Компютърна графика

Графични процесори
Програмиране за GPU

доц. Милена Лазарова, кат. КС, ФКСУ

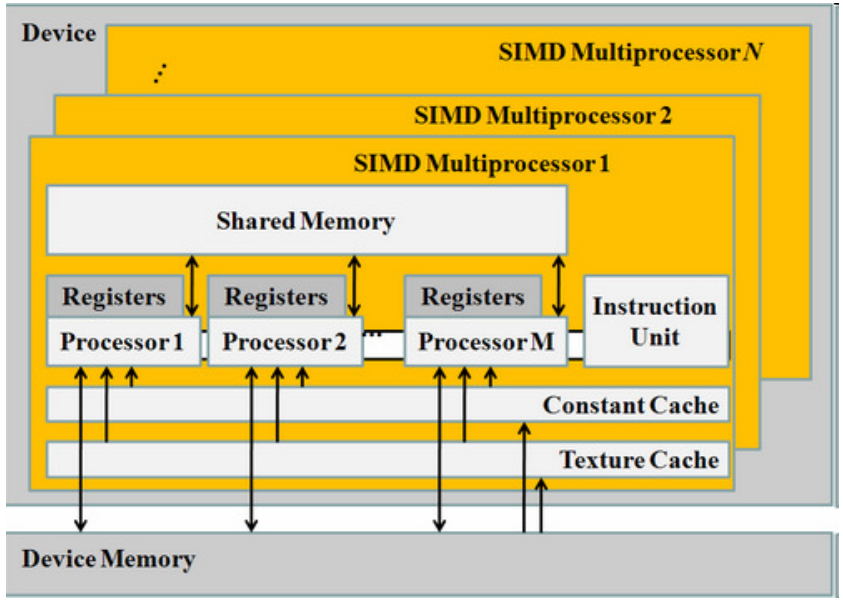
Компютърна графика – обобщение

- Трансформации за моделиране на обекти
- Визуализираща трансформация
- Проекция (перспективна, паралелна)
- Определяне на видими повърхнини
- Осветеност
- Рендериране

Архитектура на GPU

- ***GPU = Graphics Processing Unit***
- Основен хардуерен елемент
  - ***потоков мултипроцесор*** “streaming multiprocessor” (SM)

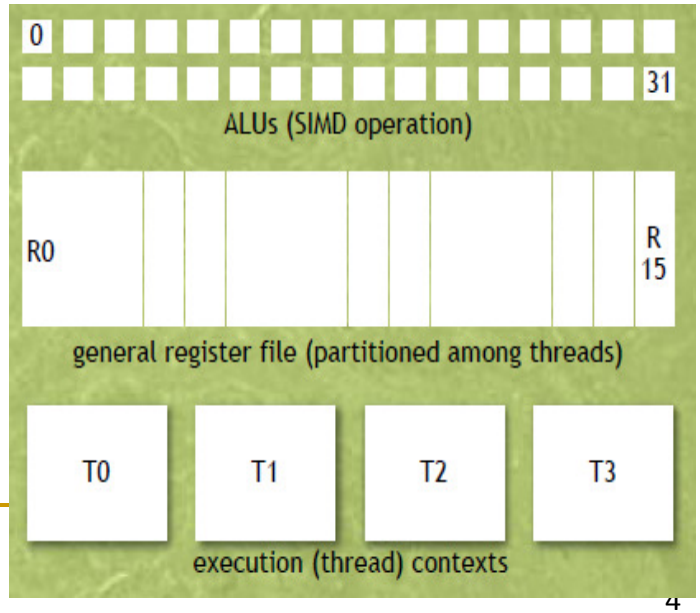
    пример
    - 8 ядра
    - 2048 регистъра за всяко ядро
    - 16KB споделена памет
    - 8KB кеш за константи
    - 8KB кеш за текстури



- Различните графични процесори имат различен брой потокови мултипроцесори

Архитектура на GPU

- Основна хардуерна характеристика
  - 8^те ядра на потоковия мултипроцесор са ***SIMT ядра***
    - ***Single Instruction Multiple Threads***
  - всичките 8 ядра изпълняват една и съща инструкция едновременно над различни данни
    - подобно на векторната архитектура на суперкомпютрите CRAY
  - минимум 4 нишки на ядро
  - минимум 32 нишки изпълняват една и съща инструкция в (почти) едно и също време
    - естествено изпълнение на графична обработка и голям брой изчислителни задачи



Архитектура на GPU

- CPU Intel quad-core Xeon
  - 4 ядра
  - 10-40 Gflops
  - 20-30GB/s memory bandwidth
- GPU NVIDIA Tesla
  - 448 ядра
  - 1 Tflops single precision, 515 Gflops double precision
  - 144 GB/sec memory bandwidth
- един графичен процесор NVIDIA Tesla е 5-10 пъти по-бърз от два четири ядрени процесора Intel Xeon при приблизително същата цена и консумация на енергия

Архитектура на GPU

- Следващо поколение процесори Intel CPU
  - Westmere-EP
    - 6 ядра
  - Nehalem-EX
    - 8 ядра
  - Sandy Bridge
    - 4-10 ядра

