Компьютерная графика
Москва
Лекция 12
01/02 декабря 2011

Программируемая графическая аппаратура

Алексей Игнатенко, к.ф.-м.н.

Лаборатория компьютерной графики и мультимедиа ВМК МГУ

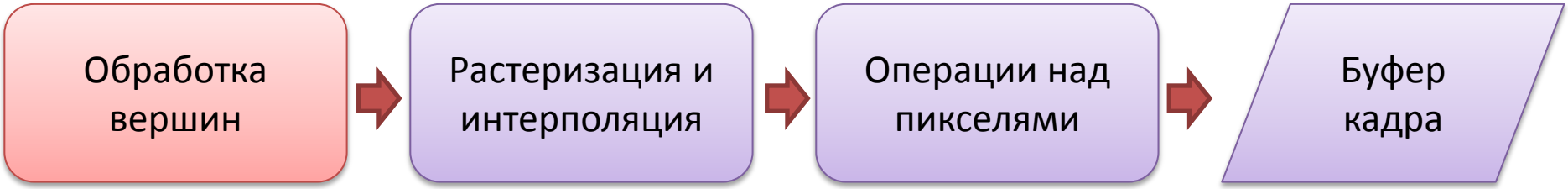
План лекции

- История появления и развития GPU
- Программируемый графический конвейер
- Языки программирования шейдеров GLSL, HLSL, Cg.
- Вычисления общего назначения. CUDA, OpenCL
- OpenGL 3+

План лекции

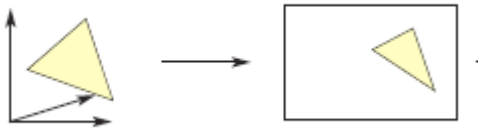
- **История появления и развития GPU**
- Программируемый графический конвейер
- Языки программирования шейдеров GLSL, HLSL, Cg.
- Вычисления общего назначения. CUDA, OpenCL
- OpenGL 3+

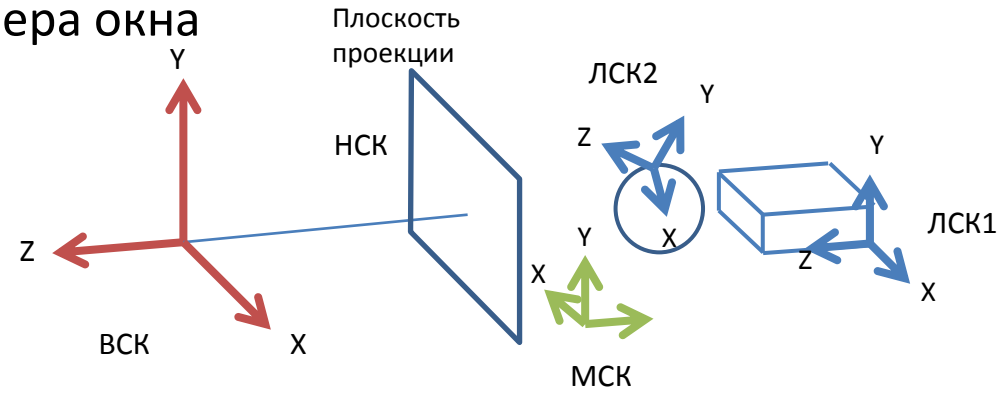
Конвейер: Обработка вершин



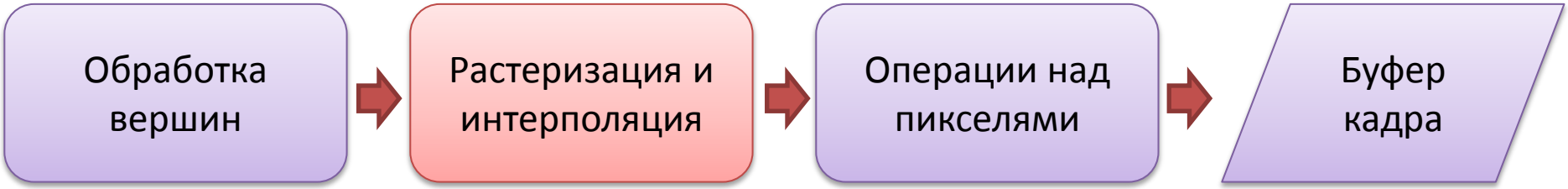
Задача: преобразовать и получить проекцию вершин на экран (x,y,z) -> (x’,y’)

- Обычно три последовательных преобразования:
  - модельное преобразование (ЛСК -> МСК)
  - видовое преобразование (МСК -> ВСК)
  - проективное преобразование (ВСК -> НСК)
- 4x4 матрицы
- Из НСК в ЭСК с учетом размера окна



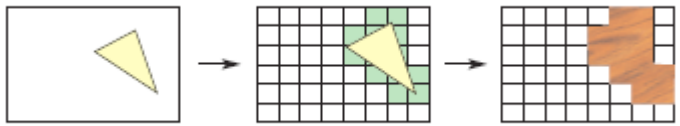


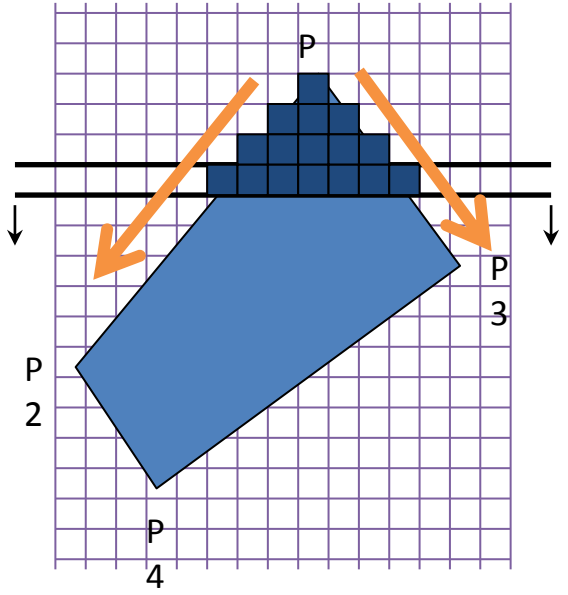
Конвейер: Растеризация и интерполяция



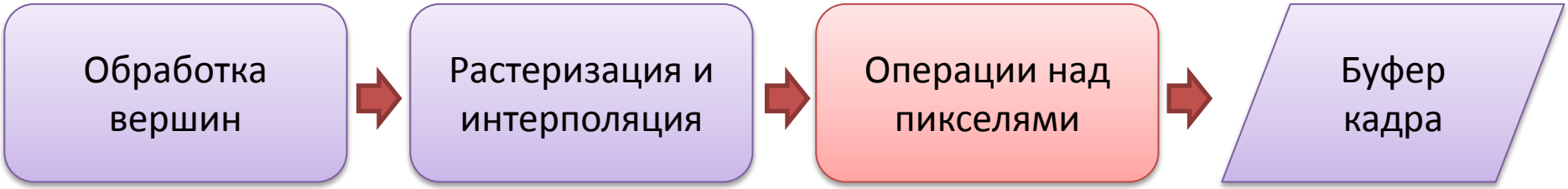
Цель: из вершин в НСК получить массив пикселей

