

# Компьютерная графика

Лекция 2: Растеризация, графический конвейер, шейдеры, интерполяция, аффинные преобразования

---

2023

- Растеризация – превращение геометрического примитива (точки, линии, треугольника, прямоугольника, круга, и т.д.) в набор соответствующих ему пикселей на экране/изображении

- Растеризация – превращение геометрического примитива (точки, линии, треугольника, прямоугольника, круга, и т.д.) в набор соответствующих ему пикселей на экране/изображении
- Превращение векторных данных в растровые

- Растеризация – превращение геометрического примитива (точки, линии, треугольника, прямоугольника, круга, и т.д.) в набор соответствующих ему пикселей на экране/изображении
- Превращение векторных данных в растровые
- За нас её делает OpenGL!

- Растеризация – превращение геометрического примитива (точки, линии, треугольника, прямоугольника, круга, и т.д.) в набор соответствующих ему пикселей на экране/изображении
- Превращение векторных данных в растровые
- За нас её делает OpenGL!
- Некоторые современные графические движки GPU (Unreal 5 Nanite) делают растеризацию сами с помощью compute шейдеров

# Почему растеризация?

- Исторически было очень много разных способов рисовать трёхмерные сцены; см. Sutherland et al - A Characterization of Ten Hidden-Surface Algorithms

# Почему растеризация?

- Исторически было очень много разных способов рисовать трёхмерные сцены; см. Sutherland et al - A Characterization of Ten Hidden-Surface Algorithms
- В современности есть два основных алгоритма: растеризация и трассировка лучей/путей

# Почему растеризация?

- Исторически было очень много разных способов рисовать трёхмерные сцены; см. Sutherland et al - A Characterization of Ten Hidden-Surface Algorithms
- В современности есть два основных алгоритма: растеризация и трассировка лучей/путей
- Особенности растеризации:
  - + Быстрее в теории (зависит от сцены)



# Почему растеризация?

- Исторически было очень много разных способов рисовать трёхмерные сцены; см. Sutherland et al - A Characterization of Ten Hidden-Surface Algorithms
- В современности есть два основных алгоритма: растеризация и трассировка лучей/путей
- Особенности растеризации:
  - + Быстрее в теории (зависит от сцены)
  - + Легче параллелится

# Почему растеризация?

- Исторически было очень много разных способов рисовать трёхмерные сцены; см. Sutherland et al - A Characterization of Ten Hidden-Surface Algorithms
- В современности есть два основных алгоритма: растеризация и трассировка лучей/путей
- Особенности растеризации:
  - + Быстрее в теории (зависит от сцены)
  - + Легче параллелится
  - + Лучше ложится на железо

# Почему растеризация?

- Исторически было очень много разных способов рисовать трёхмерные сцены; см. Sutherland et al - A Characterization of Ten Hidden-Surface Algorithms
- В современности есть два основных алгоритма: растеризация и трассировка лучей/путей
- Особенности растеризации:
  - + Быстрее в теории (зависит от сцены)
  - + Легче параллелится
  - + Лучше ложится на железо
  - + Лучше работает с памятью (data locality)

# Почему растеризация?

- Исторически было очень много разных способов рисовать трёхмерные сцены; см. Sutherland et al - A Characterization of Ten Hidden-Surface Algorithms
- В современности есть два основных алгоритма: **растеризация** и **трассировка лучей/путей**
- Особенности растеризации:
  - + Быстрее в теории (зависит от сцены)
  - + Легче параллелится
  - + Лучше ложится на железо
  - + Лучше работает с памятью (data locality)
  - - Сложнее добиться фотореалистичной графики

- Как растеризовать точку  $(x, y)$ ?

- Как растеризовать точку  $(x, y)$ ?

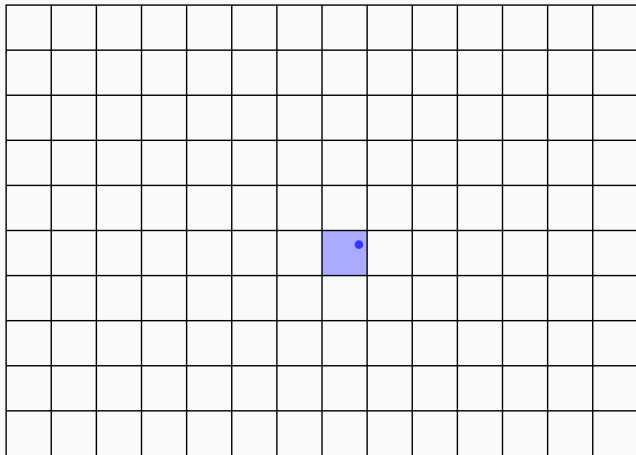
```
set_pixel(round(x), round(y), color);
```

- Как растеризовать точку  $(x, y)$ ?

```
set_pixel(round(x), round(y), color);
```

- В OpenGL: `GL_POINTS`

## Растеризация: точка





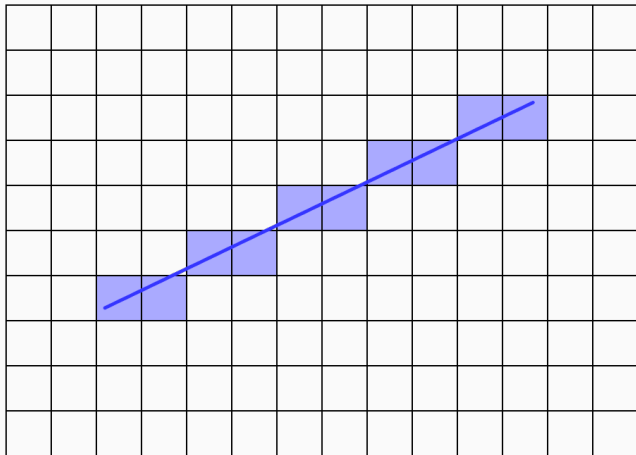
- Как растеризовать линию  $(x_1, y_1) \dots (x_2, y_2)$ ?

- Как растеризовать линию  $(x_1, y_1) \dots (x_2, y_2)$ ?
- Алгоритм Брезенхэма

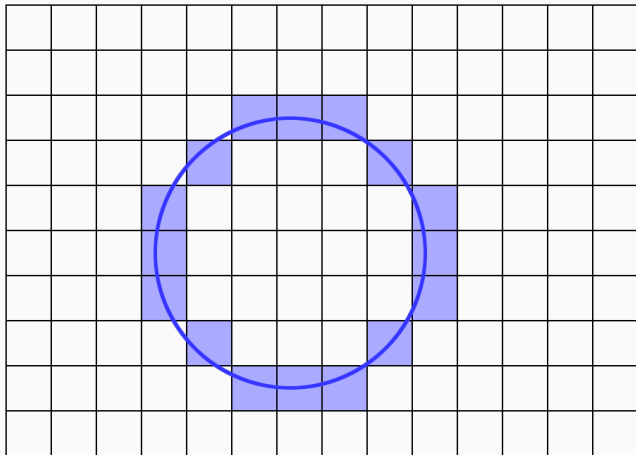
- Как растеризовать линию  $(x_1, y_1) \dots (x_2, y_2)$ ?
- Алгоритм Брезенхэма
- Есть вариация алгоритма для рисования окружностей

- Как растеризовать линию  $(x_1, y_1) \dots (x_2, y_2)$ ?
- Алгоритм Брезенхэма
- Есть вариация алгоритма для рисования окружностей
- В OpenGL: `GL_LINES`

## Растеризация: линия



## Растеризация: окружность



- Как растеризовать прямоугольник  $[x_1 \dots x_2] \times [y_1 \dots y_2]$ ?

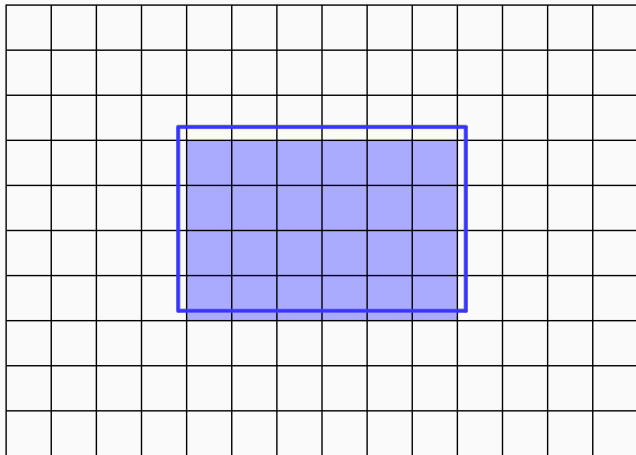
# Растеризация: прямоугольник

- Как растеризовать прямоугольник  $[x_1 \dots x_2] \times [y_1 \dots y_2]$ ?

```
for (int x = round(x_1); x <= round(x_2); ++x) {  
    for (int y = round(y_1); y <= round(y_2); ++y) {  
        set_pixel(x, y, color);  
    }  
}
```



## Растеризация: прямоугольник



## Растеризация: треугольник

- Как растеризовать треугольник с вершинами  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$ ?

## Растеризация: треугольник

- Как растеризовать треугольник с вершинами  $(x_1, y_1)$ ,  $(x_2, y_2)$ ,  $(x_3, y_3)$ ?
- Растеризуем ограничивающий прямоугольник, проверяя пиксели на вхождение в треугольник

# Растеризация: треугольник

- Как растеризовать треугольник с вершинами  $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ ?
- Растеризуем ограничивающий прямоугольник, проверяя пиксели на вхождение в треугольник

```
int xmin = min(round(x_1), round(x_2), round(x_3));
int xmax = max(round(x_1), round(x_2), round(x_3));

int ymin = min(round(y_1), round(y_2), round(y_3));
int ymax = max(round(y_1), round(y_2), round(y_3));

for (int x = xmin; x <= xmax; ++x) {
    for (int y = ymin; y <= ymax; ++y) {
        if (inside_triangle(x, y, ...))
            set_pixel(x, y, color);
    }
}
```

# Растеризация: треугольник

- Как растеризовать треугольник с вершинами  $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ ?
- Растеризуем ограничивающий прямоугольник, проверяя пиксели на вхождение в треугольник

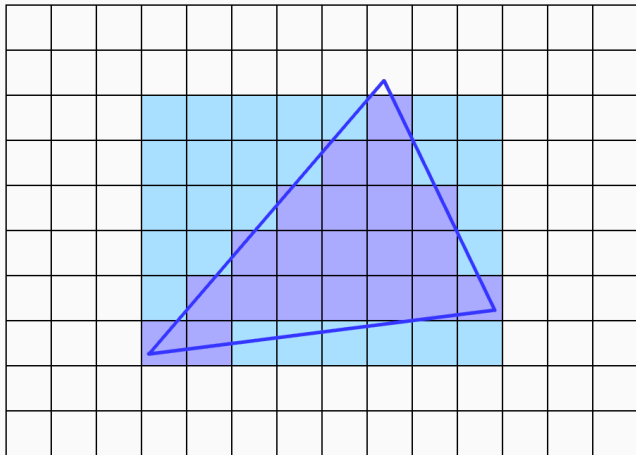
```
int xmin = min(round(x_1), round(x_2), round(x_3));
int xmax = max(round(x_1), round(x_2), round(x_3));

int ymin = min(round(y_1), round(y_2), round(y_3));
int ymax = max(round(y_1), round(y_2), round(y_3));

for (int x = xmin; x <= xmax; ++x) {
    for (int y = ymin; y <= ymax; ++y) {
        if (inside_triangle(x, y, ...))
            set_pixel(x, y, color);
    }
}
```

- В OpenGL: GL\_TRIANGLES

## Растеризация: треугольник



## Растеризация: круг

- Как растеризовать круг с центром  $(x_0, y_0)$  и радиусом  $R$ ?