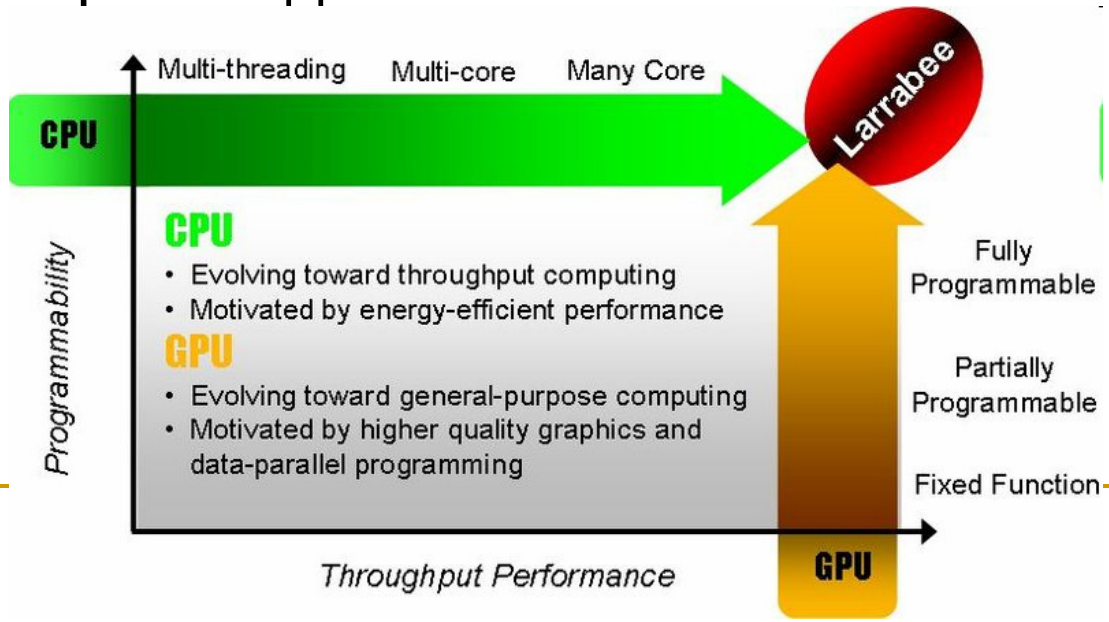
- Следващо поколение графични процесори NVIDIA GPU
  - Fermi
    - 512 ядра
    - 1.5 Tflops single precision, 800 Gflops double precision
    - L1/L2 кеш за GPU
    - 200GB/s memory bandwidth

Архитектура на GPU

- Други графични процесори
  - AMD
    - Radeon HD 6990
    - 307 GB/sec
    - поддържа стандарта OpenCL за GPGPU (General Purpose computation for GPU)
  - IBM Cell
    - трудно се програмира
    - не се развива за научни изчисления
- Intel Larrabee
  - 16-32 ядра с векторни устройства

CPU vs. GPU

- Тенденции
  - увеличаване броя на ядрата на CPU и GPU
  - улесняване на програмирането на GPU
  - използване на хетерогенни CPU-GPU системи за високопроизводителни изчисления



Програмиране за GPU

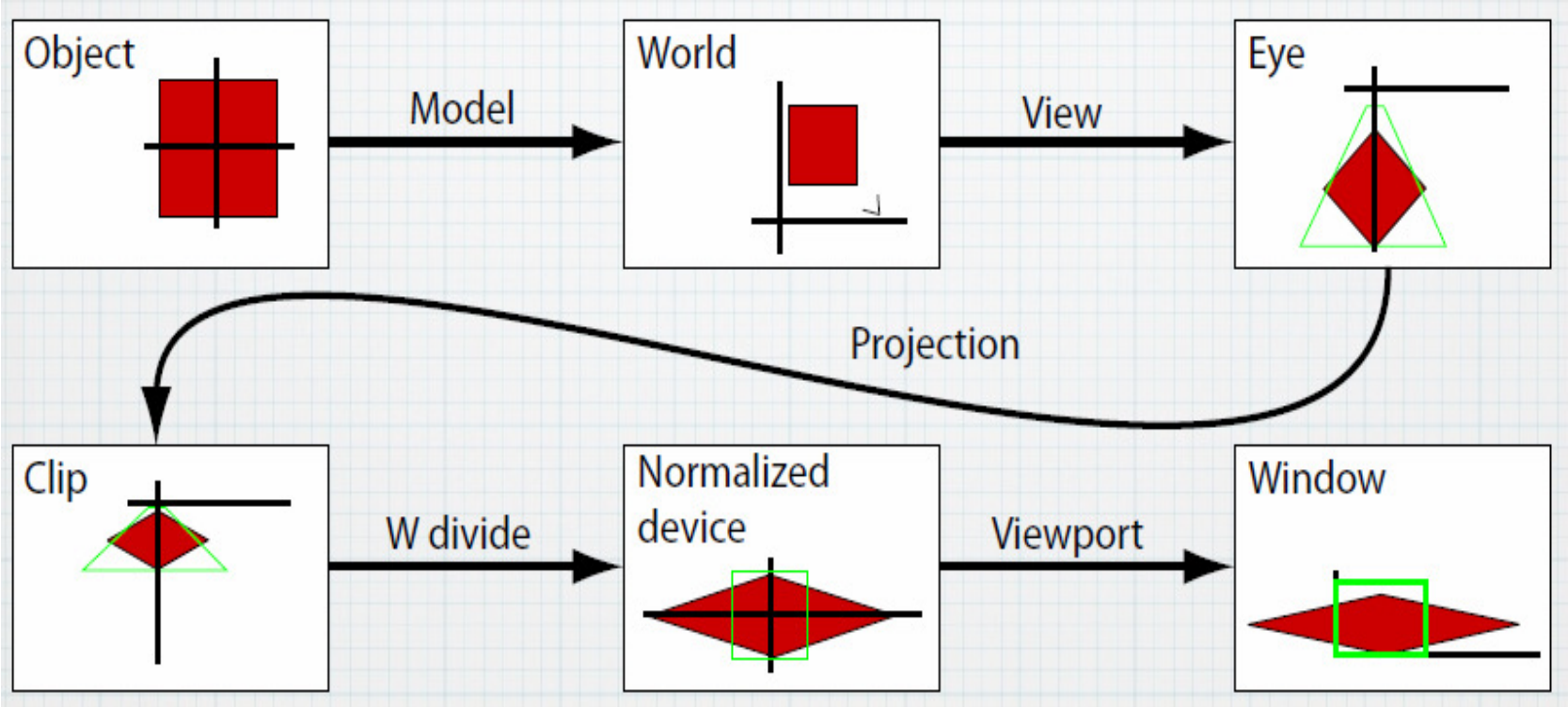
- Подходящо за всички задачи, свързани с графични операции, обработка на изображения, компютърно зрение
  - естествен паралелизъм
  - добра съвместимост
  - високо оптимизирани за графични изчисления

Основен графичен конвейер

- Цел на графичните системи и софтуерни продукти е генериране на изображения, представящи визуализация на виртуални сцени
  - сцената се определя от геометрията, ориентация, както и свойствата на материалите на повърхностите на обектите и позицията и характеристиките на източници на светлина
  - визуализирането на сцената се определя от местоположението на виртуална камера
- Интерфейсите са графична визуализация в реално време като Direct3D и OpenGL представят изчисленията за визуализиране на сцените като