- Интерполяция: цвет (+альфа-канал), текстурные координаты, глубина



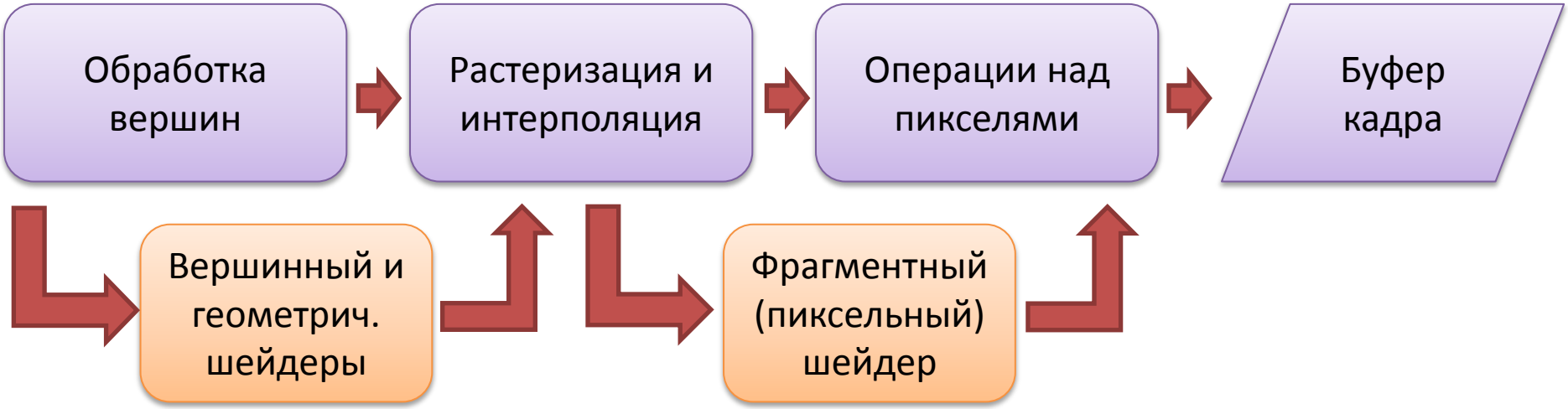


Конвейер: Операции на пикселями



- Цель: нарисовать пиксель в буфере кадра, скомбинировать с текущим содержимым
- Включает в себя z-тест, альфа-тест, stencil-тест…

Программируемый конвейер



- Шейдеры исполняются на графическом процессоре (GPU)
- Используется один из специальных языков
  - Использование возможностей GPU

Графическое процессорное устройство (GPU)

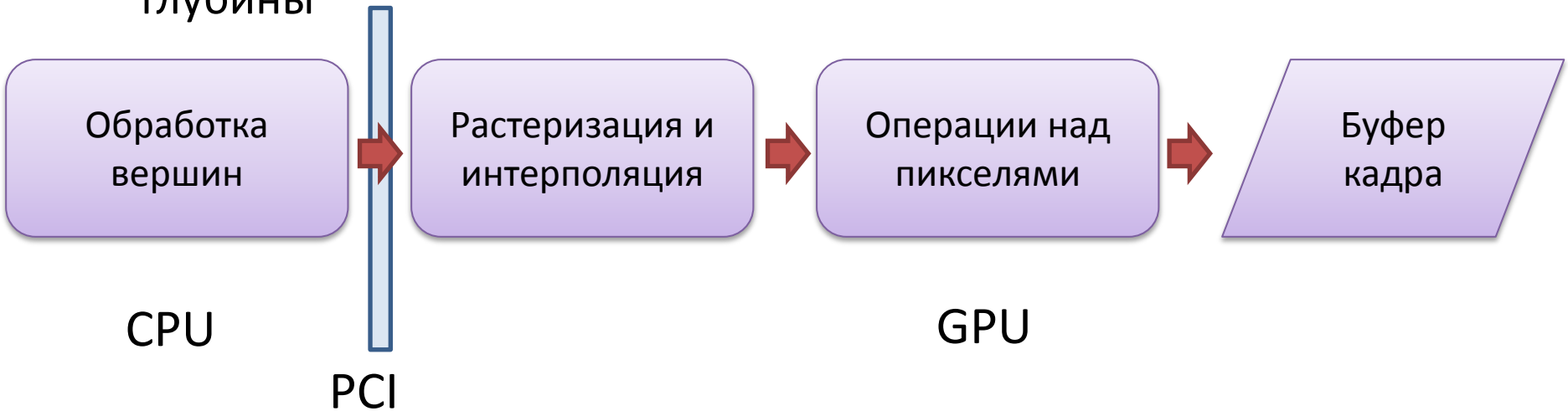
Графическое процессорное устройство
(Graphics Processing Unit, GPU)

Программируемое вычислительное устройство, предназначенное для обработки графической информации

Первое поколение

- Первые настоящие 3D-акселераторы
- Некоторые работают в комплексе с обычной 2D-видеокартой
- Трансформация вершин выполняется на CPU
- Поддержка растеризации, фильтрации текстур и буфера глубины

3Dfx Voodoo I (1996 г.)



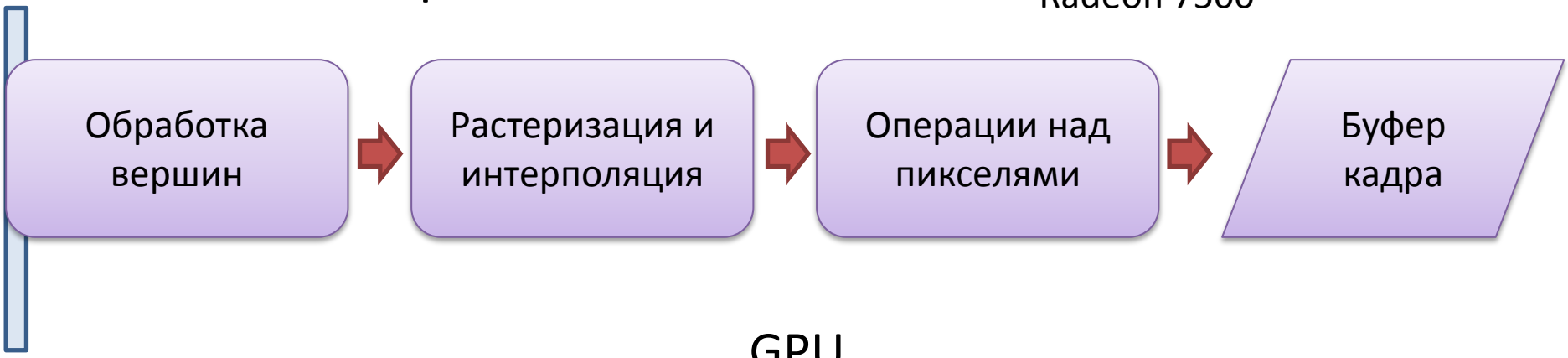
Первое поколение GPU в игровой индустрии