## Растеризация: круг

- Как растеризовать круг с центром  $(x_0, y_0)$  и радиусом  $R$ ?
- Растеризуем ограничивающий прямоугольник, проверяя пиксели на вхождение в круг



## Растеризация: круг

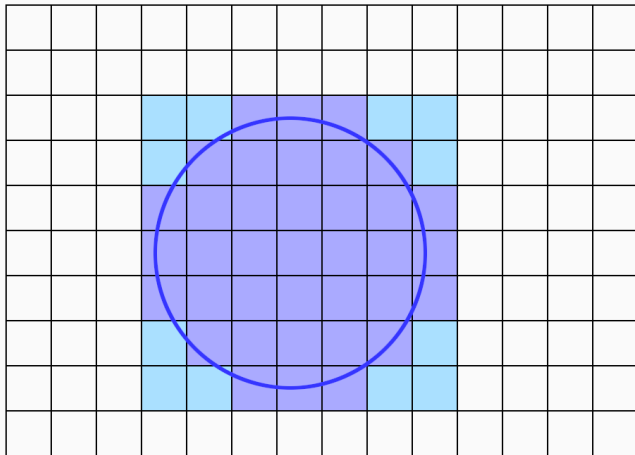
- Как растеризовать круг с центром  $(x_0, y_0)$  и радиусом  $R$ ?
- Растеризуем ограничивающий прямоугольник, проверяя пиксели на вхождение в круг

```
int xmin = round(x_0 - R);
int xmax = round(x_0 + R);

int ymin = round(y_0 - R);
int ymax = round(y_0 + R);

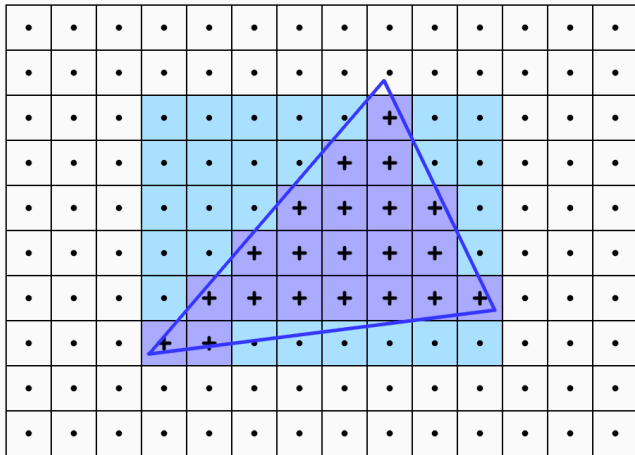
for (int x = xmin; x <= xmax; ++x) {
    for (int y = ymin; y <= ymax; ++y) {
        if (sqr(x - x_0) + sqr(y - y_0) <= sqr(R))
            set_pixel(x, y, color);
    }
}
```

## Растеризация: круг



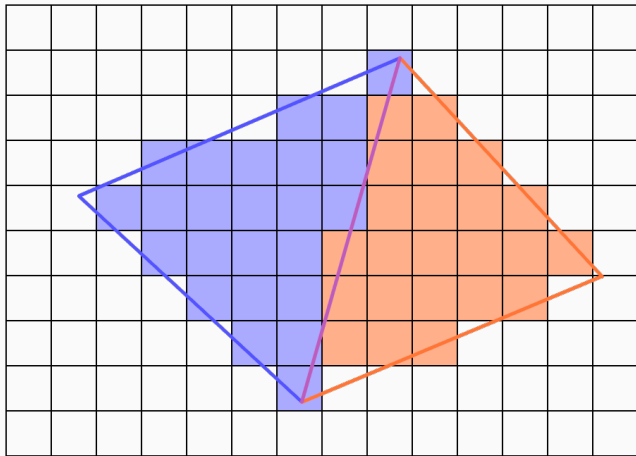
- Пиксель растеризуется, если *центр пикселя* содержится в треугольнике

# Растеризация в OpenGL



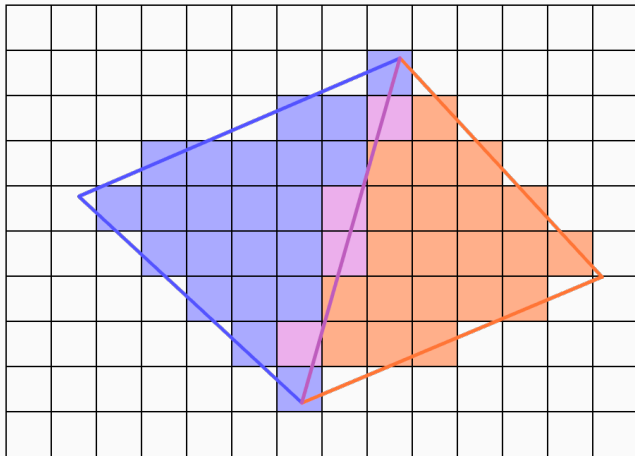
- Пиксель растеризуется, если *центр пикселя* содержится в треугольнике

- Пиксель растеризуется, если *центр пикселя* содержится в треугольнике
- Если у двух треугольников есть общее ребро (и они не пересекаются внутренностями), то
  - Каждый пиксель будет принадлежать ровно одному треугольнику, т.е. не будет наложения
  - Ни один пиксель общего ребра не будет пропущен, т.е. не будет “дырок”



# Растеризация в OpenGL

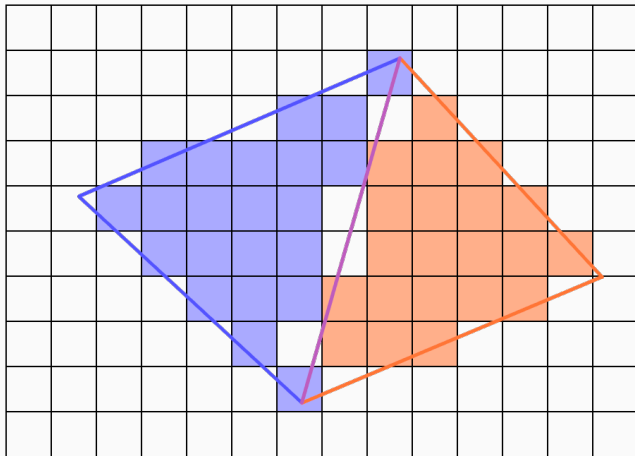
Не будет наложения пикселей:





# Растеризация в OpenGL

Не будет “дырок”:



- Пиксель растеризуется, если *центр пикселя* содержится в треугольнике
- Если у двух треугольников есть общее ребро (и они не пересекаются внутренностями), то
  - Каждый пиксель будет принадлежать ровно одному треугольнику, т.е. не будет наложения
  - Ни один пиксель общего ребра не будет пропущен, т.е. не будет “дырок”

- Пиксель растеризуется, если *центр пикселя* содержится в треугольнике
- Если у двух треугольников есть общее ребро (и они не пересекаются внутренностями), то
  - Каждый пиксель будет принадлежать ровно одному треугольнику, т.е. не будет наложения
  - Ни один пиксель общего ребра не будет пропущен, т.е. не будет “дырок”
- Подробнее: [en.wikibooks.org/wiki/GLSL\\_Programming/Rasterization](https://en.wikibooks.org/wiki/GLSL_Programming/Rasterization)

- В современном OpenGL есть только три основных примитива для растеризации:

- В современном OpenGL есть только три основных примитива для растеризации:
  - **Точки:** `GL_POINTS`

- В современном OpenGL есть только три основных примитива для растеризации:
  - **Точки:** `GL_POINTS`
  - **Линии:** `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`

- В современном OpenGL есть только три основных примитива для растеризации:
  - Точки: `GL_POINTS`
  - Линии: `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`
  - Треугольники: `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`

- В современном OpenGL есть только три основных примитива для растеризации:
  - **Точки:** `GL_POINTS`
  - **Линии:** `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`
  - **Треугольники:** `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`
- Есть особые примитивы для геометрических шейдеров:  
`GL_LINE_STRIP_ADJACENCY`, `GL_LINES_ADJACENCY`,  
`GL_TRIANGLE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY`



# Растеризация в OpenGL

- В современном OpenGL есть только три основных примитива для растеризации:
  - **Точки:** `GL_POINTS`
  - **Линии:** `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`
  - **Треугольники:** `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`
- Есть особые примитивы для геометрических шейдеров:  
`GL_LINE_STRIP_ADJACENCY`, `GL_LINES_ADJACENCY`,  
`GL_TRIANGLE_STRIP_ADJACENCY`, `GL_TRIANGLES_ADJACENCY`
- Точки и линии обычно используются для дебажного рендеринга или вместе с геометрическими шейдерами, *главный примитив – треугольник*

- Почему основным примитивом рисования стал *треугольник*?