конвейер на графичната обработка

Основен графичен конвейер

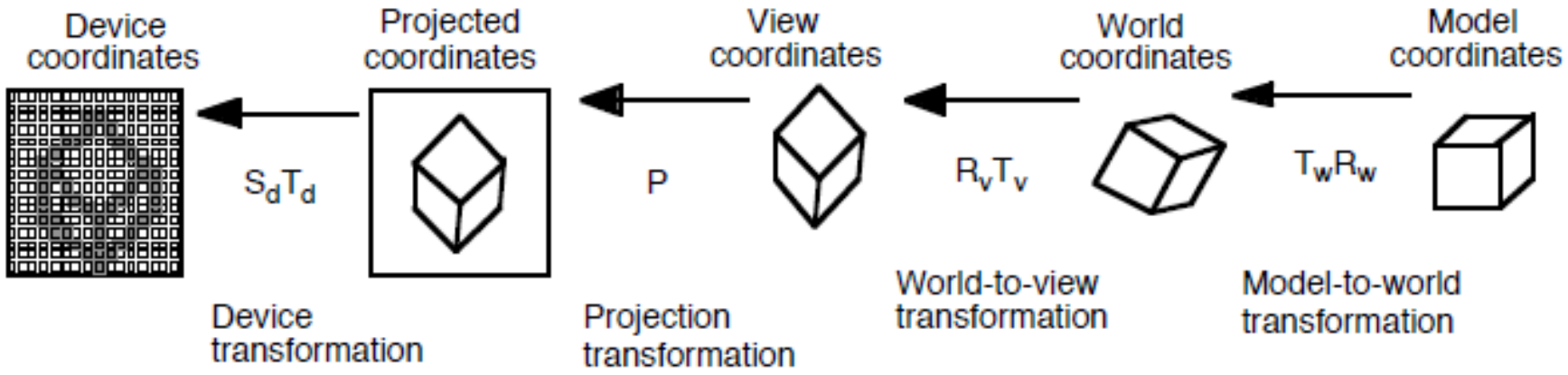


Основен графичен конвейер

- Предвижда извършване на операции над четири основни фундаментални единици
  - ***възли***
  - ***примитиви***
  - ***фрагменти***
  - ***пиксели***

Трансформации

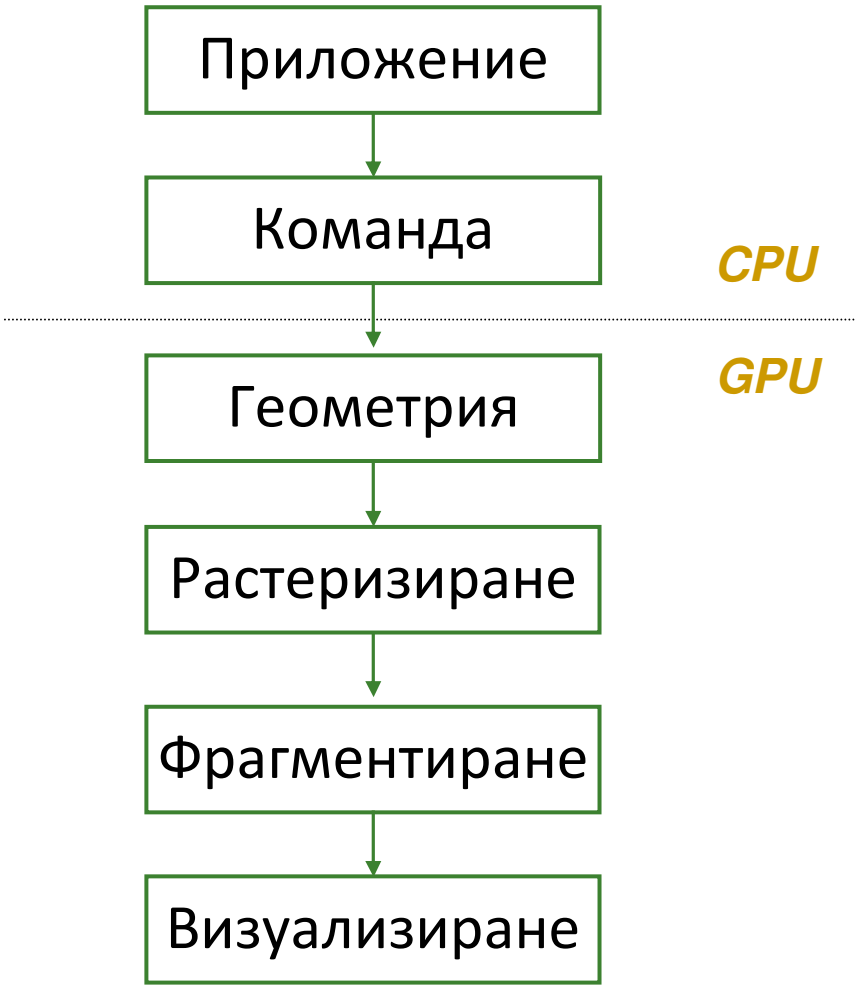
- Повечето операции в графичния конвейер се изпълняват чрез трансформации
  - матрични умножения на матрици 4×4



визуализиране ***промяна параметри на камерата*** ***преместване на камерата*** ***преместване на обекта***