Half-Life и Unreal

Второе поколение

- Внедрение технологии Transform&Lighting (T&L)
- Расчет динамического освещения на GPU
- Поддержка мультитекстурирования
- Вытеснение стандарта PCI более быстрым AGP

nVidia GeForce 256 (1999 г.)
Radeon 7500

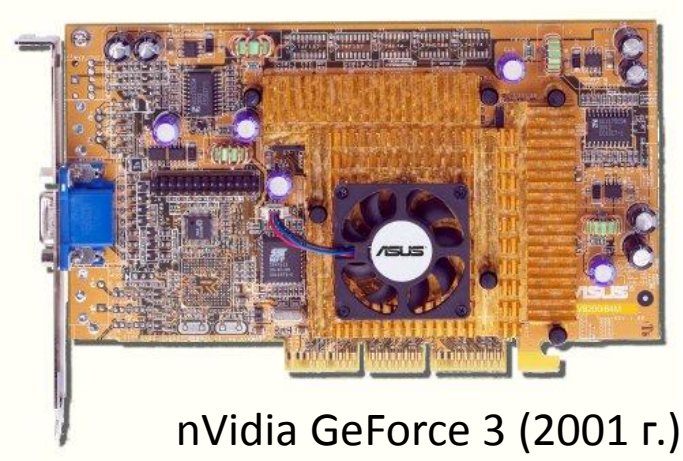


Второе поколение GPU в игровой индустрии

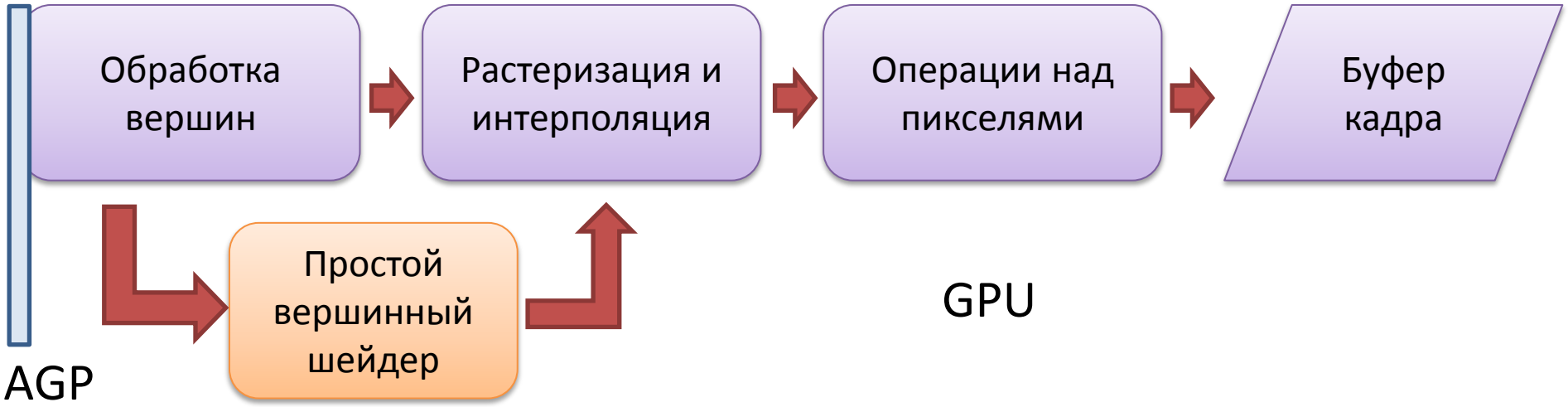
Quake III Arena и Unreal Tournament

Третье поколение

- Добавились возможности программирования
- Появление первых вершинных шейдеров



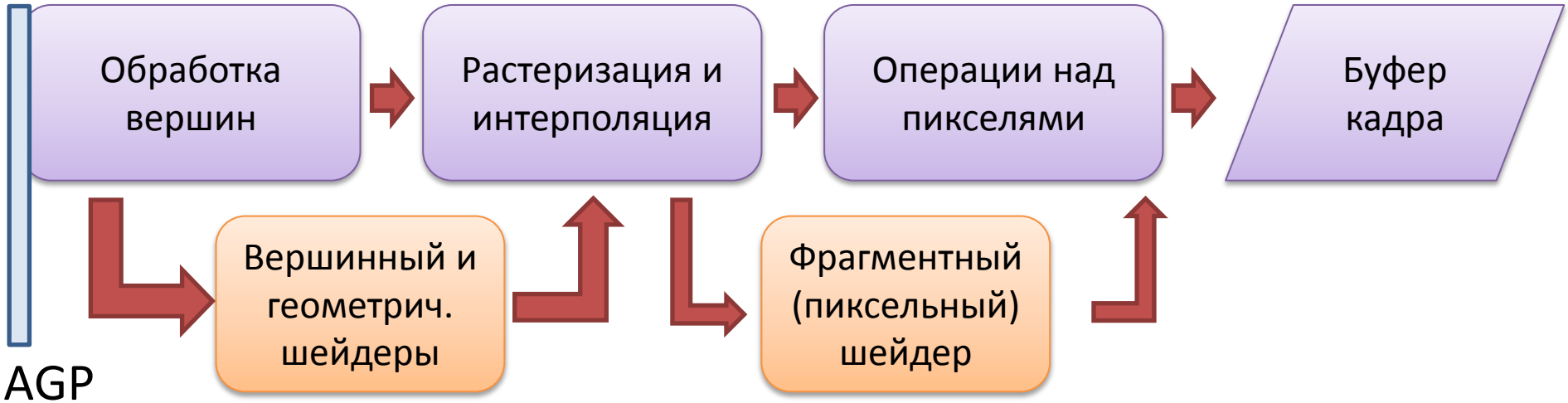
nVidia GeForce 3 (2001 г.)
Radeon 8500



Четвертое поколение

- Первое поколение полностью программируемых GPU
- Усложнение шейдеров
- Начинает складываться направление GPGPU
- Вытеснение стандарта AGP более быстрым PCI Express

ATI Radeon 9700 (2003 г.)
GeForce FX



Четвертое поколение GPU в игровой индустрии

Скриншоты из игр Half-Life 2 и Doom 3 соответственно

Пятое поколение

- Поддержка геометрических шейдеров
- Полная поддержка унифицированной шейдерной архитектуры
- GPU получили возможность аппаратно ускорять физические расчеты
- Появление средств программирования для GPGPU
  - nVidia CUDA
  - AMD FireStream

GeForce 8 (2006)