- Почему основным примитивом рисования стал *треугольник*?
- Для более сложных геометрических фигур (круг, многоугольник, и т.д.):

- Почему основным примитивом рисования стал *треугольник*?
- Для более сложных геометрических фигур (круг, многоугольник, и т.д.):
  - - Сложнее алгоритм разтеризации (особенно с учётом перспективной проекции)

- Почему основным примитивом рисования стал *треугольник*?
- Для более сложных геометрических фигур (круг, многоугольник, и т.д.):
  - - Сложнее алгоритм разтеризации (особенно с учётом перспективной проекции)
  - - Сложнее задавать и интерполировать атрибуты (накладывать цвет и текстуру, вычислять освещение)

# О треугольниках

Плюсы треугольника:

Плюсы треугольника:

- + Образ под действием перспективной проекции – тоже треугольник

Плюсы треугольника:

- + Образ под действием перспективной проекции – тоже треугольник
- + Есть единственный разумный способ интерполяции (линейная, с барицентрическими координатами)



Плюсы треугольника:

- + Образ под действием перспективной проекции – тоже треугольник
- + Есть единственный разумный способ интерполяции (линейная, с барицентрическими координатами)
- + Позволяет рисовать спрайты (прямоугольник – два треугольника)

Плюсы треугольника:

- + Образ под действием перспективной проекции – тоже треугольник
- + Есть единственный разумный способ интерполяции (линейная, с барицентрическими координатами)
- + Позволяет рисовать спрайты (прямоугольник – два треугольника)
- + Позволяет рисовать многоугольники (посредством триангуляции)

# О треугольниках

Плюсы треугольника:

- + Образ под действием перспективной проекции – тоже треугольник
- + Есть единственный разумный способ интерполяции (линейная, с барицентрическими координатами)
- + Позволяет рисовать спрайты (прямоугольник – два треугольника)
- + Позволяет рисовать многоугольники (посредством триангуляции)
- + Позволяет рисовать линии (превращая их в тонкие многоугольники)

# О треугольниках

Плюсы треугольника:

- + Образ под действием перспективной проекции – тоже треугольник
- + Есть единственный разумный способ интерполяции (линейная, с барицентрическими координатами)
- + Позволяет рисовать спрайты (прямоугольник – два треугольника)
- + Позволяет рисовать многоугольники (посредством триангуляции)
- + Позволяет рисовать линии (превращая их в тонкие многоугольники)
- + Позволяет рисовать более сложные фигуры (аппроксимируя)

- Последовательность операций на GPU, превращающих входные данные в картинку на экране

# Графический конвейер (graphics pipeline)

- Последовательность операций на GPU, превращающих входные данные в картинку на экране
- Что происходит от вызова `glDrawArrays` до появления картинки на экране

# Графический конвейер (graphics pipeline)

- Последовательность операций на GPU, превращающих входные данные в картинку на экране
- Что происходит от вызова `glDrawArrays` до появления картинки на экране
- Почти не отличается для разных графических API

# Графический конвейер (graphics pipeline)

Основные части конвейера:

- Входной поток вершин (vertex stream)



# Графический конвейер (graphics pipeline)

Основные части конвейера:

- Входной поток вершин (vertex stream)
  - Позже мы узнаем, как его задавать

# Графический конвейер (graphics pipeline)

Основные части конвейера:

- Входной поток вершин (vertex stream)
  - Позже мы узнаем, как его задавать
- Вершинный шейдер: обрабатывает вершины по одной

# Графический конвейер (graphics pipeline)

Основные части конвейера:

- Входной поток вершин (vertex stream)
  - Позже мы узнаем, как его задавать
- Вершинный шейдер: обрабатывает вершины по одной
- Сборка примитивов (primitive assembly) из отдельных вершин

# Графический конвейер (graphics pipeline)

Основные части конвейера:

- Входной поток вершин (vertex stream)
  - Позже мы узнаем, как его задавать
- Вершинный шейдер: обрабатывает вершины по одной
- Сборка примитивов (primitive assembly) из отдельных вершин
- Преобразование в оконную систему координат (viewport transform)

# Графический конвейер (graphics pipeline)

Основные части конвейера:

- Входной поток вершин (vertex stream)
  - Позже мы узнаем, как его задавать
- Вершинный шейдер: обрабатывает вершины по одной
- Сборка примитивов (primitive assembly) из отдельных вершин
- Преобразование в оконную систему координат (viewport transform)
- Отсечение невидимых граней (back-face culling)

# Графический конвейер (graphics pipeline)

Основные части конвейера:

- Входной поток вершин (vertex stream)
  - Позже мы узнаем, как его задавать
- Вершинный шейдер: обрабатывает вершины по одной
- Сборка примитивов (primitive assembly) из отдельных вершин
- Преобразование в оконную систему координат (viewport transform)
- Отсечение невидимых граней (back-face culling)
- Растеризация примитивов

# Графический конвейер (graphics pipeline)

Основные части конвейера:

- Входной поток вершин (vertex stream)
  - Позже мы узнаем, как его задавать
- Вершинный шейдер: обрабатывает вершины по одной
- Сборка примитивов (primitive assembly) из отдельных вершин
- Преобразование в оконную систему координат (viewport transform)
- Отсечение невидимых граней (back-face culling)
- Растеризация примитивов
- Пиксельный (фрагментный) шейдер: обрабатывает пиксели по одному

- Мы пропустили много важных частей конвейера
- Будем их по чуть-чуть добавлять в течение курса
- [khronos.org/opengl/wiki/Rendering\\_Pipeline\\_Overview](https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview)





- SIMD – Single Instruction, Multiple Data

## Два слова о GPU

- SIMD – Single Instruction, Multiple Data
- GPU – как SIMD-расширения процессоров (SSE, AVX), но на более крупном масштабе: сотни/тысячи одновременных вычислений

## Два слова о GPU

- SIMD – Single Instruction, Multiple Data
- GPU – как SIMD-расширения процессоров (SSE, AVX), но на более крупном масштабе: сотни/тысячи одновременных вычислений
- Для производительности вычислительные ядра организованы в группы (wavefronts) по 32/64 элемента

## Два слова о GPU

- SIMD – Single Instruction, Multiple Data
- GPU – как SIMD-расширения процессоров (SSE, AVX), но на более крупном масштабе: сотни/тысячи одновременных вычислений
- Для производительности вычислительные ядра организованы в группы (wavefronts) по 32/64 элемента
- Все ядра одного wavefront'а выполняют одну и ту же операцию (но над разными данными)

## Два слова о GPU

- SIMD – Single Instruction, Multiple Data
- GPU – как SIMD-расширения процессоров (SSE, AVX), но на более крупном масштабе: сотни/тысячи одновременных вычислений
- Для производительности вычислительные ядра организованы в группы (wavefronts) по 32/64 элемента
- Все ядра одного wavefront'a выполняют одну и ту же операцию (но над разными данными)
- Если мы рисуем только один треугольник, под него всё равно выделяется целый wavefront и большинство его ядер делают бесполезную работу

## Два слова о GPU

- SIMD – Single Instruction, Multiple Data
- GPU – как SIMD-расширения процессоров (SSE, AVX), но на более крупном масштабе: сотни/тысячи одновременных вычислений
- Для производительности вычислительные ядра организованы в группы (wavefronts) по 32/64 элемента
- Все ядра одного wavefront'а выполняют одну и ту же операцию (но над разными данными)
- Если мы рисуем только один треугольник, под него всё равно выделяется целый wavefront и большинство его ядер делают бесполезную работу
- Максимизация загрузки wavefront'ов – важная часть низкоуровневой оптимизации GPU

- Конвейер – удобная абстракция, внутри GPU всё устроено сложнее



# Графический конвейер (graphics pipeline)

- Конвейер – удобная абстракция, внутри GPU всё устроено сложнее
- Каждая часть конвейера выполняется сразу на большом объёме входных данных параллельно

# Графический конвейер (graphics pipeline)

- Конвейер – удобная абстракция, внутри GPU всё устроено сложнее
- Каждая часть конвейера выполняется сразу на большом объёме входных данных параллельно
- Разные части конвейера могут выполняться одновременно (на разных данных)

# Графический конвейер (graphics pipeline)

- Конвейер – удобная абстракция, внутри GPU всё устроено сложнее
- Каждая часть конвейера выполняется сразу на большом объёме входных данных параллельно
- Разные части конвейера могут выполняться одновременно (на разных данных)
- Разные вызовы команд рисования (`glDrawArrays` и др.) могут обрабатываться параллельно



- Входные данные:

# Вершинный (vertex) шейдер

- Входные данные:
  - Атрибуты вершин (мы позже узнаем, как их задавать) – свои для каждой вершины

# Вершинный (vertex) шейдер

- Входные данные:
  - Атрибуты вершин (мы позже узнаем, как их задавать) – свои для каждой вершины
  - Uniform-переменные – глобальные значения, не меняющиеся в течение одного вызова команды рисования (напр. `glDrawArrays`): `uniform float scale;`

# Вершинный (vertex) шейдер

- Входные данные:
  - Атрибуты вершин (мы позже узнаем, как их задавать) – свои для каждой вершины
  - Uniform-переменные – глобальные значения, не меняющиеся в течение одного вызова команды рисования (напр. `glDrawArrays`): `uniform float scale;`
- Выходные данные:
  - `vec4 gl_Position;`



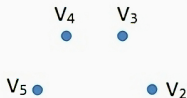
# Вершинный (vertex) шейдер

- Входные данные:
  - Атрибуты вершин (мы позже узнаем, как их задавать) – свои для каждой вершины
  - Uniform-переменные – глобальные значения, не меняющиеся в течение одного вызова команды рисования (напр. `glDrawArrays`): `uniform float scale;`
- Выходные данные:
  - `vec4 gl_Position;`
  - Переменные, интерполированное значение которых попадёт во фрагментный (пиксельный) шейдер: `out vec3 color;`

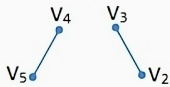
## Сборка примитивов (primitive assembly)

- Превращает набор независимых вершин в *примитивы* растеризации: точки/линии/треугольники
- Правила сборки зависят от выбранного режима: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`

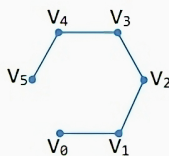
# Сборка примитивов (primitive assembly)



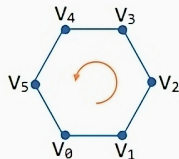
GL\_POINTS



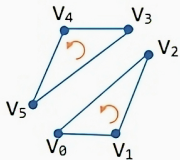
GL\_LINES



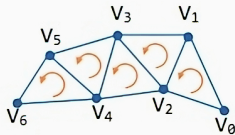
GL\_LINE\_STRIP



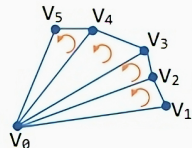
GL\_LINE\_LOOP



GL\_TRIANGLES



GL\_TRIANGLE\_STRIP



GL\_TRIANGLE\_FAN

## Сборка примитивов (primitive assembly)

- Превращает набор независимых вершин в *примитивы* растеризации: точки/линии/треугольники
- Правила сборки зависят от выбранного режима: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`

## Сборка примитивов (primitive assembly)

- Превращает набор независимых вершин в *примитивы* растеризации: точки/линии/треугольники
- Правила сборки зависят от выбранного режима: `GL_POINTS`, `GL_LINES`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLES`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN`
- Порядок вершин в `GL_TRIANGLE_STRIP` и `GL_TRIANGLE_FAN` важен для *back-face culling*

# Viewport transform

- Преобразование в оконную (пиксельную) систему координат
- $X : [-1, 1] \rightarrow [X_{min}, X_{max}]$
- $Y : [-1, 1] \rightarrow [Y_{max}, Y_{min}]$  (-1 внизу, 1 вверху)

# Viewport transform

- Преобразование в оконную (пиксельную) систему координат
- $X : [-1, 1] \rightarrow [X_{min}, X_{max}]$
- $Y : [-1, 1] \rightarrow [Y_{max}, Y_{min}]$  (-1 внизу, 1 вверху)
- Настроить: `glViewport(xmin, ymin, width, height)` (в пикселях)

# Viewport transform

- Преобразование в оконную (пиксельную) систему координат
- $X : [-1, 1] \rightarrow [X_{min}, X_{max}]$
- $Y : [-1, 1] \rightarrow [Y_{max}, Y_{min}]$  (-1 внизу, 1 вверху)
- Настроить: `glViewport(xmin, ymin, width, height)` (в пикселях)
- Вне прямоугольника  $[X_{min}, X_{max}] \times [Y_{min}, Y_{max}]$  *ничего не нарисует*: GPU обрежет все примитивы по этим границам