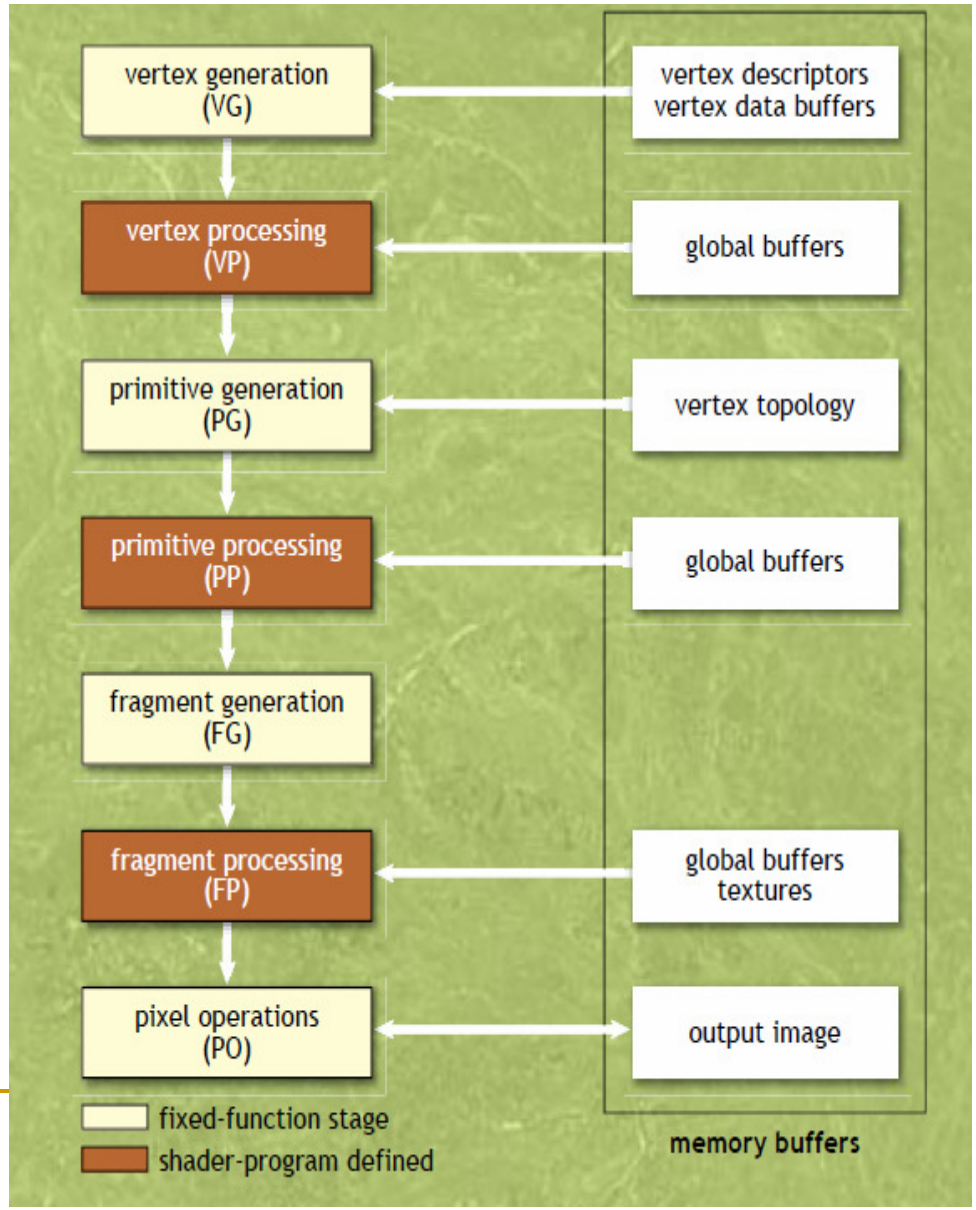
Основен графичен конвейер

- Традиционния конвейер
  - фиксирани функции
  - конфигурират се чрез интерфейс
  - ограничени характеристики



Програмируем графичен конвейер

- Добавя се възможност за програмиране в 3 от фазите на конвейера



Фаза възли

- ***Генериране на възли*** – VG (vertex generation)
  - графичните приложения за реално време представят повърхностите като съвкупност от прости геометрични примитиви (точки, линии, триъгълници)
  - всеки примитив се дефинира чрез множество от възли
  - за да се инициира визуализирането графичното приложение предоставя списък с описание на възли
  - фазата на генериране на възли на графичния конвейер прочита данните от паметта и обработва списъка като генерира поток от данни за възлите
  - данните за всеки възел са 3D координати на възела в сцената и допълнителни параметри за възела

Фаза възли

- ***Обработване на възли*** – VP (vertex processing)
  - обработва се всеки възел поотделно в зависимост от конкретното приложение
  - от данните за всеки входен възел се генерира точно един изходен възел
- ***Данни за възлите***
  - позиция, цвят, текстурни координати, нормален вектор
- ***Операции (в програма обработваща възли)***
  - изпълнение на последователност от математически операции за всеки възел
  - трансформиране на позициите на възел в екранни координати за растеризиране
  - генериране на текстурни координати за текстуриране
  - определяне на осветеност на възел за генериране на цвета му

Фаза примитиви

- ***Генериране на примитиви*** – PG (primitive generation)
  - използва топологични данни за възлите, предоставени от графичното приложение, за да групира възлите в подреден поток от примитиви
    - данните за всеки примитив за обединението на данните за няколко възела
  - топологията на възлите определя реда на примитивите в изходния поток

Фаза примитиви

- ***Обработване на примитиви*** – PP (primitive processing)
  - обработват се поотделно всички входни примитиви за да се генерират 0 или повече изходни примитиви
    - резултатът е нов подреден поток от данни за примитиви, който може да е по-дълъг или по-къс от входния
- ***Операции с примитиви***
  - групиране на възли в геометрични примитиви (триъгълници, линии, точки)
  - изрязване по визуалния обем и други изрязващи равнини
  - елиминиране на задните невидими стени (culling)
  - растеризиране на геометрични примитиви до фрагменти
    - определяне на множество координати на пиксели и фрагменти

Фаза фрагменти

- ***Фрагмент***
  - *потенциален* пиксел (който все още може да бъде премахнат)
- ***Данни за фрагмент***
  - цвят, дълбочина, местоположение, множества текстурни координати
- ***Генериране на фрагмент*** – FG (fragment generation)
  - генерират се данни за изходни фрагменти – позициите в изображението на повърхността, описана с фрагмента, разстояние до виртуалната камера, интерполиране на параметри на възлите