ATI Radeon X1950

Пятое поколение GPU в игровой индустрии

Crysis и Call of Duty: Modern Warfare 2

Будущее GPU на ближайшие годы

Достигнуть уровня кино

- Прямое освещение
- Тени
- Частицы (взрывы и т.п.)
- Отражения

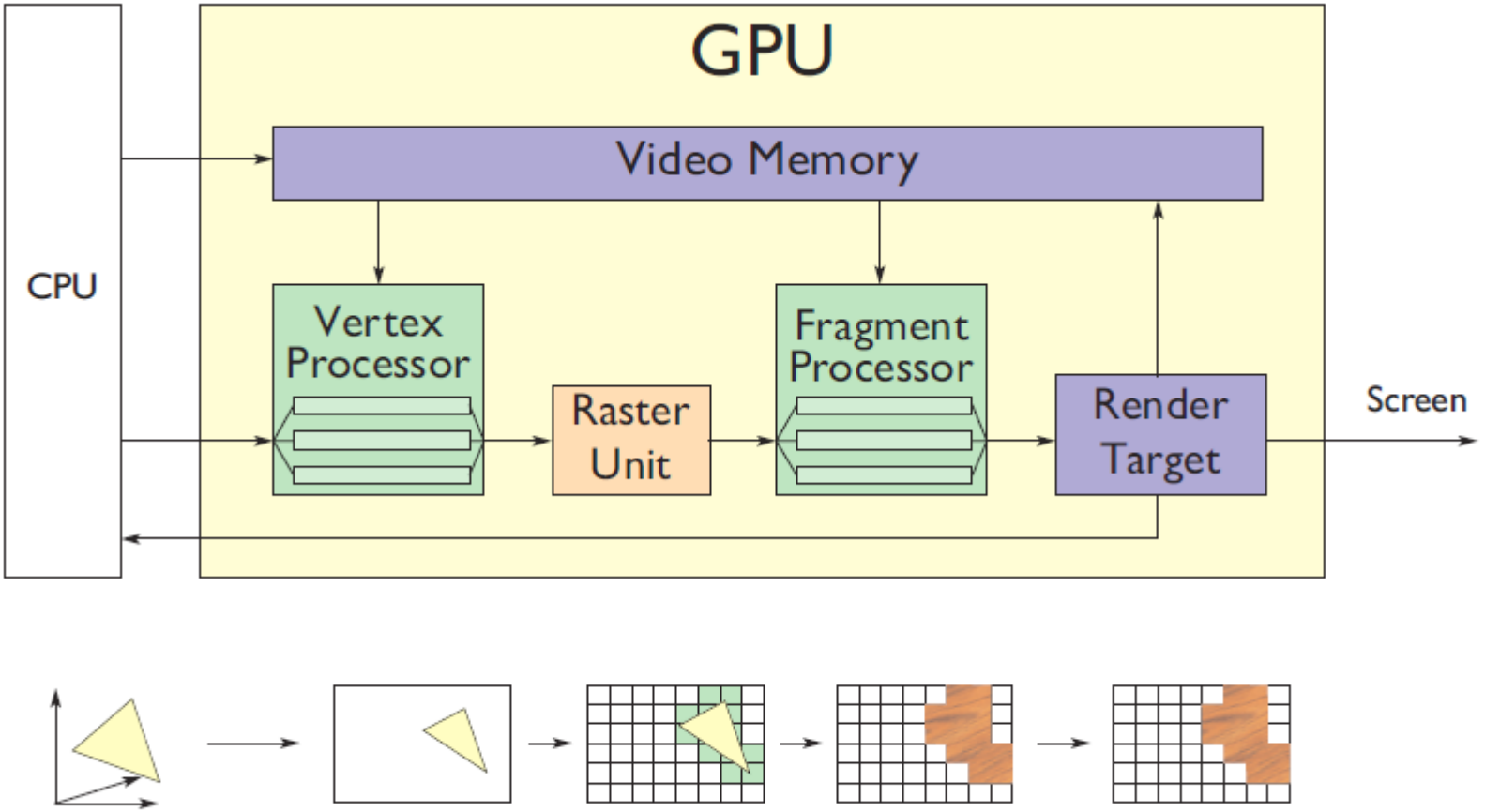
Улучшить:

- Анимацию
- Детализацию объектов
- Вторичное освещение

План лекции

- История появления и развития GPU
- **Программируемый графический конвейер**
- Языки программирования шейдеров GLSL, HLSL, Cg.
- Вычисления общего назначения. CUDA, OpenCL
- OpenGL 3+

Современный Графический конвейер



Основные свойства

- Строго последовательное исполнение (конвейер)
- Текущий фрагментный / вершинный шейдер выполняется на GPU для каждого фрагмента / вершины
- Все вычисления во floating-point
- Работа идет над своими примитивами для каждого типа шейдеров
  - Вершинный: вершина
  - Фрагментный: фрагмент
- Шейдерные программы называется ядрами (kernel)

Мелкозернистый параллелизм

- Каждый примитив рассматривается как независимый
- Поток выполнения для каждого примитива рассматривается как (почти) одинаковый
- Примитивы обрабатывается параллельно
- Планирование выполнения не контролируется программистом
- Нельзя одновременно читать и писать в буферы

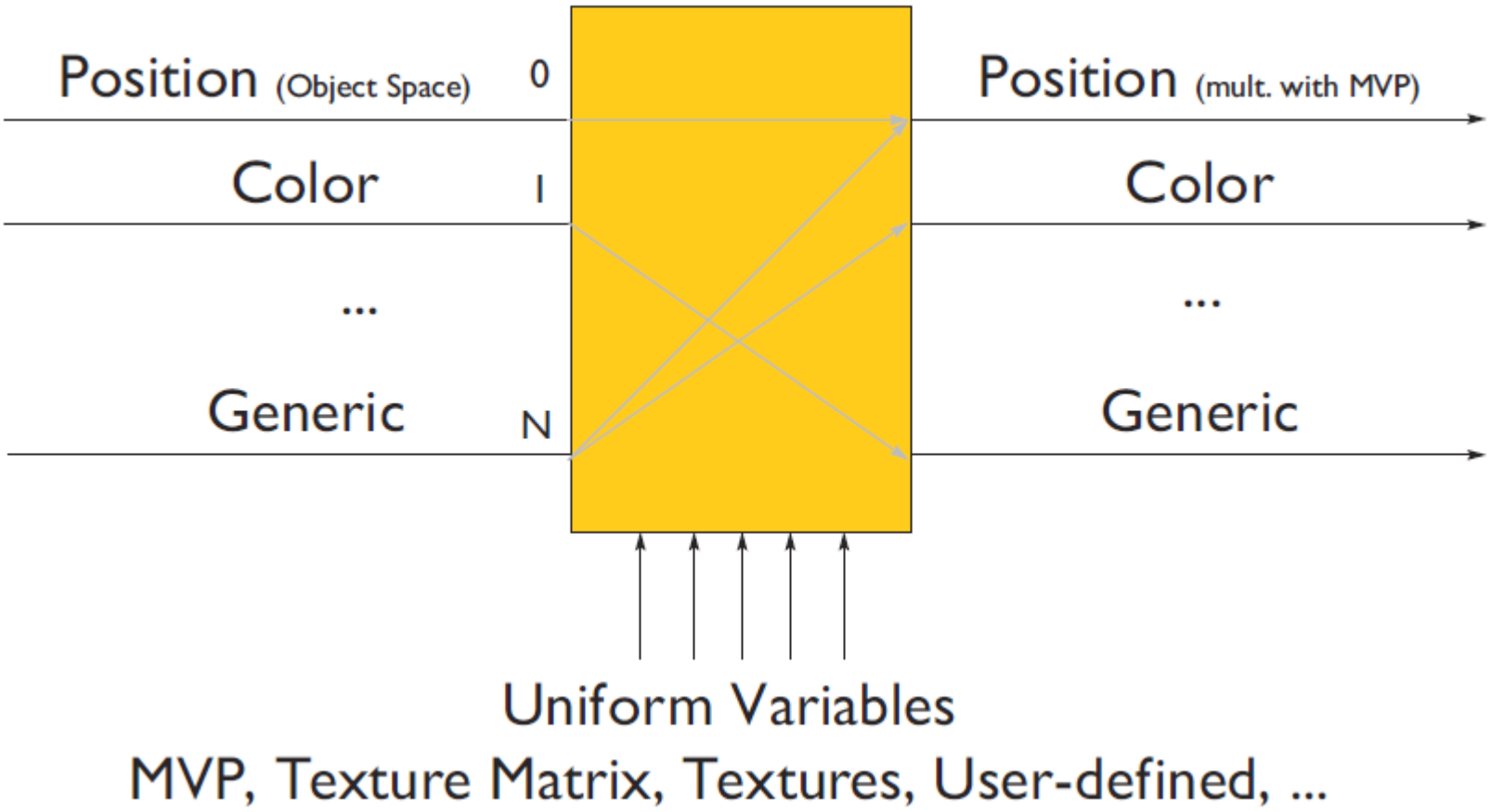
Параметры шейдерных программ: uniform

- Меняются редко
- Доступны на всех этапах конвейера
- Задаются пользователем
- Примеры:
  - Модельно-видовая матрица
  - Текстуры
  - Параметры модели освещения (тонирования)