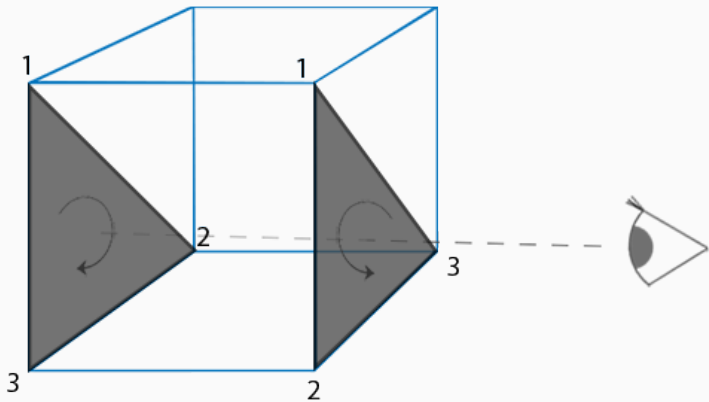
# Back-face culling

- Обычно в OpenGL треугольники, вершины которых оказываются на экране в порядке обхода *по часовой стрелке*, **не рисуются**, чтобы не рисовать треугольники, которые будут скрыты другими треугольниками ближе к камере

# Back-face culling

- Обычно в OpenGL треугольники, вершины которых оказываются на экране в порядке обхода *по часовой стрелке*, **не рисуются**, чтобы не рисовать треугольники, которые будут скрыты другими треугольниками ближе к камере
  - Из-за этого в некоторых играх вы видите объекты насквозь, когда проваливаетесь в них

# Back-face culling



# Back-face culling

- Обычно в OpenGL треугольники, вершины которых оказываются на экране в порядке обхода *по часовой стрелке*, **не рисуются**, чтобы не рисовать треугольники, которые будут скрыты другими треугольниками ближе к камере
  - Из-за этого в некоторых играх вы видите объекты насквозь, когда проваливаетесь в них

# Back-face culling

- Обычно в OpenGL треугольники, вершины которых оказываются на экране в порядке обхода *по часовой стрелке*, **не рисуются**, чтобы не рисовать треугольники, которые будут скрыты другими треугольниками ближе к камере
  - Из-за этого в некоторых играх вы видите объекты насквозь, когда проваливаетесь в них
- Работает только для сплошных 3D моделей без “дырок”, имеющих внутренность (напр. *manifold mesh*)

# Back-face culling

- Обычно в OpenGL треугольники, вершины которых оказываются на экране в порядке обхода *по часовой стрелке*, **не рисуются**, чтобы не рисовать треугольники, которые будут скрыты другими треугольниками ближе к камере
  - Из-за этого в некоторых играх вы видите объекты насквозь, когда проваливаетесь в них
- Работает только для сплошных 3D моделей без “дырок”, имеющих внутренность (напр. *manifold mesh*)
- Работает только при договорённости, что все грани описаны *против часовой стрелки*, если смотреть на модель *снаружи*

# Back-face culling

- Обычно в OpenGL треугольники, вершины которых оказываются на экране в порядке обхода *по часовой стрелке*, **не рисуются**, чтобы не рисовать треугольники, которые будут скрыты другими треугольниками ближе к камере
  - Из-за этого в некоторых играх вы видите объекты насквозь, когда проваливаетесь в них
- Работает только для сплошных 3D моделей без “дырок”, имеющих внутренность (напр. *manifold mesh*)
- Работает только при договорённости, что все грани описаны *против часовой стрелки*, если смотреть на модель *снаружи*
- Порядок вершин в `GL_TRIANGLE_STRIP` и `GL_TRIANGLE_FAN` чётко определён, чтобы не сломать back-face culling

## Back-face culling

- Выключен по умолчанию



# Back-face culling

- Выключен по умолчанию
- Включить/выключить это поведение: `glEnable(GL_CULL_FACE)` или `glDisable(GL_CULL_FACE)`

# Back-face culling

- Выключен по умолчанию
- Включить/выключить это поведение: `glEnable(GL_CULL_FACE)` или `glDisable(GL_CULL_FACE)`
- Настроить, какие треугольники будут скрываться:  
`glCullFace(GL_BACK)`, `glCullFace(GL_FRONT)`,  
`glCullFace(GL_FRONT_AND_BACK)`

# Back-face culling

- Выключен по умолчанию
- Включить/выключить это поведение: `glEnable(GL_CULL_FACE)` или `glDisable(GL_CULL_FACE)`
- Настроить, какие треугольники будут скрываться:  
`glCullFace(GL_BACK)`, `glCullFace(GL_FRONT)`,  
`glCullFace(GL_FRONT_AND_BACK)`
- Настроить, какие треугольники считаются FRONT, а какие – BACK: `glFrontFace(GL_CCW)`, `glFrontFace(GL_CW)`

# Back-face culling

- Выключен по умолчанию
- Включить/выключить это поведение: `glEnable(GL_CULL_FACE)` или `glDisable(GL_CULL_FACE)`
- Настроить, какие треугольники будут скрываться:  
`glCullFace(GL_BACK)`, `glCullFace(GL_FRONT)`,  
`glCullFace(GL_FRONT_AND_BACK)`
- Настроить, какие треугольники считаются FRONT, а какие – BACK: `glFrontFace(GL_CCW)`, `glFrontFace(GL_CW)`
- По умолчанию стоит `glFrontFace(GL_CCW)`, лучше всего его *никогда не менять* – большинство форматов и инструментов следуют этой договорённости

# Фрагментный (пиксельный, fragment) шейдер

# Фрагментный (пиксельный, fragment) шейдер

- Входные данные:

# Фрагментный (пиксельный, fragment) шейдер

- Входные данные:
  - Uniform-переменные

# Фрагментный (пиксельный, fragment) шейдер

- Входные данные:
  - Uniform-переменные
  - Проинтерполированные **out**-переменные вершинного шейдера: **in** **vec3** color;



# Фрагментный (пиксельный, fragment) шейдер

- Входные данные:
  - Uniform-переменные
  - Проинтерполированные **out**-переменные вершинного шейдера: **in** **vec3** color;
  - **gl\_FragCoord** – координаты центра пикселя, напр. (42.5, 239.5)

# Фрагментный (пиксельный, fragment) шейдер

- Входные данные:
  - Uniform-переменные
  - Проинтерполированные **out**-переменные вершинного шейдера: `in vec3 color;`
  - `gl_FragCoord` – координаты центра пикселя, напр. (42.5, 239.5)
  - И много других: [khronos.org/opengl/wiki/Fragment\\_Shader/Defined\\_Inputs](https://www.khronos.org/opengl/wiki/Fragment_Shader/Defined_Inputs)

# Фрагментный (пиксельный, fragment) шейдер

- Входные данные:
  - Uniform-переменные
  - Проинтерполированные **out**-переменные вершинного шейдера: **in vec3 color;**
  - **gl\_FragCoord** – координаты центра пикселя, напр. (42.5, 239.5)
  - И много других: [khronos.org/opengl/wiki/Fragment\\_Shader/Defined\\_Inputs](https://www.khronos.org/opengl/wiki/Fragment_Shader/Defined_Inputs)
- Выходные данные:
  - **layout (location = 0) out vec4 out\_color;** – выходной цвет в формате RGBA

# Фрагментный (пиксельный, fragment) шейдер

- Входные данные:
  - Uniform-переменные
  - Проинтерполированные **out**-переменные вершинного шейдера: **in vec3 color**;
  - **gl\_FragCoord** – координаты центра пикселя, напр. (42.5, 239.5)
  - И много других: [khronos.org/opengl/wiki/Fragment\\_Shader/Defined\\_Inputs](https://www.khronos.org/opengl/wiki/Fragment_Shader/Defined_Inputs)
- Выходные данные:
  - **layout (location = 0) out vec4 out\_color**; – выходной цвет в формате RGBA
  - Может быть несколько, об этом поговорим позже

- OpenGL, Vulkan: GLSL (GL Shading Language)

- OpenGL, Vulkan: GLSL (GL Shading Language)
- DirectX: HLSL (High-Level Shading Language)

- OpenGL, Vulkan: GLSL (GL Shading Language)
- DirectX: HLSL (High-Level Shading Language)
- DirectX (до 2012): Cg (C for Graphics), deprecated

- OpenGL, Vulkan: GLSL (GL Shading Language)
- DirectX: HLSL (High-Level Shading Language)
- DirectX (до 2012): Cg (C for Graphics), deprecated
- WebGPU: WGSL (WebGpu Shading Language)



- OpenGL, Vulkan: [GLSL](#) (GL Shading Language)
- DirectX: HLSL (High-Level Shading Language)
- DirectX (до 2012): Cg (C for Graphics), deprecated
- WebGPU: WGSL (WebGpu Shading Language)
- И другие: [en.wikipedia.org/wiki/Shading\\_language](https://en.wikipedia.org/wiki/Shading_language)

- Версионировается параллельно OpenGL

- Версионирование параллельно OpenGL
- До OpenGL 3.3:
  - OpenGL 2.0 → GLSL 1.10
  - OpenGL 2.1 → GLSL 1.20
  - OpenGL 3.0 → GLSL 1.30
  - OpenGL 3.1 → GLSL 1.40
  - OpenGL 3.2 → GLSL 1.50

- Версионирование параллельно OpenGL
- До OpenGL 3.3:
  - OpenGL 2.0 → GLSL 1.10
  - OpenGL 2.1 → GLSL 1.20
  - OpenGL 3.0 → GLSL 1.30
  - OpenGL 3.1 → GLSL 1.40
  - OpenGL 3.2 → GLSL 1.50
- Начиная с OpenGL 3.3 версии GLSL и OpenGL совпадают:
  - OpenGL 3.3 → GLSL 3.30
  - OpenGL 4.0 → GLSL 4.00
  - OpenGL 4.1 → GLSL 4.10
  - И т.д.

- OpenGL ES и WebGL используют язык GLSL ES
  - OpenGL ES 2.0 → GLSL ES 1.00
  - OpenGL ES 3.0 → GLSL ES 3.00
  - OpenGL ES 3.1 → GLSL ES 3.10
  - OpenGL ES 3.2 → GLSL ES 3.20
  - WebGL 1.0 → GLSL ES 1.00
  - WebGL 2.0 → GLSL ES 3.00

- Программа должна начинаться с  
`#version <версия> [ <профиль> ]`
  - У нас будет `#version 330 core`

- Программа должна начинаться с  
`#version <версия> [ <профиль> ]`
  - У нас будет `#version 330 core`
  - Для GLSL ES `#version <версия> es`

- Программа должна начинаться с `#version <версия> [ <профиль> ]`
  - У нас будет `#version 330 core`
  - Для GLSL ES `#version <версия> es`
- Программа должна содержать функцию `void main()`



- Похож на C

- Похож на C
- Типы данных:

# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`

# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`
  - Векторные: `bvec2`, `bvec3`, `bvec4`, `ivec2`, ..., `uvec2`, ..., `vec2`, ...

# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`
  - Векторные: `bvec2`, `bvec3`, `bvec4`, `ivec2`, ..., `uvec2`, ..., `vec2`, ...
  - Матричные: `mat2`, `mat3`, `mat4`, `mat2x4`, ...

# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`
  - Векторные: `bvec2`, `bvec3`, `bvec4`, `ivec2`, ..., `uvec2`, ..., `vec2`, ...
  - Матричные: `mat2`, `mat3`, `mat4`, `mat2x4`, ...
  - В GLSL 400 или с расширением `GL_ARB_gpu_shader_fp64` есть `double`, `dvec2`, ..., `dmat2`, ...

# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`
  - Векторные: `bvec2`, `bvec3`, `bvec4`, `ivec2`, ..., `uvec2`, ..., `vec2`, ...
  - Матричные: `mat2`, `mat3`, `mat4`, `mat2x4`, ...
  - В GLSL 400 или с расширением `GL_ARB_gpu_shader_fp64` есть `double`, `dvec2`, ..., `dmat2`, ...
- Конструкторы векторов:

# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`
  - Векторные: `bvec2`, `bvec3`, `bvec4`, `ivec2`, ..., `uvec2`, ..., `vec2`, ...
  - Матричные: `mat2`, `mat3`, `mat4`, `mat2x4`, ...
  - В GLSL 400 или с расширением `GL_ARB_gpu_shader_fp64` есть `double`, `dvec2`, ..., `dmat2`, ...
- Конструкторы векторов:
  - `vec3(x, y, z)`



# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`
  - Векторные: `bvec2`, `bvec3`, `bvec4`, `ivec2`, ..., `uvec2`, ..., `vec2`, ...
  - Матричные: `mat2`, `mat3`, `mat4`, `mat2x4`, ...
  - В GLSL 400 или с расширением `GL_ARB_gpu_shader_fp64` есть `double`, `dvec2`, ..., `dmat2`, ...
- Конструкторы векторов:
  - `vec3(x, y, z)`
  - `vec3(x) == vec3(x, x, x)`

# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`
  - Векторные: `bvec2`, `bvec3`, `bvec4`, `ivec2`, ..., `uvec2`, ..., `vec2`, ...
  - Матричные: `mat2`, `mat3`, `mat4`, `mat2x4`, ...
  - В GLSL 400 или с расширением `GL_ARB_gpu_shader_fp64` есть `double`, `dvec2`, ..., `dmat2`, ...
- Конструкторы векторов:
  - `vec3(x, y, z)`
  - `vec3(x) == vec3(x, x, x)`
  - `vec3(vec2(x, y), z) == vec3(x, vec2(y, z))`

# Язык описания шейдеров GLSL

- Похож на C
- Типы данных:
  - Скалярные: `bool`, `int`, `uint`, `float`
  - Векторные: `bvec2`, `bvec3`, `bvec4`, `ivec2`, ..., `uvec2`, ..., `vec2`, ...
  - Матричные: `mat2`, `mat3`, `mat4`, `mat2x4`, ...
  - В GLSL 400 или с расширением `GL_ARB_gpu_shader_fp64` есть `double`, `dvec2`, ..., `dmat2`, ...
- Конструкторы векторов:
  - `vec3(x, y, z)`
  - `vec3(x) == vec3(x, x, x)`
  - `vec3(vec2(x, y), z) == vec3(x, vec2(y, z))`
- Подробнее:  
[khronos.org/opengl/wiki/Data\\_Type\\_\(GLSL\)](http://khronos.org/opengl/wiki/Data_Type_(GLSL))

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $==$ , ...

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $==$ , ...
  - Для векторов и матриц применяются покомпонентно

# Язык описания шейдеров GLSL

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $=$ , ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры

# Язык описания шейдеров GLSL

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $==$ , ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры
  - Исключение: умножение матрицы на вектор это умножение в смысле алгебры

# Язык описания шейдеров GLSL

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $==$ , ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры
  - Исключение: умножение матрицы на вектор это умножение в смысле алгебры
- Доступ к координатам векторов:  $a.x = b.y$



# Язык описания шейдеров GLSL

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $==$ , ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры
  - Исключение: умножение матрицы на вектор это умножение в смысле алгебры
- Доступ к координатам векторов:  $a.x = b.y$
- Swizzling:  $a.xxzy == \text{vec4}(a.x, a.x, a.z, a.y)$

# Язык описания шейдеров GLSL

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $==$ , ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры
  - Исключение: умножение матрицы на вектор это умножение в смысле алгебры
- Доступ к координатам векторов:  $a.x = b.y$
- Swizzling:  $a.xxzy == \text{vec4}(a.x, a.x, a.z, a.y)$
- Доступ к элементам матриц:  $m[\text{column}][\text{row}]$

# Язык описания шейдеров GLSL

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $=$ , ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры
  - Исключение: умножение матрицы на вектор это умножение в смысле алгебры
- Доступ к координатам векторов:  $a.x = b.y$
- Swizzling:  $a.xxzy == \text{vec4}(a.x, a.x, a.z, a.y)$
- Доступ к элементам матриц:  $m[\text{column}][\text{row}]$
- Есть полезные математические функции:  $\text{pow}$ ,  $\text{sin}$ ,  $\text{cos}$ , ...

# Язык описания шейдеров GLSL

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $=$ , ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры
  - Исключение: умножение матрицы на вектор это умножение в смысле алгебры
- Доступ к координатам векторов:  $a.x = b.y$
- Swizzling:  $a.xxzy == \text{vec4}(a.x, a.x, a.z, a.y)$
- Доступ к элементам матриц:  $m[\text{column}][\text{row}]$
- Есть полезные математические функции:  $\text{pow}$ ,  $\text{sin}$ ,  $\text{cos}$ , ...
  - Для векторов и матриц применяются покомпонентно

# Язык описания шейдеров GLSL

- Есть стандартные операции:  $+$ ,  $-$ ,  $*$ ,  $/$ ,  $\%$ ,  $<$ ,  $=$ , ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры
  - Исключение: умножение матрицы на вектор это умножение в смысле алгебры
- Доступ к координатам векторов:  $a.x = b.y$
- Swizzling:  $a.xxzy == \text{vec4}(a.x, a.x, a.z, a.y)$
- Доступ к элементам матриц:  $m[\text{column}][\text{row}]$
- Есть полезные математические функции:  $\text{pow}$ ,  $\text{sin}$ ,  $\text{cos}$ , ...
  - Для векторов и матриц применяются покомпонентно
- Есть математические функции векторной алгебры:  $\text{dot}$ ,  $\text{cross}$ ,  $\text{length}$ ,  $\text{normalize}$ , ...

# Язык описания шейдеров GLSL

- Есть стандартные операции: `+`, `-`, `*`, `/`, `%`, `<`, `==`, ...
  - Для векторов и матриц применяются покомпонентно
  - Исключение: умножение двух матриц это умножение в смысле алгебры
  - Исключение: умножение матрицы на вектор это умножение в смысле алгебры
- Доступ к координатам векторов: `a.x = b.y`
- Swizzling: `a.xxzy == vec4(a.x, a.x, a.z, a.y)`
- Доступ к элементам матриц: `m[column][row]`
- Есть полезные математические функции: `pow`, `sin`, `cos`, ...
  - Для векторов и матриц применяются покомпонентно
- Есть математические функции векторной алгебры: `dot`, `cross`, `length`, `normalize`, ...
- Линейная и нелинейная интерполяция: `mix`, `smoothstep`

- Есть массивы: `float array[5];`

- Есть массивы: `float array[5];`
  - Инициализация:  
`float array[5] = float[5](0.0, 1.0, 2.0, 3.0, 4.0);`



- Есть массивы: `float array[5];`
  - Инициализация:  
`float array[5] = float[5](0.0, 1.0, 2.0, 3.0, 4.0);`
- Константы (известные на момент компиляции):  
`const float PI = 3.141592;`

- Ветвление: `if (condition) { ... } else { ... }`

- Ветвление: `if (condition) { ... } else { ... }`
- Тернарный if: `condition ? x : y`

# Язык описания шейдеров GLSL

- Ветвление: `if (condition) { ... } else { ... }`
- Тернарный if: `condition ? x : y`
- Циклы: `for (int i = 0; i < 10; ++i) { ... }`
  - Бесконечные циклы могут проявиться по-разному, вплоть до краша всей системы :)

# Язык описания шейдеров GLSL

- Ветвление: `if (condition) { ... } else { ... }`
- Тернарный if: `condition ? x : y`
- Циклы: `for (int i = 0; i < 10; ++i) { ... }`
  - Бесконечные циклы могут проявиться по-разному, вплоть до краша всей системы :)
- Функции:

```
vec3 reflect(vec3 v, vec3 n) {  
    return v - 2.0 * n * dot(v, n);  
}
```

# Язык описания шейдеров GLSL

- Ветвление: `if (condition) { ... } else { ... }`
- Тернарный if: `condition ? x : y`
- Циклы: `for (int i = 0; i < 10; ++i) { ... }`
  - Бесконечные циклы могут проявиться по-разному, вплоть до краша всей системы :)
- Функции:

```
vec3 reflect(vec3 v, vec3 n) {  
    return v - 2.0 * n * dot(v, n);  
}
```

- Могут вызывать другие функции

# Язык описания шейдеров GLSL

- Ветвление: `if (condition) { ... } else { ... }`
- Тернарный if: `condition ? x : y`
- Циклы: `for (int i = 0; i < 10; ++i) { ... }`
  - Бесконечные циклы могут проявиться по-разному, вплоть до краша всей системы :)
- Функции:

```
vec3 reflect(vec3 v, vec3 n) {  
    return v - 2.0 * n * dot(v, n);  
}
```

- Могут вызывать другие функции
- Рекурсия **запрещена**

- GLSL ES строже, чем GLSL
  - Нет неявных преобразований типов
  - Нет бесконечных циклов (цикл должен быть ограничен константным значением)



- GLSL ES строже, чем GLSL
  - Нет неявных преобразований типов
  - Нет бесконечных циклов (цикл должен быть ограничен константным значением)
- Драйверы часто разрешают писать нестандартный GLSL-код, или код, не соответствующий выбранной версии GLSL
  - Например, не разрешают неявные преобразования или сложные циклы

- GLSL ES строже, чем GLSL
  - Нет неявных преобразований типов
  - Нет бесконечных циклов (цикл должен быть ограничен константным значением)
- Драйверы часто разрешают писать нестандартный GLSL-код, или код, не соответствующий выбранной версии GLSL
  - Например, не разрешают неявные преобразования или сложные циклы
- Как и в обычных компиляторах, в компиляторах шейдеров могут быть баги :)

- Тьюториал: [learnopengl.com/Getting-started/Shader](https://learnopengl.com/Getting-started/Shader)
- Тьюториал: [lighthouse3d.com/tutorials/glsl-tutorial](https://lighthouse3d.com/tutorials/glsl-tutorial)

- Тьюториал: [learnopengl.com/Getting-started/Shader](https://learnopengl.com/Getting-started/Shader)
- Тьюториал: [lighthouse3d.com/tutorials/glsl-tutorial](https://lighthouse3d.com/tutorials/glsl-tutorial)
- Книжка-учебник с большим количеством примеров сложных шейдеров: **The Book of Shaders**

- Тьюториал: [learnopengl.com/Getting-started/Shader](https://learnopengl.com/Getting-started/Shader)s
- Тьюториал: [lighthouse3d.com/tutorials/glsl-tutorial](https://lighthouse3d.com/tutorials/glsl-tutorial)
- Книжка-учебник с большим количеством примеров сложных шейдеров: **The Book of Shaders**
- Онлайн-редактор шейдеров: [shadertoy.com](https://shadertoy.com)

# Четырёхмерные векторы?

- GLSL поддерживает операции с матрицами и векторами размерностей вплоть до 4
- `gl_Position` имеет 4 компоненты

# Четырёхмерные векторы?

- GLSL поддерживает операции с матрицами и векторами размерностей вплоть до 4
- `gl_Position` имеет 4 компоненты
- Мы хотим рисовать двухмерные и трёхмерные объекты, зачем четвёртая координата?

