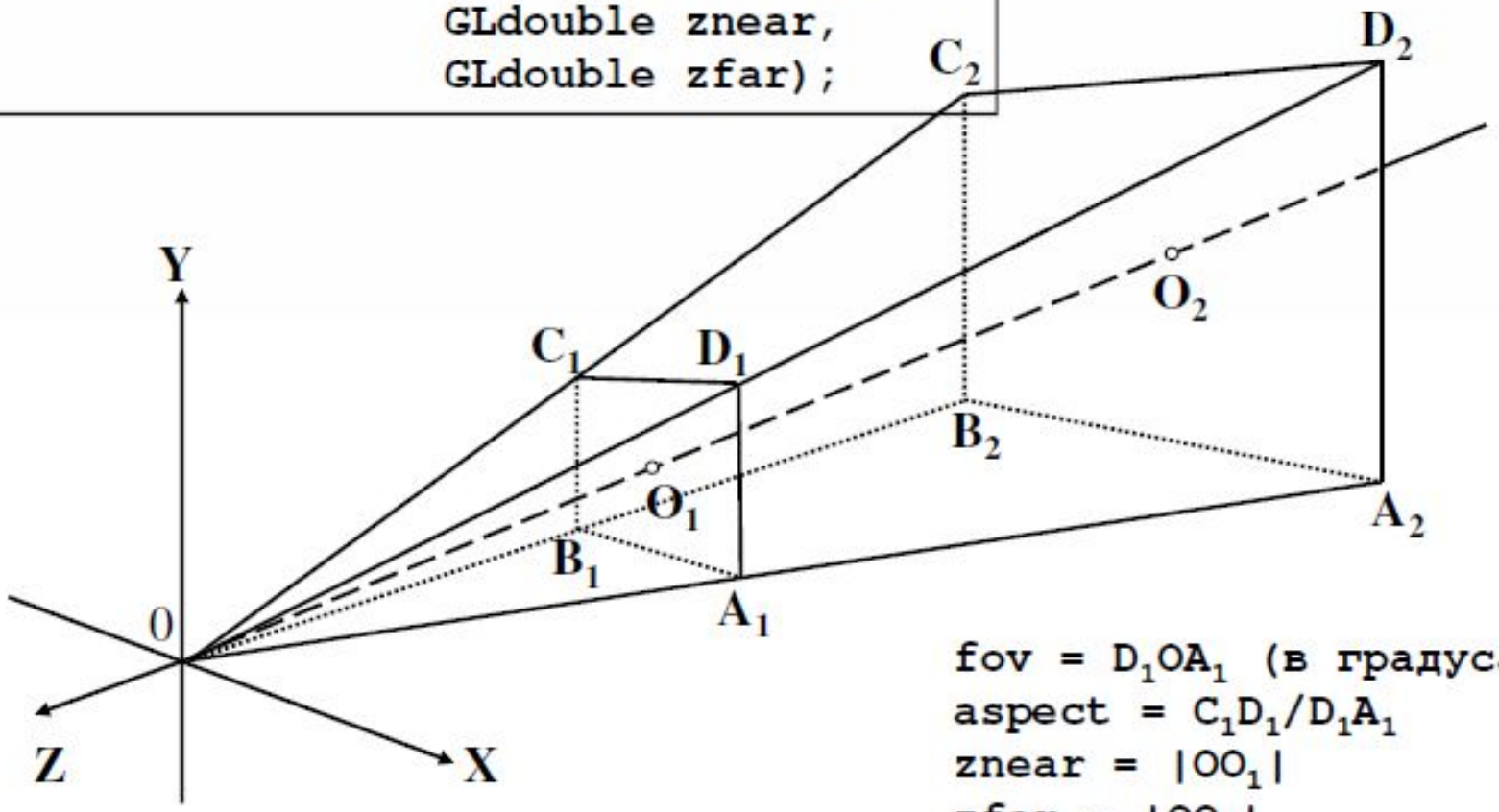
- Настройка виртуальной камеры

$$\text{gluLookAt}(\ eye_x, eye_y, eye_z,\ aim_x, aim_y, aim_z,\ up_x, up_y, up_z)$$

- eye – координаты наблюдателя
- aim – координаты “цели”
- up – направление вверх

Как работает gluPerspective

```
void gluPerspective(GLdouble fov,
                    GLdouble aspect,
                    GLdouble znear,
                    GLdouble zfar);
```



fov = D_1OA_1 (в градусах)
aspect = C_1D_1/D_1A_1
znear = $|OO_1|$
zfar = $|OO_2|$