Varying-параметры

- Меняются для каждого примитива
- Доступность зависит от этапа конвейера
- Могут интерполироваться в модуле растеризации
- Определяются пользователем
- Примеры:
  - Положение вершины
  - Цвет вершины
  - Цвет фрагмента

Вершинный процессор



Типичная функциональность

- Преобразования вершин
- Пребразования и вычисления нормалей и текстурных координат
- Освещение

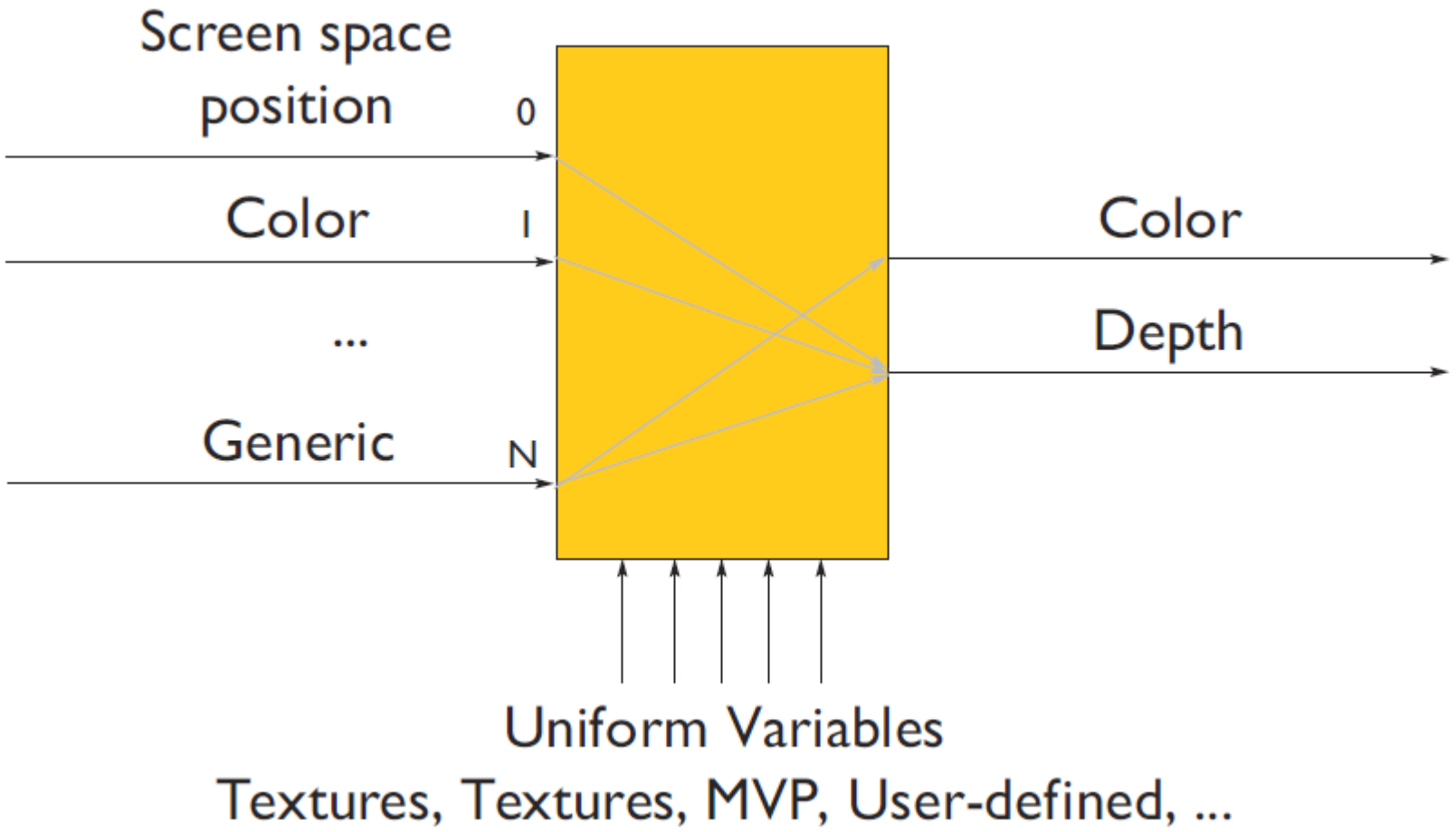
- Генерация положения вершин
- Генерация текстурных координат

Модуль растеризации

- Перспективное деление
- Сборка примитивов
- Отсечение
- Интерполяция всех варьируемых параметров
- Растеризация
- Раннее z-отсечение

- **Не программируется!**

Фрагментный процессор



Функциональность

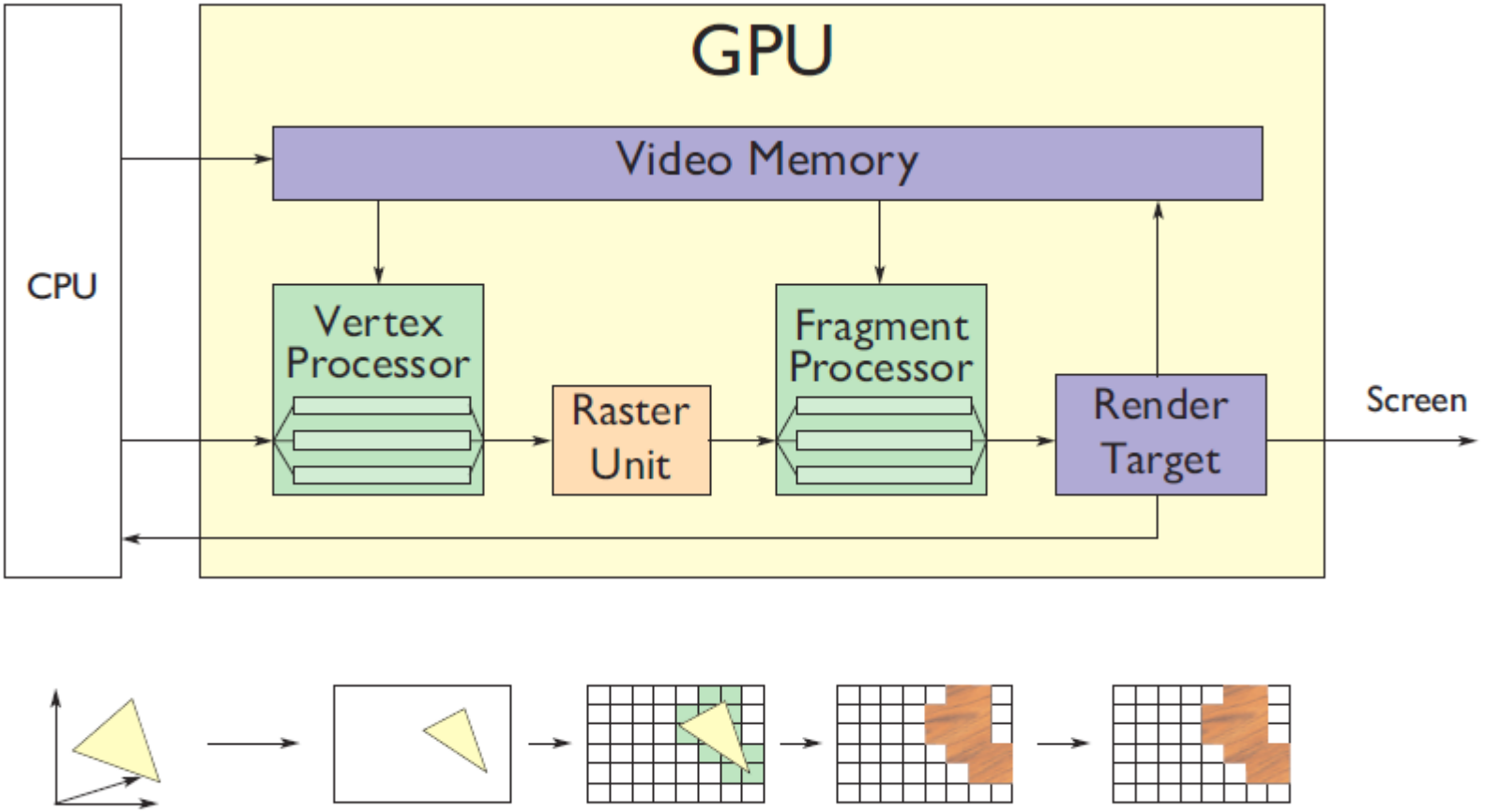
- Крайне высокая мощность вычислений в вещественных числах
- Фиксированное положение фрагмента
- Вычисление или изменение z-координаты возможно
- Выборки из текстур
  - Texture sampler
  - Зависимые выборки из текстур
  - 1D, 2D, 3D текстуры
  - Явная адресация уровней мипмапа