# Четырёхмерные векторы?

- GLSL поддерживает операции с матрицами и векторами размерностей вплоть до 4
- `gl_Position` имеет 4 компоненты
- Мы хотим рисовать двухмерные и трёхмерные объекты, зачем четвёртая координата?
- Три причины:



# Четырёхмерные векторы?

- GLSL поддерживает операции с матрицами и векторами размерностей вплоть до 4
- `gl_Position` имеет 4 компоненты
- Мы хотим рисовать двухмерные и трёхмерные объекты, зачем четвёртая координата?
- Три причины:
  - Для некоторых *типов данных*: кватернионы, цвета (rgba)

# Четырёхмерные векторы?

- GLSL поддерживает операции с матрицами и векторами размерностей вплоть до 4
- `gl_Position` имеет 4 компоненты
- Мы хотим рисовать двухмерные и трёхмерные объекты, зачем четвёртая координата?
- Три причины:
  - Для некоторых *типов данных*: кватернионы, цвета (rgba)
  - Для *перспективной проекции* (4ая лекция)

# Четырёхмерные векторы?

- GLSL поддерживает операции с матрицами и векторами размерностей вплоть до 4
- `gl_Position` имеет 4 компоненты
- Мы хотим рисовать двухмерные и трёхмерные объекты, зачем четвёртая координата?
- Три причины:
  - Для некоторых *типов данных*: кватернионы, цвета (rgba)
  - Для *перспективной проекции* (4ая лекция)
  - Для *аффинных преобразований*

- Векторные (линейные) пространства – круто: большая теория, эффективные операции

- Векторные (линейные) пространства – круто: большая теория, эффективные операции
- Не очень подходит для того, чтобы моделировать пространство

- Векторные (линейные) пространства – круто: большая теория, эффективные операции
- Не очень подходит для того, чтобы моделировать пространство
  - Где в пространстве точка, соответствующая нулевому вектору?

- Векторные (линейные) пространства – круто: большая теория, эффективные операции
- Не очень подходит для того, чтобы моделировать пространство
  - Где в пространстве точка, соответствующая нулевому вектору?
  - Что такое сумма двух точек?

- Векторные (линейные) пространства – круто: большая теория, эффективные операции
- Не очень подходит для того, чтобы моделировать пространство
  - Где в пространстве точка, соответствующая нулевому вектору?
  - Что такое сумма двух точек?
- Мысль: точки  $\neq$  векторы



- Аффинное пространство: векторное пространство, в котором “забыли” нулевой вектор

# Аффинное пространство

- Аффинное пространство: векторное пространство, в котором “забыли” нулевой вектор
- Формальнее, аффинное пространство  $\mathbb{A}$  над векторным пространством  $\mathbb{V}$  – это:

# Аффинное пространство

- Аффинное пространство: векторное пространство, в котором “забыли” нулевой вектор
- Формальнее, аффинное пространство  $\mathbb{A}$  над векторным пространством  $\mathbb{V}$  – это:
  - Множество точек  $\mathbb{A}$

# Аффинное пространство

- Аффинное пространство: векторное пространство, в котором “забыли” нулевой вектор
- Формальнее, аффинное пространство  $\mathbb{A}$  над векторным пространством  $\mathbb{V}$  – это:
  - Множество точек  $\mathbb{A}$
  - Операция “точка + вектор”:  $\oplus : \mathbb{A} \times \mathbb{V} \rightarrow \mathbb{A}$

# Аффинное пространство

- Аффинное пространство: векторное пространство, в котором “забыли” нулевой вектор
- Формальнее, аффинное пространство  $\mathbb{A}$  над векторным пространством  $\mathbb{V}$  – это:
  - Множество точек  $\mathbb{A}$
  - Операция “точка + вектор”:  $\oplus : \mathbb{A} \times \mathbb{V} \rightarrow \mathbb{A}$
  - Операция “точка - точка”:  $\ominus : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{V}$

# Аффинное пространство

- Аффинное пространство: векторное пространство, в котором “забыли” нулевой вектор
- Формальнее, аффинное пространство  $\mathbb{A}$  над векторным пространством  $\mathbb{V}$  – это:
  - Множество точек  $\mathbb{A}$
  - Операция “точка + вектор”:  $\oplus : \mathbb{A} \times \mathbb{V} \rightarrow \mathbb{A}$
  - Операция “точка - точка”:  $\ominus : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{V}$
  - Аксиомы, связывающие их с операциями векторного пространства, например  $(p \oplus v_1) \oplus v_2 = p \oplus (v_1 + v_2)$

# Аффинное пространство

- Аффинное пространство: векторное пространство, в котором “забыли” нулевой вектор
- Формальнее, аффинное пространство  $\mathbb{A}$  над векторным пространством  $\mathbb{V}$  – это:
  - Множество точек  $\mathbb{A}$
  - Операция “точка + вектор”:  $\oplus : \mathbb{A} \times \mathbb{V} \rightarrow \mathbb{A}$
  - Операция “точка - точка”:  $\ominus : \mathbb{A} \times \mathbb{A} \rightarrow \mathbb{V}$
  - Аксиомы, связывающие их с операциями векторного пространства, например  $(p \oplus v_1) \oplus v_2 = p \oplus (v_1 + v_2)$
- Любую точку  $p_0 \in \mathbb{A}$  можно принять за начало координат
- Это даёт биекцию  $\mathbb{A} \rightarrow \mathbb{V}$  по формуле  $p \mapsto p - p_0$

- Множество решений уравнения  $Lx = 0$  в векторном пространстве – *векторное подпространство*  $\ker L$



- Множество решений уравнения  $Lx = 0$  в векторном пространстве – *векторное подпространство*  $\ker L$
- Множество решений уравнения  $Lx = y$  в векторном пространстве – **не** векторное подпространство

- Множество решений уравнения  $Lx = 0$  в векторном пространстве – *векторное подпространство*  $\ker L$
- Множество решений уравнения  $Lx = y$  в векторном пространстве – **не** векторное подпространство
- Множество решений уравнения  $Lx = y$  в векторном пространстве – *аффинное пространство* над  $\ker L$

# Аффинные комбинации

- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$

# Аффинные комбинации

- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$
- Выберем начало координат  $p_0$  и определим
$$\sum \lambda_i p_i = p_0 + \sum \lambda_i (p_i - p_0) \in \mathbb{A}$$

# Аффинные комбинации

- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$
- Выберем начало координат  $p_0$  и определим
$$\sum \lambda_i p_i = p_0 + \sum \lambda_i (p_i - p_0) \in \mathbb{A}$$
- Теорема: если  $\sum \lambda_i = 1$ , то результат (точка) не зависит от выбора  $p_0$

# Аффинные комбинации

- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$
- Выберем начало координат  $p_0$  и определим
$$\sum \lambda_i p_i = p_0 + \sum \lambda_i (p_i - p_0) \in \mathbb{A}$$
- Теорема: если  $\sum \lambda_i = 1$ , то результат (точка) не зависит от выбора  $p_0$
- В этом случае  $q = \sum \lambda_i p_i$  называется аффинной комбинацией точек  $p_i$  с коэффициентами  $\lambda_i$

# Аффинные комбинации

- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$
- Выберем начало координат  $p_0$  и определим
$$\sum \lambda_i p_i = p_0 + \sum \lambda_i (p_i - p_0) \in \mathbb{A}$$
- Теорема: если  $\sum \lambda_i = 1$ , то результат (точка) не зависит от выбора  $p_0$
- В этом случае  $q = \sum \lambda_i p_i$  называется аффинной комбинацией точек  $p_i$  с коэффициентами  $\lambda_i$
- Коэффициенты  $\lambda_i$  называются барицентрическими координатами точки  $q$

# Аффинные комбинации

- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$
- Выберем начало координат  $p_0$  и определим 
$$\sum \lambda_i p_i = p_0 + \sum \lambda_i (p_i - p_0) \in \mathbb{A}$$
- Теорема: если  $\sum \lambda_i = 1$ , то результат (точка) не зависит от выбора  $p_0$
- В этом случае  $q = \sum \lambda_i p_i$  называется аффинной комбинацией точек  $p_i$  с коэффициентами  $\lambda_i$
- Коэффициенты  $\lambda_i$  называются барицентрическими координатами точки  $q$
- Поиск  $\lambda_i$  по известным  $q$  и  $p_i$  сводится к решению линейной системы уравнений (квадратная только если  $N = D + 1$ , где  $D$  – размерность пространства)



# Аффинные комбинации

- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$
- Выберем начало координат  $p_0$  и определим 
$$\sum \lambda_i p_i = p_0 + \sum \lambda_i (p_i - p_0) \in \mathbb{A}$$
- Теорема: если  $\sum \lambda_i = 1$ , то результат (точка) не зависит от выбора  $p_0$
- В этом случае  $q = \sum \lambda_i p_i$  называется аффинной комбинацией точек  $p_i$  с коэффициентами  $\lambda_i$
- Коэффициенты  $\lambda_i$  называются барицентрическими координатами точки  $q$
- Поиск  $\lambda_i$  по известным  $q$  и  $p_i$  сводится к решению линейной системы уравнений (квадратная только если  $N = D + 1$ , где  $D$  – размерность пространства)
- Аффинная комбинация пустого множества точек не определена

## Аффинные комбинации: пример



## Аффинные комбинации: пример

$$\lambda_1 = 1$$

$$\lambda_2 = 0$$



$p_1$



$p_2$

## Аффинные комбинации: пример

$$\lambda_1 = 0$$

$$\lambda_2 = 1$$



$p_1$

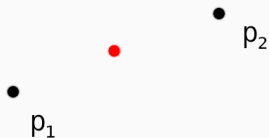


$p_2$

## Аффинные комбинации: пример

$$\lambda_1 = 0.5$$

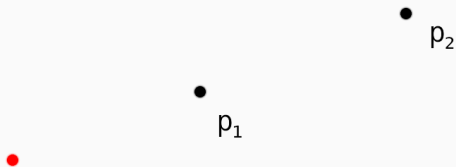
$$\lambda_2 = 0.5$$



## Аффинные комбинации: пример

$$\lambda_1 = 2$$

$$\lambda_2 = -1$$



- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$

- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$
- Выберем начало координат  $p_0$  и определим
$$\sum \lambda_i p_i = \sum \lambda_i (p_i - p_0) \in \mathbb{V}$$