Фаза фрагменти

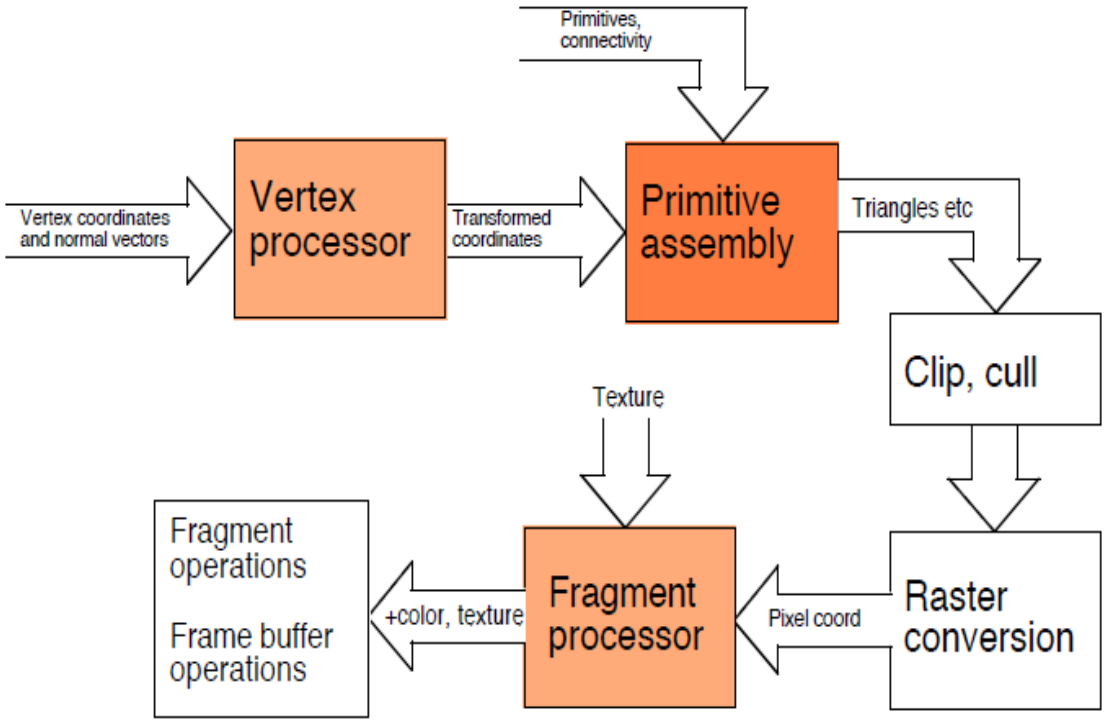
- ***Обработване на фрагмент*** – FP (fragment processing)
  - симулира се взаимодействието на светлината с повърхностите в сцената за определяне на цвета на всяка точка (sample point) от фрагмента
  - операциите на тази фаза са програмируеми
- ***Операции с фрагменти (в програма обработваща фрагменти)***
  - интерполация, текстуриране и определяне на цвета
  - математически операции

Фаза пиксели

- ***Операции с пиксели*** – PO (pixel operations)
  - използва се позицията на екрана на всеки фрагмент за да се изчислят данни за резултантния пиксел (координати и цвят)
  - за тази фаза а конвейера се гарантира обработване в реда специфициран от потока данни
    - например когато фрагменти от няколко примитива участват при определянето на стойността на единствен пиксел (както е при наслагващи се полу-прозрачни повърхности) обновяването на стойностите на пикселите е в реда определен от позициите на пикселите в потока данни, който се получава от предишната фаза

OpenGL pipeline

- възлите и фрагментите са вектори (с размерност до 4)
- фазите на възли и фрагменти са програмируеми
  - в по-новите графични процесори и фазата на примитивите също

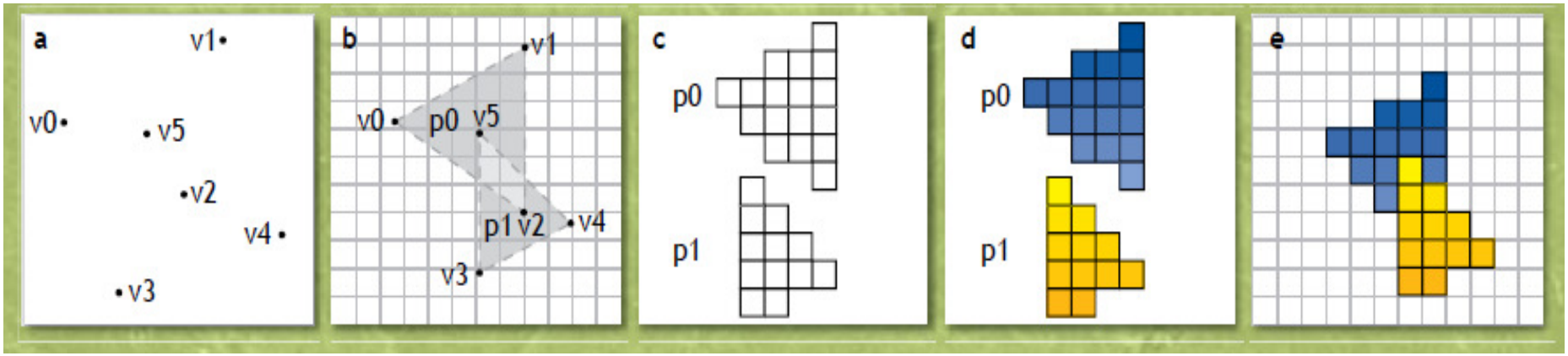


Основен графичен конвейер

(A) шест възела от изходния поток на фазата на възлите дефинират позициите и ориентацията на 2 триъгълника

(C) след фазата на генериране на фрагменти двата примитива водят до получаване на множества от фрагменти, съответстващи на *p0* и *p1*

(E) последната фаза на конвейера обновява пикселите в генерираното изображение с отчитане на видимостта на повърхностите



(B) след фазите на обработване на възли и генериране на примитиви възлите се трансформират в екранни координати и се групират в 2 примитива - триъгълниците *p0* и *p1*

(D) обработването на сегменти определя как се визуализират повърхностите във всяка точка

(в примера *p1* е по-близо до камерата от *p0*, в резултат на което *p1* се закрива от *p0*)