Буфер кадра

- Может быть экранным или внеэкранным буфером
- До четырех буферов цвета (MRT)
- Буфер глубины, маски (stencil), аккумулятор (accumulation)

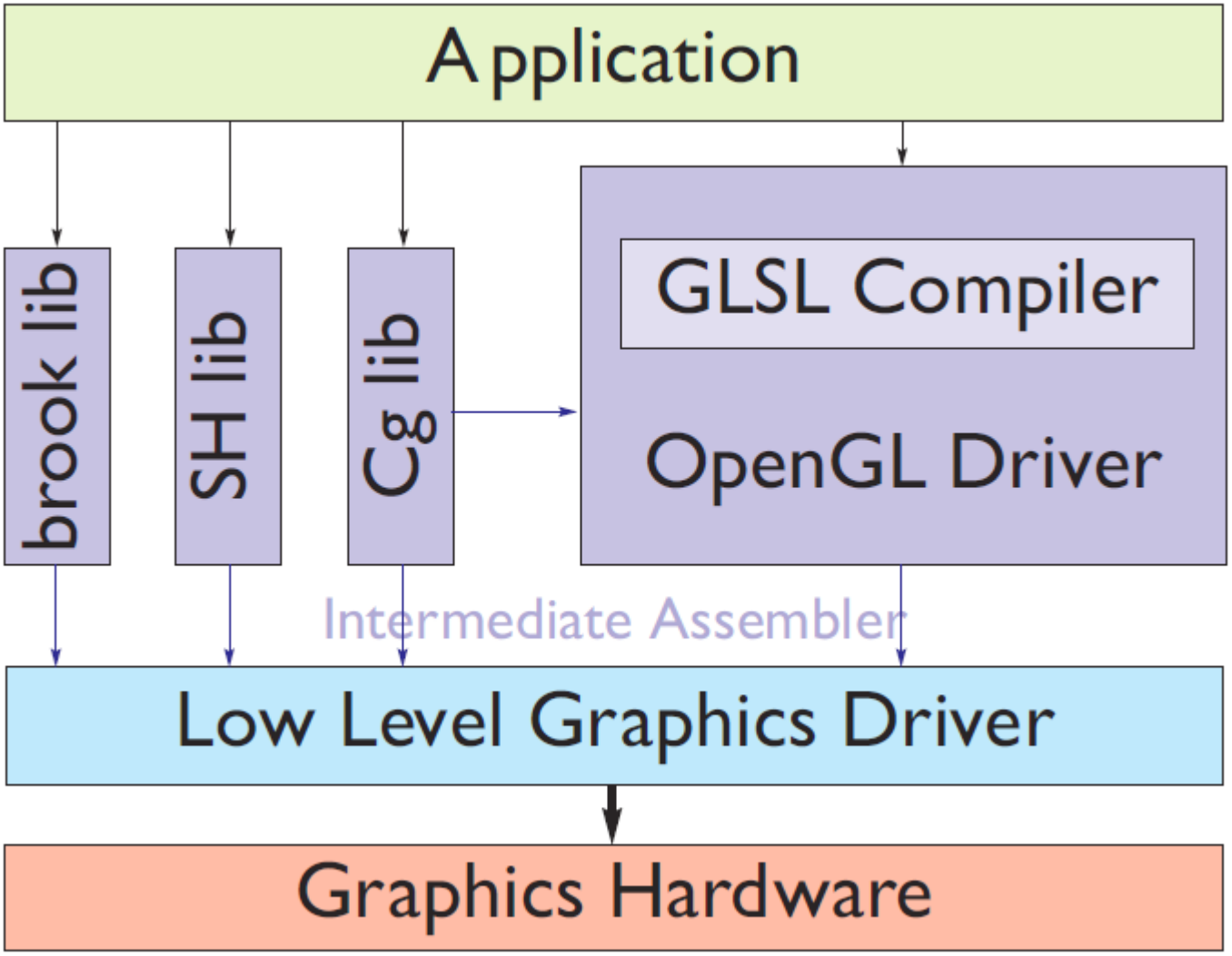
- Экранный буфер – предоставляется системой
- Внеэкранный буфер – Framebuffer Object (FBO)

Современный Графический конвейер



План лекции

- История появления и развития GPU
- Программируемый графический конвейер
- **Языки программирования шейдеров GLSL, HLSL, Cg.**
- Вычисления общего назначения. CUDA, OpenCL
- OpenGL 3+


Application
brook lib
SH lib
Cg lib
GLSL Compiler
OpenGL Driver
Intermediate Assembler
Low Level Graphics Driver
Graphics Hardware

Общие соображения по языкам

- Основаны на C / C++
- Включаются в себя расширения для поддержки графики
- Ограничены по сравнению с C / C++

- Рассмотрим:
  - GLSL
  - Cg
  - HLSL

GLSL

- GLSL: OpenGL Shading Language
- Часть OpenGL 2.0 и далее (3.3, 4.2)
- Разработан компанией 3D Labs, поддерживается ARB

Cg

- Gg: C for Graphics
- Разработан и поддерживается Nvidia
- Не зависит от графического API
- Закрытый, но может быть использован с видеокартами AMD

HLSL

- HLSL: High Level Shading Language
- Разработан и поддерживается Microsoft
- Язык шейдеров для DirectX

Расширения для графики

- Типы данные для выборок из текстур
  - sampler1D, sampler2D, samplerRECT, sampler3D
  - texture1D, texture2D
  - texture2DShadow (специальные выборки для наложения теней)

- Дополнительные спецификаторы для типов
  - attribute
  - uniform
  - Varying

- Встроенные типы данные для векторов и матриц
  - vec2, vec3, vec4, ivec2, ivec3, bvec2, bvec3…
  - mat2, mat3, mat4

Расширения для графики

- Встроенные функции для часто применяемых операций
  - operator*(mat, mat), operator*(mat, vec)
  - operator+
  - dot, cross, normalize, length
- Discard (обрывание дальнейшей обработки фрагмента)
- Специальные функции для булевских операцией над векторами
  - any(), all()