- Пусть есть набор точек  $p_1, \dots, p_N$  и набор коэффициентов  $\lambda_1, \dots, \lambda_N$
- Выберем начало координат  $p_0$  и определим
$$\sum \lambda_i p_i = \sum \lambda_i (p_i - p_0) \in \mathbb{V}$$
- Теорема: если  $\sum \lambda_i = 0$ , то результат (вектор) не зависит от выбора  $p_0$

- Пусть есть набор точек  $p_1, \dots, p_N$

- Пусть есть набор точек  $p_1, \dots, p_N$
- Их *аффинная оболочка* – множество всех возможных аффинных комбинаций, т.е. множество  $\sum \lambda_i p_i$  для всех возможных  $\lambda_i$  таких, что  $\sum \lambda_i = 1$

- Пусть есть набор точек  $p_1, \dots, p_N$
- Их *аффинная оболочка* – множество всех возможных аффинных комбинаций, т.е. множество  $\sum \lambda_i p_i$  для всех возможных  $\lambda_i$  таких, что  $\sum \lambda_i = 1$
- Аффинная оболочка точки – сама *точка*

# Аффинная оболочка

- Пусть есть набор точек  $p_1, \dots, p_N$
- Их *аффинная оболочка* – множество всех возможных аффинных комбинаций, т.е. множество  $\sum \lambda_i p_i$  для всех возможных  $\lambda_i$  таких, что  $\sum \lambda_i = 1$
- Аффинная оболочка точки – сама *точка*
- Аффинная оболочка двух точек – содержащая их *прямая*

# Аффинная оболочка

- Пусть есть набор точек  $p_1, \dots, p_N$
- Их *аффинная оболочка* – множество всех возможных аффинных комбинаций, т.е. множество  $\sum \lambda_i p_i$  для всех возможных  $\lambda_i$  таких, что  $\sum \lambda_i = 1$
- Аффинная оболочка точки – сама *точка*
- Аффинная оболочка двух точек – содержащая их *прямая*
- Аффинная оболочка трёх точек – содержащая их *плоскость*

# Аффинная оболочка

- Пусть есть набор точек  $p_1, \dots, p_N$
- Их *аффинная оболочка* – множество всех возможных аффинных комбинаций, т.е. множество  $\sum \lambda_i p_i$  для всех возможных  $\lambda_i$  таких, что  $\sum \lambda_i = 1$
- Аффинная оболочка точки – сама *точка*
- Аффинная оболочка двух точек – содержащая их *прямая*
- Аффинная оболочка трёх точек – содержащая их *плоскость*
- И т.д.

- Линейная интерполяция = интерполяция, использующая барицентрические координаты как коэффициенты



# Линейная интерполяция

- Линейная интерполяция = интерполяция, использующая барицентрические координаты как коэффициенты
- Пусть есть точки  $p_i$  и точка  $q = \sum \lambda_i p_i$

# Линейная интерполяция

- Линейная интерполяция = интерполяция, использующая барицентрические координаты как коэффициенты
- Пусть есть точки  $p_i$  и точка  $q = \sum \lambda_i p_i$
- Пусть в точках  $p_i$  задано значение некоторой величины  $f_i$

# Линейная интерполяция

- Линейная интерполяция = интерполяция, использующая барицентрические координаты как коэффициенты
- Пусть есть точки  $p_i$  и точка  $q = \sum \lambda_i p_i$
- Пусть в точках  $p_i$  задано значение некоторой величины  $f_i$
- Линейная интерполяция величины  $f$  в точке  $q$  – значение  $\sum \lambda_i f_i$

# Линейная интерполяция

- Линейная интерполяция = интерполяция, использующая барицентрические координаты как коэффициенты
- Пусть есть точки  $p_i$  и точка  $q = \sum \lambda_i p_i$
- Пусть в точках  $p_i$  задано значение некоторой величины  $f_i$
- Линейная интерполяция величины  $f$  в точке  $q$  – значение  $\sum \lambda_i f_i$
- Для этого нужно вычислять  $\lambda_i$  по известным  $p_i$  и  $q$ , что сводится к системе линейных уравнений

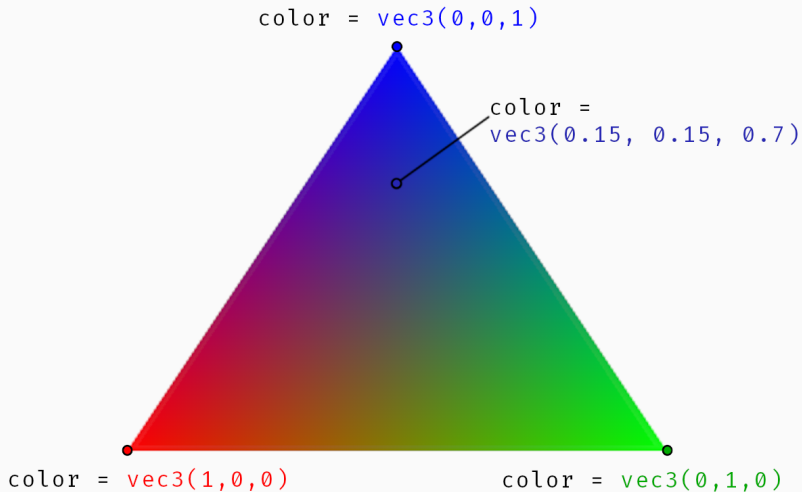
# Линейная интерполяция

- Линейная интерполяция = интерполяция, использующая барицентрические координаты как коэффициенты
- Пусть есть точки  $p_i$  и точка  $q = \sum \lambda_i p_i$
- Пусть в точках  $p_i$  задано значение некоторой величины  $f_i$
- Линейная интерполяция величины  $f$  в точке  $q$  – значение  $\sum \lambda_i f_i$
- Для этого нужно вычислять  $\lambda_i$  по известным  $p_i$  и  $q$ , что сводится к системе линейных уравнений
- **Ровно это** происходит при интерполяции значений, переданных из вершинного шейдера во фрагментный

# Линейная интерполяция

- Линейная интерполяция = интерполяция, использующая барицентрические координаты как коэффициенты
- Пусть есть точки  $p_i$  и точка  $q = \sum \lambda_i p_i$
- Пусть в точках  $p_i$  задано значение некоторой величины  $f_i$
- Линейная интерполяция величины  $f$  в точке  $q$  – значение  $\sum \lambda_i f_i$
- Для этого нужно вычислять  $\lambda_i$  по известным  $p_i$  и  $q$ , что сводится к системе линейных уравнений
- **Ровно это** происходит при интерполяции значений, переданных из вершинного шейдера во фрагментный (с точностью до перспективной проекции, о чём мы поговорим позже)

# Линейная интерполяция цветов



- Выпуклая комбинация: аффинная комбинация, в которой все коэффициенты неотрицательны  $\lambda_i \geq 0$



- Выпуклая комбинация: аффинная комбинация, в которой все коэффициенты неотрицательны  $\lambda_i \geq 0$ 
  - Вместе с  $\sum \lambda_i = 1$  это означает, что  $\lambda_i \in [0, 1]$

# Выпуклые оболочки

- Выпуклая комбинация: аффинная комбинация, в которой все коэффициенты неотрицательны  $\lambda_i \geq 0$ 
  - Вместе с  $\sum \lambda_i = 1$  это означает, что  $\lambda_i \in [0, 1]$
- Множество всех выпуклых комбинаций набора точек = *выпуклая оболочка*

# Выпуклые оболочки

- Выпуклая комбинация: аффинная комбинация, в которой все коэффициенты неотрицательны  $\lambda_i \geq 0$ 
  - Вместе с  $\sum \lambda_i = 1$  это означает, что  $\lambda_i \in [0, 1]$
- Множество всех выпуклых комбинаций набора точек = *выпуклая оболочка*
- Выпуклая оболочка точки – сама *точка*

# Выпуклые оболочки

- Выпуклая комбинация: аффинная комбинация, в которой все коэффициенты неотрицательны  $\lambda_i \geq 0$ 
  - Вместе с  $\sum \lambda_i = 1$  это означает, что  $\lambda_i \in [0, 1]$
- Множество всех выпуклых комбинаций набора точек = *выпуклая оболочка*
- Выпуклая оболочка точки – сама *точка*
- Выпуклая оболочка двух точек – *отрезок* между этими точками

# Выпуклые оболочки

- Выпуклая комбинация: аффинная комбинация, в которой все коэффициенты неотрицательны  $\lambda_i \geq 0$ 
  - Вместе с  $\sum \lambda_i = 1$  это означает, что  $\lambda_i \in [0, 1]$
- Множество всех выпуклых комбинаций набора точек = *выпуклая оболочка*
- Выпуклая оболочка точки – сама *точка*
- Выпуклая оболочка двух точек – *отрезок* между этими точками
- Выпуклая оболочка трёх точек – *треугольник* с этими точками в качестве вершин

# Выпуклые оболочки

- Выпуклая комбинация: аффинная комбинация, в которой все коэффициенты неотрицательны  $\lambda_i \geq 0$ 
  - Вместе с  $\sum \lambda_i = 1$  это означает, что  $\lambda_i \in [0, 1]$
- Множество всех выпуклых комбинаций набора точек = *выпуклая оболочка*
- Выпуклая оболочка точки – сама *точка*
- Выпуклая оболочка двух точек – *отрезок* между этими точками
- Выпуклая оболочка трёх точек – *треугольник* с этими точками в качестве вершин
- Выпуклая оболочка четырёх точек – *тетраэдр* с этими точками в качестве вершин

# Выпуклые оболочки

- Выпуклая комбинация: аффинная комбинация, в которой все коэффициенты неотрицательны  $\lambda_i \geq 0$ 
  - Вместе с  $\sum \lambda_i = 1$  это означает, что  $\lambda_i \in [0, 1]$
- Множество всех выпуклых комбинаций набора точек = *выпуклая оболочка*
- Выпуклая оболочка точки – сама *точка*
- Выпуклая оболочка двух точек – *отрезок* между этими точками
- Выпуклая оболочка трёх точек – *треугольник* с этими точками в качестве вершин
- Выпуклая оболочка четырёх точек – *тетраэдр* с этими точками в качестве вершин
- И т.д.