Програмиране на shader функции

- Поведението на фазите на конвейера, които са програмируеми от графичното приложение (VP, PP, FP) се дефинира чрез
  shader функции (или ***shaders***)
- при програмирането се задават в явен вид shader функции за върхове, примитиви и фрагменти на ***shading език*** от високо ниво
- сорсът се компилира в байткод и след това преобразува в специфичен двоичен формат за GPU от графичния драйвер по време на изпълнение
- Shading езиците поддържат сложни типове данни, както и богат набор от конструкции за управление на потока данни
- не съдържат примитиви, свързани с изрично паралелно изпълнение

Програмиране на shader функции

- По този начин дефиницията за shader функция е подобна на C функция, която последователно изчислява изходни единици на базата на единствена входна единица
- Извикването на всяка функция е абстракция за независимо управление и изчисление в пълна изолация от обработването на останалите единици в потока
- Shader функциите имат достъп до входните и изходните данни и достъп за четене, но не и модифициране на големи глобално споделени буфери с данни

Vertex shader

- Заменя фиксирана функционалност на фазата на обработване на възли
  - може да
    - трансформира възли, нормали и текстурни координати
    - генерира текстурни координати
    - изчислява осветеност на всеки възел
    - определя параметри за интерполация, които да се използват от fragment shader
  - не може да
    - променя перспектива, рамка за визуализиране
    - визуален обем
    - примитиви (!)
    - премахване на скрити стени (culling)

Fragment shader

- Нарича се още pixel shader
- Заменя фиксирана функционалност на фазата на обработване на фрагменти
  - може да
    - определя и задава цвят на фрагмент
    - определя цветове от текстура
    - изчислява мъгла и цветовете при други ефекти
    - използва интерполирани данни за възлите
  - не може да
    - променя координати на фрагмент
    - променя текстури
    - манипулира данните като променя параметрите за stencil, scissor, alpha, depth

Програмиране на shader функции

- Shader функциите се изпълнява върху GPU докато останалата програма се изпълнява от CPU
- Всеки shader е в отделен модул
  - компилира се отделно
  - свързва се с OpenGL програмата
- Типични примери за shader функции в КГ
  - манипулиране на възли
  - изчисляване на осветеност
  - multitexturing
  - bump mapping

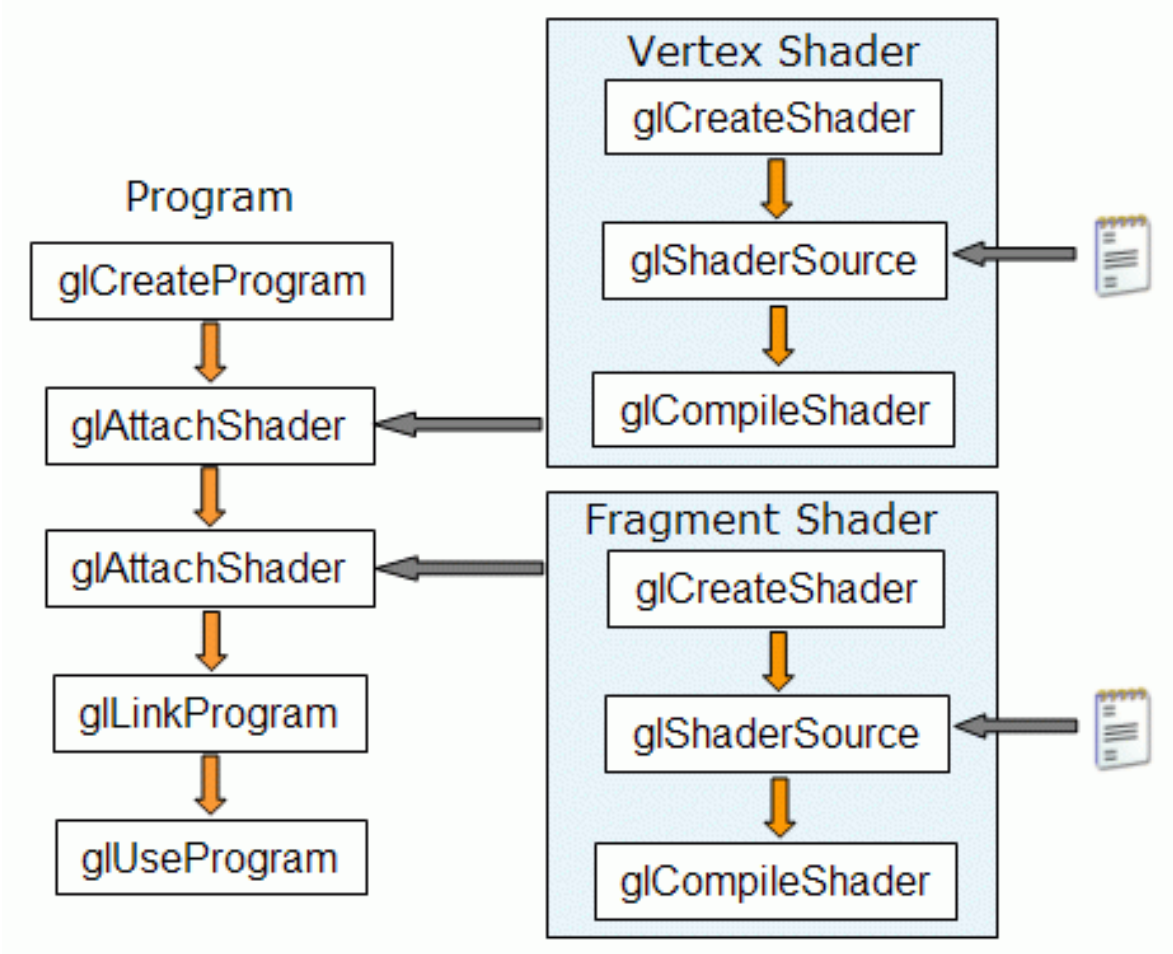
Shader езици

- Четири различни варианта
  - Асемблер
    - старо решение, не се прилага често и не се обновява
  - Cg
    - “C for graphics”, NVidia
  - HLSL
    - “High-level shading language”, Microsoft
  - GLSL
    - “OpenGL shading language”
- Изборът зависи от платформата, нуждите и предпочитанията на програмиста
  - GLSL е с най-добра съвместимост

GLSL

- Част от стандарта OpenGL 2.0
- Език от високо ниво, подобен на C
- Нови типове данни
  - Matrices
  - Vectors
  - Samplers
- Вградени променливи и функции
- Всеки shader има main функция като входна точка
- Компилатор за GLSL вграден в драйвера на графичната карта