Ограничения по сравнению с C/C++

- Нет double, unsigned int..
- Нет строк, символов
- Нет динамического выделения памяти и указателей
- Нет enum, union
- Нет классов и шаблонов
- Нет битовых операций
- Нет стека для регистров (нет переключения контекста, настоящих вызовов функций)
- Нет goto
- If, for доступны, но намного дороже, чем для CPU

Пример: вершинный шейдер

```
#version 330 core

uniform mat4 projectionMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 viewMatrix;
uniform mat3 normalMatrix;

in vec3 position;
in vec3 normal;

out vec3 fragmentNormal;
out vec3 fragmentEye;
out vec3 lightLocationEye;

const vec4 lightLocation = vec4(0., 100., 0., 1.);
```

Пример: вершинный шейдер

```
void main(void)
{
  fragmentEye = (modelViewMatrix * vec4(position, 1.0)).xyz;
  fragmentNormal = normalMatrix * normal;
  lightLocationEye = (viewMatrix * lightLocation).xyz;

  gl_Position = projectionMatrix * modelViewMatrix *
                vec4(position, 1.0);

 }
```

Пример: фрагментный шейдер

```
#version 330 core

in vec3 fragmentNormal;
in vec3 fragmentEye;
in vec3 lightLocationEye;
out vec4 fragmentColor;

uniform vec4 materialKd;

const vec4 ka=  vec4(0.05, 0.05, 0.05, 1);
const vec4 ks = vec4(0.4, 0.4, 0.4, 0.4);
const float kp = 20;
```

Пример: фрагментный шейдер

```
void main(void)
{
  vec3 light = -normalize(fragmentEye - lightLocationEye);
  vec3 normal = normalize(fragmentNormal);
  vec3 eye = normalize(fragmentEye);

  float diffuseIntensity = clamp(max(dot(normal, light), 0.0),
                                 0.0, 1.0);
  vec3 reflection = normalize(reflect(light, normal));
  float specularIntensity = pow(clamp(max(dot(reflection, eye),
                                 0.0), 0.0, 1.0), kp );
  fragmentColor = ka + materialKd * diffuseIntensity +
                  ks * specularIntensity;
}
```

Как использовать шейдеры

1) формируем текст шейдеров

```
char *vsSource = file2string("wave.vert");
char *fsSource = file2string("wave.frag");
```

Как использовать шейдеры

2) Компилируем и загружаем программы на GPU

```
GLuint vs, /* Vertex Shader */
fs, /* Fragment Shader */
sp; /* Shader Program */


vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vsSource, NULL);
glCompileShader(vs);

fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fsSource, NULL);
glCompileShader(fs);
```

Как использовать шейдеры

3) Собираем шейдерную программу

```
sp = glCreateProgram();
glAttachShader(sp, vs);
glAttachShader(sp, fs);
glLinkProgram(sp);
printLog(sp);
```

4) Активируем программу для рендеринга

```
glUseProgram(sp);
```

Как использовать шейдеры

5) Задаем uniform-переменные

```
GLint waveTimeLoc = glGetUniformLocation(sp, "waveTime");
GLint waveWidthLoc = glGetUniformLocation(sp, "waveWidth");
GLint waveHeightLoc = glGetUniformLocation(sp, "waveHeight");

glUniform1f(waveTimeLoc, waveTime);
glUniform1f(waveWidthLoc, waveWidth);
glUniform1f(waveHeightLoc, waveHeight);
```

6) Передаем геометрию на исполнение

```
glVertex3f(…)
…
```

План лекции

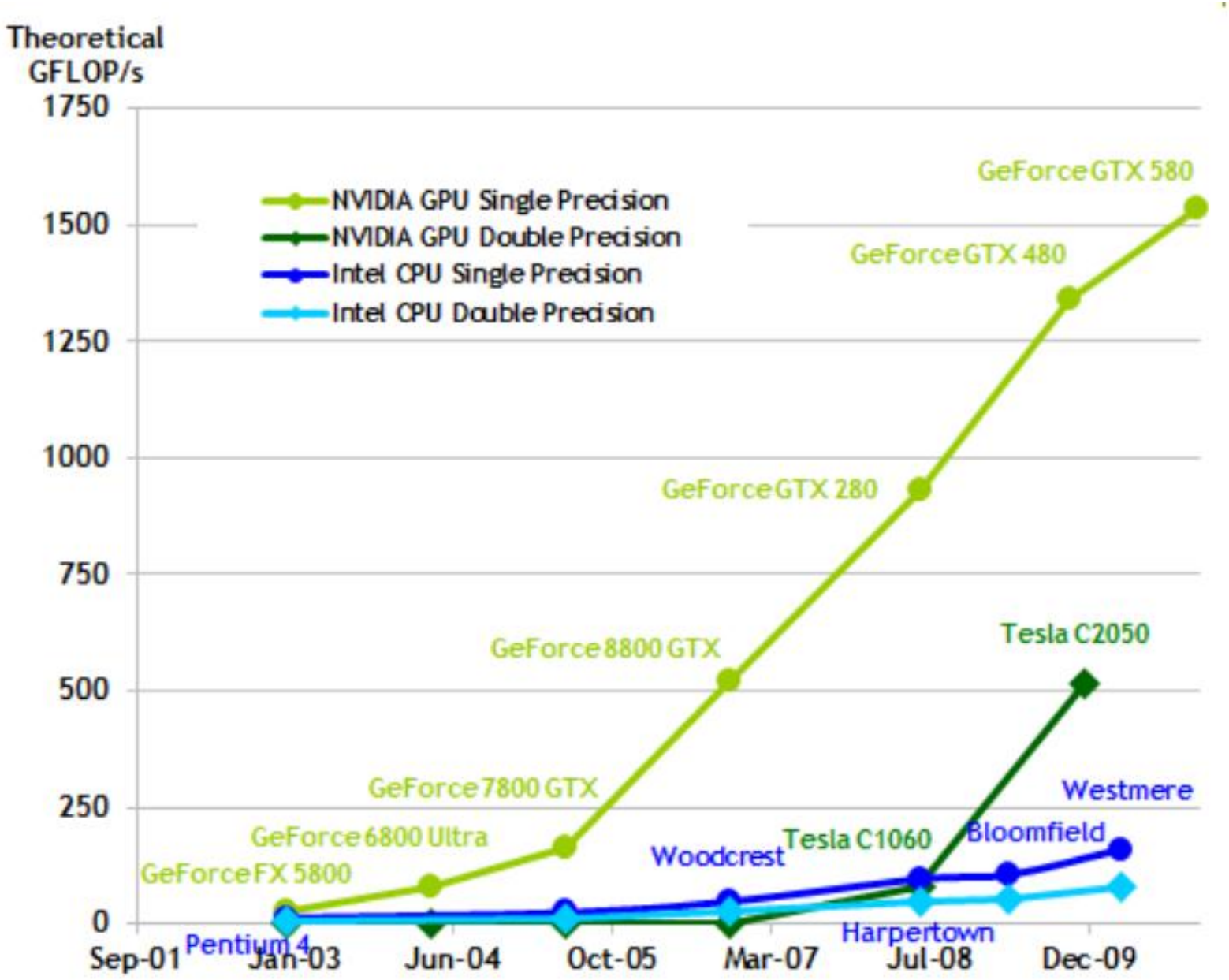
- История появления и развития GPU
- Программируемый графический конвейер
- Языки программирования шейдеров GLSL, HLSL, Cg.
- **Вычисления общего назначения. CUDA, OpenCL**
- OpenGL 3+

GPGPU: What

- GPGPU: General Purpose GPU Computations
- Цель – использование GPU как стандартный вычислительный элемент