- При растеризации центры пикселей попадают внутрь треугольника



# Выпуклые оболочки

- При растеризации центры пикселей попадают внутрь треугольника
- $\Rightarrow$  Пиксели лежат в *выпуклой оболочке* исходных вершин

# Выпуклые оболочки

- При растеризации центры пикселей попадают внутрь треугольника
- $\Rightarrow$  Пиксели лежат в *выпуклой оболочке* исходных вершин
- $\Rightarrow$  Коэффициенты интерполяции всегда в диапазоне  $\lambda_i \in [0, 1]$

# Выпуклые оболочки

- При растеризации центры пикселей попадают внутрь треугольника
- $\Rightarrow$  Пиксели лежат в *выпуклой оболочке* исходных вершин
- $\Rightarrow$  Коэффициенты интерполяции всегда в диапазоне  $\lambda_i \in [0, 1]$
- $\Rightarrow$  Проинтерполированные значения лежат в выпуклой оболочке значений в вершинах
  - Например, если *out*-значения величины в вершинах были 1.5, 0.3 и 5.7, то *in*-значения в пикселях будут в диапазоне  $[0.3, 5.7]$  (с точностью до *floating-point* ошибок)

# Выпуклые оболочки

- При растеризации центры пикселей попадают внутрь треугольника
- $\Rightarrow$  Пиксели лежат в *выпуклой оболочке* исходных вершин
- $\Rightarrow$  Коэффициенты интерполяции всегда в диапазоне  $\lambda_i \in [0, 1]$
- $\Rightarrow$  Проинтерполированные значения лежат в выпуклой оболочке значений в вершинах
  - Например, если *out*-значения величины в вершинах были 1.5, 0.3 и 5.7, то *in*-значения в пикселях будут в диапазоне  $[0.3, 5.7]$  (с точностью до *floating-point* ошибок)
- N.B. Это может нарушаться при *мультисэмплинге*, о котором мы поговорим позднее

# Аффинные преобразования

- Линейные преобразования – сохраняют *линейные комбинации* векторов:  $S(\sum \lambda_i v_i) = \sum \lambda_i S v_i$

# Аффинные преобразования

- Линейные преобразования – сохраняют *линейные комбинации* векторов:  $S(\sum \lambda_i v_i) = \sum \lambda_i S v_i$
- Аффинные преобразования – сохраняют *аффинные комбинации* точек:  $S(\sum \lambda_i p_i) = \sum \lambda_i S p_i$

# Аффинные преобразования

- Линейные преобразования – сохраняют *линейные комбинации* векторов:  $S(\sum \lambda_i v_i) = \sum \lambda_i S v_i$
- Аффинные преобразования – сохраняют *аффинные комбинации* точек:  $S(\sum \lambda_i p_i) = \sum \lambda_i S p_i$
- Если выбрано начало координат  $p_0$ , преобразование  $S : \mathbb{A} \rightarrow \mathbb{A}$  можно понимать как преобразование соответствующего векторного пространства  $S : \mathbb{V} \rightarrow \mathbb{V}$  по формуле  $S(v) = S(p_0 + v) - p_0$

# Аффинные преобразования

- Линейные преобразования – сохраняют *линейные комбинации* векторов:  $S(\sum \lambda_i v_i) = \sum \lambda_i S v_i$
- Аффинные преобразования – сохраняют *аффинные комбинации* точек:  $S(\sum \lambda_i p_i) = \sum \lambda_i S p_i$
- Если выбрано начало координат  $p_0$ , преобразование  $S : \mathbb{A} \rightarrow \mathbb{A}$  можно понимать как преобразование соответствующего векторного пространства  $S : \mathbb{V} \rightarrow \mathbb{V}$  по формуле  $S(v) = S(p_0 + v) - p_0$
- Можно показать, что в таком виде любое аффинное преобразование это линейное преобразование + сдвиг на фиксированный вектор:  
 $S(v) = Av + b$  где
  - $A$  – линейное преобразование пространства  $\mathbb{V}$
  - $b$  – вектор из пространства  $\mathbb{V}$



# Аффинные преобразования

- В коде аффинное преобразование пространства размерности  $N$  можно представлять как *пару*  $(A, b)$  из матрицы  $N \times N$  и вектора размерности  $N$

# Аффинные преобразования

- В коде аффинное преобразование пространства размерности  $N$  можно представлять как *пару*  $(A, b)$  из матрицы  $N \times N$  и вектора размерности  $N$
- Пара  $(A, b)$  действует на точку/вектор  $v$  по формуле  $(A, b) \cdot v = Av + b$

# Аффинные преобразования

- В коде аффинное преобразование пространства размерности  $N$  можно представлять как пару  $(A, b)$  из матрицы  $N \times N$  и вектора размерности  $N$
- Пара  $(A, b)$  действует на точку/вектор  $v$  по формуле  $(A, b) \cdot v = Av + b$
- Композиция преобразований:
$$(A_1, b_1) \cdot (A_2, b_2) \cdot v = (A_1, b_1) \cdot (A_2v + b_2) = A_1(A_2v + b_2) + b_1 = (A_1A_2)v + (A_1b_2 + b_1) = (A_1A_2, A_1b_2 + b_1) \cdot v$$

# Аффинные преобразования

- В коде аффинное преобразование пространства размерности  $N$  можно представлять как пару  $(A, b)$  из матрицы  $N \times N$  и вектора размерности  $N$
- Пара  $(A, b)$  действует на точку/вектор  $v$  по формуле  $(A, b) \cdot v = Av + b$
- Композиция преобразований:
$$(A_1, b_1) \cdot (A_2, b_2) \cdot v = (A_1, b_1) \cdot (A_2v + b_2) = A_1(A_2v + b_2) + b_1 = (A_1A_2)v + (A_1b_2 + b_1) = (A_1A_2, A_1b_2 + b_1) \cdot v$$
- Тожественное преобразование:  $(I, 0)$

# Аффинные преобразования

- В коде аффинное преобразование пространства размерности  $N$  можно представлять как пару  $(A, b)$  из матрицы  $N \times N$  и вектора размерности  $N$
- Пара  $(A, b)$  действует на точку/вектор  $v$  по формуле  $(A, b) \cdot v = Av + b$
- Композиция преобразований:
$$(A_1, b_1) \cdot (A_2, b_2) \cdot v = (A_1, b_1) \cdot (A_2v + b_2) = A_1(A_2v + b_2) + b_1 = (A_1A_2)v + (A_1b_2 + b_1) = (A_1A_2, A_1b_2 + b_1) \cdot v$$
- Тожественное преобразование:  $(I, 0)$
- Обратное преобразование:
$$(A, b)^{-1} = (A^{-1}, -A^{-1}b)$$

- Повсеместно используются для задания положения, ориентации и размера объектов

- Повсеместно используются для задания положения, ориентации и размера объектов
- Векторная часть преобразования: положение 3D-объекта

- Повсеместно используются для задания положения, ориентации и размера объектов
- Векторная часть преобразования: положение 3D-объекта
- Матричная часть преобразования: вращение и размер объекта



# Аффинные преобразования

- Повсеместно используются для задания положения, ориентации и размера объектов
- Векторная часть преобразования: положение 3D-объекта
- Матричная часть преобразования: вращение и размер объекта
- При чём тут четвёртая координата?

- Трюк: можно вложить *вектор* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 0 в качестве последней координаты

# Однородные координаты

- Трюк: можно вложить *вектор* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 0 в качестве последней координаты
- Трюк: можно вложить *точку* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 1 в качестве последней координаты

# Однородные координаты

- Трюк: можно вложить *вектор* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 0 в качестве последней координаты
- Трюк: можно вложить *точку* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 1 в качестве последней координаты
- Такое представление точек и векторов называется *однородными координатами*

# Однородные координаты

- Трюк: можно вложить *вектор* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 0 в качестве последней координаты
- Трюк: можно вложить *точку* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 1 в качестве последней координаты
- Такое представление точек и векторов называется *однородными координатами*
- Сопласуется с операциями на векторах и точках

# Однородные координаты

- Трюк: можно вложить *вектор* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 0 в качестве последней координаты
- Трюк: можно вложить *точку* размерности  $N$  в векторное пространство размерности  $N + 1$ , добавив 1 в качестве последней координаты
- Такое представление точек и векторов называется *однородными координатами*
- Сопласуется с операциями на векторах и точках
- Сопласуется с аффинными комбинациями

- Позволяет представить аффинное преобразование  $N$ -мерного пространства как матрицу  $(N + 1) \times (N + 1)$ :

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

# Однородные координаты

- Позволяет представить аффинное преобразование  $N$ -мерного пространства как матрицу  $(N + 1) \times (N + 1)$ :

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

- Можно применять как к точкам, так и к векторам (векторы игнорируют сдвиг на  $b$ )



- Позволяет представить аффинное преобразование  $N$ -мерного пространства как матрицу  $(N + 1) \times (N + 1)$ :

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

- Можно применять как к точкам, так и к векторам (векторы игнорируют сдвиг на  $b$ )
- Композиция преобразований = умножение матриц

# Однородные координаты

- Позволяет представить аффинное преобразование  $N$ -мерного пространства как матрицу  $(N + 1) \times (N + 1)$ :

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

- Можно применять как к точкам, так и к векторам (векторы игнорируют сдвиг на  $b$ )
- Композиция преобразований = умножение матриц
- $\Rightarrow$  Позволяет удобно комбинировать преобразования (напр. масштабирование + сдвиг + поворот + другой сдвиг)

# Однородные координаты

- Позволяет представить аффинное преобразование  $N$ -мерного пространства как матрицу  $(N + 1) \times (N + 1)$ :

$$\begin{pmatrix} A & b \\ 0 & 1 \end{pmatrix}$$

- Можно применять как к точкам, так и к векторам (векторы игнорируют сдвиг на  $b$ )
- Композиция преобразований = умножение матриц
- $\Rightarrow$  Позволяет удобно комбинировать преобразования (напр. масштабирование + сдвиг + поворот + другой сдвиг)
- $\Rightarrow$  Позволяет собрать положение и ориентацию камеры и её проекцию в одну матрицу (обсудим на 4ой лекции)

# Точка в однородных координатах

$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 0 \end{pmatrix}$$

# Аффинное преобразование в однородных координатах

$$\begin{pmatrix} A_{1,1} & A_{1,2} & A_{1,3} & b_1 \\ A_{2,1} & A_{2,2} & A_{2,3} & b_2 \\ A_{3,1} & A_{3,2} & A_{3,3} & b_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

## Матрица сдвига на фиксированный вектор

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix}$$

# Матрица изотропного масштабирования

$$\begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} sp_x \\ sp_y \\ sp_z \\ 1 \end{pmatrix}$$
$$\begin{pmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} sv_x \\ sv_y \\ sv_z \\ 0 \end{pmatrix}$$



## Матрица поворота на угол $\theta$ в плоскости $XU$

$$\begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \cos \theta - p_y \sin \theta \\ p_x \sin \theta + p_y \cos \theta \\ p_z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \\ 0 \end{pmatrix} = \begin{pmatrix} v_x \cos \theta - v_y \sin \theta \\ v_x \sin \theta + v_y \cos \theta \\ v_z \\ 0 \end{pmatrix}$$

- [learnopengl.com/Getting-started/Transformations](https://learnopengl.com/Getting-started/Transformations)
- [open.gl/transformations](https://open.gl/transformations)
- [en.wikipedia.org/wiki/Affine\\_space](https://en.wikipedia.org/wiki/Affine_space)
- [en.wikipedia.org/wiki/Affine\\_transformation](https://en.wikipedia.org/wiki/Affine_transformation)