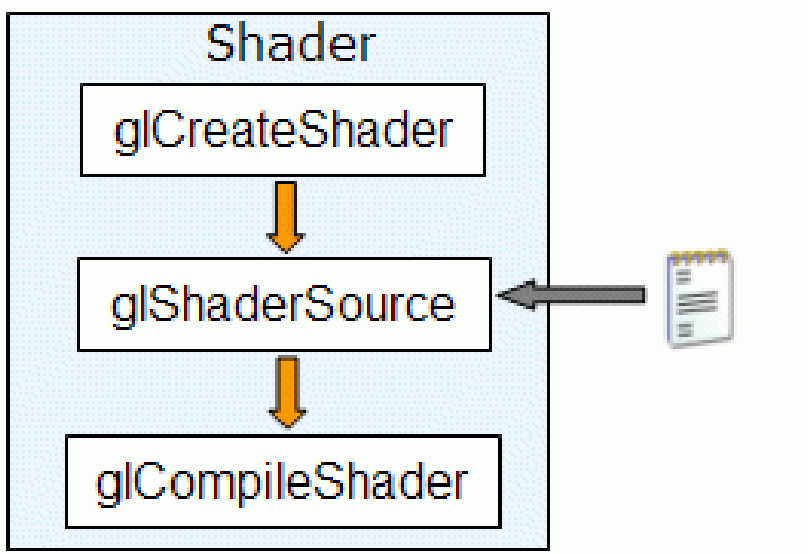
GLSL


Vertex Shader
glCreateShader
glShaderSource
glCompileShader
Program
glCreateProgram
glAttachShader
glAttachShader
glLinkProgram
glUseProgram
Fragment Shader
glCreateShader
glShaderSource
glCompileShader

Създаване на shader

- Създава се контейнер за shader
  - връща handle за shader
- Задава се сорс на shader
  - масив от символи

    или указател към него
- Компилира се



Създаване на shader

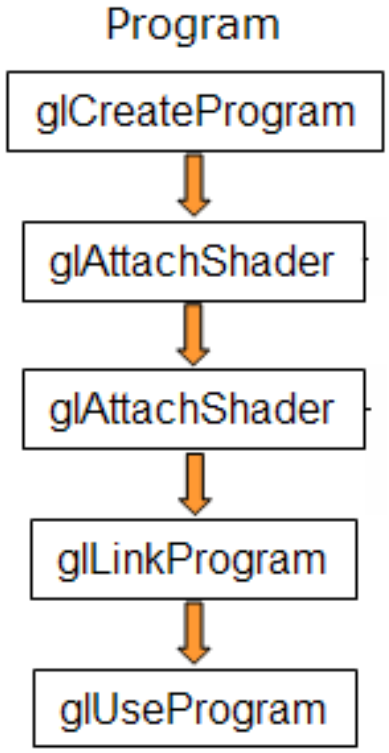
```
void glShaderSource (
     GLuint shader,
     int numOfStrings,
     const char **strings,
     int *lenOfStrings);
```

- shader – handler за shader
- numOfStrings – дължина на масива от символи
- strings – масив от символи
- lenOfStrings – масив с дължината на всеки стринг

Използване на shader

- Създава се контейнер за програма
  - могат да се използват няколко и да се превключват
- Задава се сорс на shader
  - масив от символи

    или указател към него
- Прилага се shader
  - задава се handler
- Свързва се
  - сорсът трябва да е компилиран
- Изпълняват се в програмата всички приложени shaders



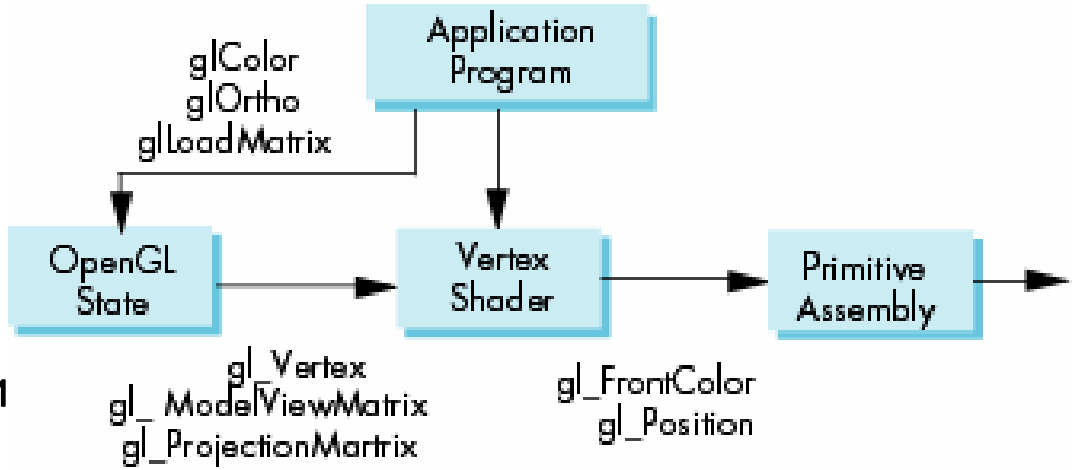
Използване на shader

```
void setShaders() {

    char *vs,*fs;

    v = glCreateShader
        (GL_VERTEX_SHADER);
    f = glCreateShader
        (GL_FRAGMENT_SHADER);
    vs = textFileRead
        ("toon.vert");
    fs = textFileRead
        ("toon.frag");

    const char * vv = vs;
    const char * ff = fs;
```

```
glShaderSource(v, 1, &vv, NULL);
glShaderSource(f, 1, &ff, NULL);

free(vs); free(fs);

glCompileShader(v);
glCompileShader(f);

p = glCreateProgram();

glAttachShader(p,v);
glAttachShader(p,f);

glLinkProgram(p);
glUseProgram(p);
}
```

Vertex Shader

- ***Вход***: атрибути per-vertex
  - Positions: glVertex
  - Normal: gl_Normal
  - Color: gl_Color
  - Texture coordinates
- Достъп до глобални данни
  - OpenGL states
  - Texture
- ***Изход***: интерполирани стойности от растеризирането
  - Position (required): gl_Position
    - 2D clip координати в канонична КС
  - Други стойности (optional): gl_FrontColor