GPGPU: Why

- Современные CPU содержат до 12-ти ядер на чипе (AMD Opteron 6000)
- GPU содержат сотни ядер (240 для Tesla)
- Производительность каждого небольшая, но в сумме GPU очень хорошо подходят для решения массивно-параллельных задач



© Nvidia CUDA Programming guide

GPGPU: Who

- NVidia: CUDA
- AMD (ATI): Stream
- OpenCL
- Microsoft: DX 11 Compute shaders

GPGPU: пример (C++)

```
void compute(float A[N][N], float B[N][N], float
C[N][N])
{
  for (int i=0;i<N;i++)

    for (int j=0;j<N;j++)

      C[i][j] = A[i][j] + B[i][j];
}

int main(){
  compute(A,B,C);
}
```

GPGPU: пример (CUDA)

```
__global__ void compute(float A[N][N], float
B[N][N], float C[N][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}

int main(){
  //call the GPU-kernel
  compute<<N_blocks,N_threads>>(A, B, C);
}
```

GPGPU: приложения и проблемы

- Не универсальная технология
- Выигрыш получают приложения, которые могут быть сильно распараллелены
- Проблемы с доступом к памяти
- Необходимость переписывать все алгоритмы

План лекции

- История появления и развития GPU
- Программируемый графический конвейер
- Языки программирования шейдеров GLSL, HLSL, Cg.
- **Вычисления общего назначения. CUDA, OpenCL**
- OpenGL 3+

План

- История последних версий OpenGL
- Отличия работы с API GL 1.x-2.x и 3.x-4.x
- Поддержка видеокартами спецификаций OpenGL
- Демонстрация: сравнение рисования прямоугольника на GL 1.1 и GL 3.3 Core Profile
- **OpenGL ES и WebGL**

Почему мы уделяем столько внимания различным версиям OpenGL?

- Одновременно встречается много версий (в том числе в книгах)
- При разработке надо понимать, какие видеокарты будут поддерживать используемую функциональность

OpenGL 2.1

- 2006й год
- Первый релиз, добавлящий поддержку GLSL в ядро OpenGL (ранее были доступны только в виде расширений)
- Полностью поддерживается FF
- Можно смешивать FF и шейдеры, что создает проблемы для больших приложений

OpenGL 3.0-3.3

- 2008й год
- Появляются профили (profiles)
  - Core-Profile
  - Compatibility-Profile
- Core-Profile: доступ только к функциям OpenGL 3.2+
- Compatibility-Profile: возможно использование всех функций GL 1.0+ (FF)

OpenGL 4.0-4.2

- Сохраняет философию GL 3.0+
- Включает новый типы шейдеров для аппаратной тесселяции (подразбиения геометрии, визуализацию сплайновых поверхностей)
- Используется одновременно с GL 3.0+ с помощью указания нужного профиля при создании контекста (окна) OpenGL

Эволюция GL 1.x, 2.x -> GL 3.x, 4.x. Что потеряли и что приобрели

- Нет поддержки glVertex()
- Нет поддержки преобразований ModelView, Projection и т.п. Нет стека матриц.
- Нет встроенного освещения и материалов

- Теперь это нужно делать вручную
- Без шейдеров теперь нельзя нарисовать ничего

Эволюция GL 1.x, 2.x -> GL 3.x, 4.x. Что потеряли и что приобрели

- Плюсы
  - Можно сделать такой процесс, какой нужно
  - Больше гибкости
  - Выше эффективность – нет ненужных операций, проще драйвер => выше скорость
- Минусы
  - Больше кода
  - Сложная кривая обучения

Использование расширений на Windows: библиотека GLEW

- В Windows только GL 1.1 поддерживается на уровне операционной системы
- opengl.h содержит функции только для OpenGL 1.1
- Все более новые функции должны подключаться вручную!
  - Они есть в библиотеке opengl32.dll, но нужен доступ к ним через h/lib

- Надо использовать библиотеку GLEW (GL Extensions Wrangler) http://glew.sourceforge.net/

Работа с GLEW: инициализация

- Подключаем и инициализуем

```
#include <GL/glew.h> // включать перед другими заголовками GL
// #include <GL/gl.h> уже включена ранее!
void initGLEW()
{
  GLenum err = glewInit(); // инициализируем
  if (err != GLEW_OK) // ошибка?
  {
    cout << "GLEW Error: " << glewGetErrorString(err);
    exit(1);
  }
}
```

Работа с GLEW: проверка совместимости

- Проверка поддерживаемой версии

```
if (glewIsSupported("GL_VERSION_3_2"))
{
  // OpenGL 3.2 поддерживается на этой системе
}
```

- Проверка конкретного расширения OpenGL

```
if (GLEW_ARB_geometry_shader4)
{
  // Геометрические шейдеры поддерживаются на этой системе
}
```

Таблица совместимости поколений видеокарт и версий OpenGL (драйверы самые новые)

	1.1	1.2	1.3	1.4	1.5	2.0	2.1	3.0	3.1	3.2	3.3	4.0	4.1	4.2
GF2	*	*												
GF3 (III)	*	*	*											
GF4	*	*	*											
GF FX (IV)	*	*	*	*	*	*	*							
GF 6	*	*	*	*	*	*	*							
GF 7	*	*	*	*	*	*	*							
GF 8 (V)	*	*	*	*	*	*	*	*	*	*	*			
GF 9	*	*	*	*	*	*	*	*	*	*	*			
GF 100	*	*	*	*	*	*	*	*	*	*	*			
GF 200	*	*	*	*	*	*	*	*	*	*	*			
GF 300	*	*	*	*	*	*	*	*	*	*	*			
GF 400 (VI?)	*	*	*	*	*	*	*	*	*	*	*	*	*	*
GF 500	*	*	*	*	*	*	*	*	*	*	*	*	*	*

2004

2008

2010

OpenGL ES

- OpenGL ES (Embedded Systems) – подмножество стандарта OpenGL, предназначенное для мобильных и встроенных систем

- OpenGL ES 1.0
  - GL 1.3
  - Поддержка fixed-point типов
  - Много удалено: glBegin/glEnd, texgen, 3d texture, accumulation buffer, display lists…
  - Поддержка: Android, Symbian, PS3, QNX

OpenGL ES

- OpenGL ES 1.1
  - GL 1.5
  - Multitexture, mipmaps generation, VBO
  - Поддержка: Android 1.6, iPad, iPhone, BlackBerry 5.0, webOS
- OpenGL ES 2.0
  - GL 2.0
  - FF удален!
  - Нет обратной совместимости с 1.1
  - Поддержка: iPad, iPhone 3GS и позже, Android 2.2+
  - Выбран для WebGL

WebGL

- Расширение JavaScript для поддержки 3D-графики в броузерах без плагинов (HTML 5)
- WebGL 1.0 – Март 2011.
- Поддержка в броузерах:
  - FireFox 4.0+
  - Chrome 9+
  - Safari 5.1+ (disabled by default)
  - Opera 11.5 (dev build, windows)
  - IE – не поддерживается (из-за проблем с безопасностью)

Ресурсы

- http://cgm.computergraphics.ru/issues/issue18/gpuhistory
- http://cgm.computergraphics.ru/issues/issue18/atistream
- http://cgm.computergraphics.ru/issues/issue16/cuda
- http://gridtalk-project.blogspot.com/2010/07/future-of-computing-gpgpu.html
- From Voodoo to GeForce: The Awesome History of 3D Graphics