Vertex Shader

- Заменя геометричните изчисления за възли във фиксирания конвейер
- данни per-vertex
  - координати на възел (glVertex)
  - вектор на нормала
  - текстурни координати
  - цвят RGBA
  - други данни: цветови индекси, флаг за контур
  - допълнителни потребителски дефинирани данни
- операции per-vertex
  - трансформации чрез матриците model-view и projection
  - изчисляване на осветеност
  - …

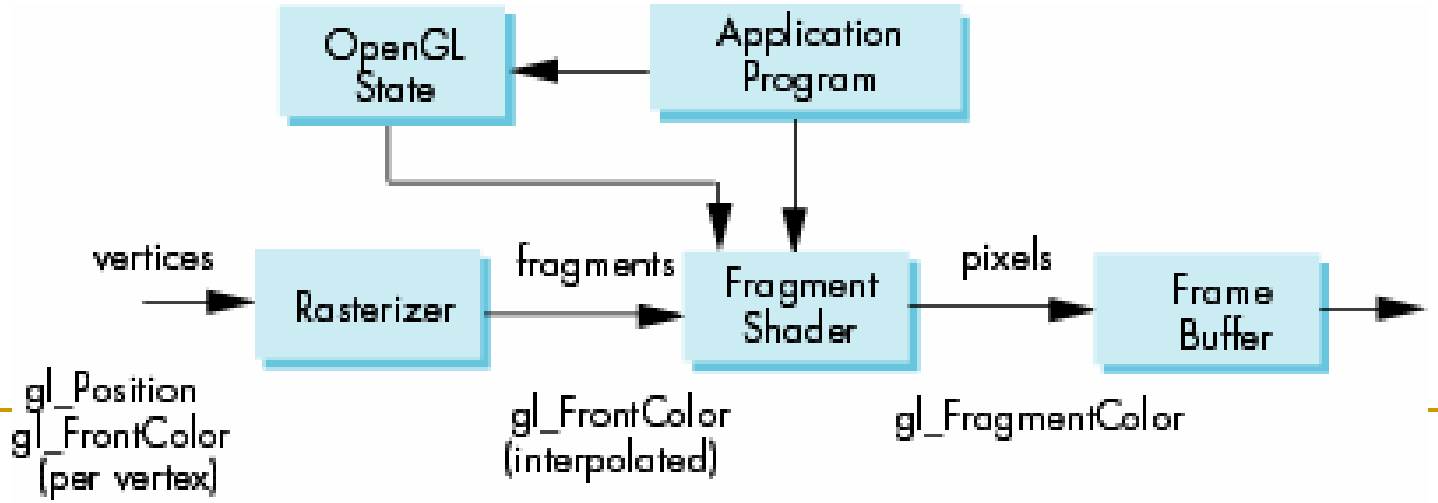
Приложения на Vertex Shader

- Трансформиране на възли
  - Morphing
  - Wave motion
  - Fractals
  - Particle systems
- Осветеност
  - реалистични модели за осветеност
  - Cartoon shaders
- Пример за Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main(void)
{
  gl_Position = ftransform();
   gl_FrontColor = red;
}
```

Fragment Shader

- ***Вход***: фрагменти получени след растеризиране
- Операции с фрагменти
  - texture mapping, fog, anti-aliasing, Scissoring, Blending
- ***Изход***: атрибути на фрагмент
  - Color: gl_FragColor
  - Depth-value: gl_FragDepth
- Фрагментите преминават през операции за alpha test, depth test и др. преди да бъдат записани във frame buffer



Приложения на Fragment Shader

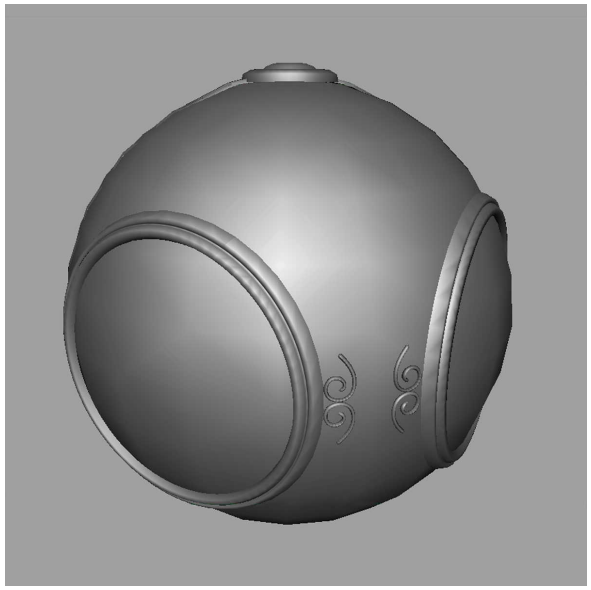
- Изчисляване на осветеност per fragment

per vertex lighting

per fragment lighting

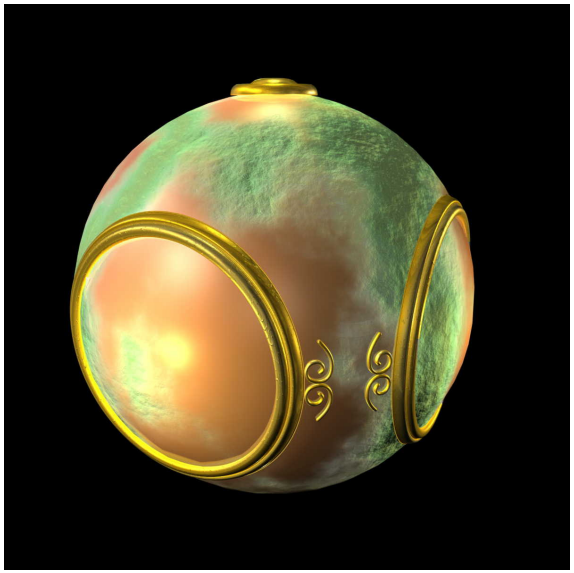
Приложения на Fragment Shader

- Texture mapping



smooth shading

environment mapping



bump mapping

КРАЙ