Компьютерная графика

Лекция 3: Объекты OpenGL, буферы, аттрибуты вершин, индексированный рендеринг

2021

Объекты OpenGL

- Шейдеры, шейдерные программы - программируемая часть конвейера

Объекты OpenGL

- Шейдеры, шейдерные программы - программируемая часть конвейера
- Буферы - хранят данные на GPU

Объекты OpenGL

- Шейдеры, шейдерные программы - программируемая часть конвейера
- Буферы - хранят данные на GPU
- Vertex Array - описывают аттрибуты вершин

Объекты OpenGL

- Шейдеры, шейдерные программы - программируемая часть конвейера
- Буферы - хранят данные на GPU
- Vertex Array - описывают аттрибуты вершин
- Текстуры - изображения, которые можно читать из шейдера, и в которые можно рисовать

Объекты OpenGL

- Шейдеры, шейдерные программы - программируемая часть конвейера
- Буферы - хранят данные на GPU
- Vertex Array - описывают аттрибуты вершин
- Текстуры - изображения, которые можно читать из шейдера, и в которые можно рисовать
- Renderbuffer - буферы, в которые можно рисовать

Объекты OpenGL

- Шейдеры, шейдерные программы - программируемая часть конвейера
- Буферы - хранят данные на GPU
- Vertex Array - описывают аттрибуты вершин
- Текстуры - изображения, которые можно читать из шейдера, и в которые можно рисовать
- Renderbuffer - буферы, в которые можно рисовать
- Framebuffer - содержат настройки рисования в текстуры и renderbuffer'ы

Создание/удаление объектов OpenGL

- Объект представляется идентификатором типа `GLuint`
  - Id уникален среди объектов одного типа (шейдеры, программы, ...)

Создание/удаление объектов OpenGL

- Объект представляется идентификатором типа `GLuint`
  - Id уникален среди объектов одного типа (шейдеры, программы, ...)
- Шейдеры и программы:
  - `glCreateShader()`/`glDeleteShader(shader)`
  - `glCreateProgram()`/`glDeleteProgram(program)`

Создание/удаление объектов OpenGL

- Объект представляется идентификатором типа `GLuint`
  - Id уникален среди объектов одного типа (шейдеры, программы, ...)
- Шейдеры и программы:
  - `glCreateShader()/glDeleteShader(shader)`
  - `glCreateProgram()/glDeleteProgram(program)`
- Остальные объекты:

`glGenBuffers(count, ptr)/glDeleteBuffers(count, ptr)`
`glGenVertexArrays/glDeleteVertexArrays`
`glGenTextures/glDeleteTextures`
`glGenRenderbuffers/glDeleteRenderbuffers`
`glGenFramebuffers/glDeleteFramebuffers`

Создание/удаление объектов OpenGL

- Объект представляется идентификатором типа `GLuint`
  - Id уникален среди объектов одного типа (шейдеры, программы, ...)
- Шейдеры и программы:
  - `glCreateShader()/glDeleteShader(shader)`
  - `glCreateProgram()/glDeleteProgram(program)`
- Остальные объекты:

  `glGenBuffers(count, ptr)/glDeleteBuffers(count, ptr)`
  `glGenVertexArrays/glDeleteVertexArrays`
  `glGenTextures/glDeleteTextures`
  `glGenRenderbuffers/glDeleteRenderbuffers`
  `glGenFramebuffers/glDeleteFramebuffers`

  - Можно создать/удалить один объект:

```
GLuint texture;
glGenTextures(1, &texture);
...
glDeleteTexture(1, &texture);
```

Объекты OpenGL

- Подразумевается, что объекты переиспользуются по максимуму
- Не нужно создавать новую текстуру каждый кадр - создайте один раз и переиспользуйте её
- Не нужно создавать новый буфер при каждом обновлении данных - создайте один раз и переиспользуйте его

Объекты OpenGL с нулевым id

- Как правило, объект с нулевым id считается несуществующим (как нулевой указатель)

Объекты OpenGL с нулевым id

- Как правило, объект с нулевым id считается несуществующим (как нулевой указатель)
- Исключения:
  - Framebuffer с нулевым id - default framebuffer, рисует в окно, привязанное к контексту OpenGL, его нельзя менять

Объекты OpenGL с нулевым id

- Как правило, объект с нулевым id считается несуществующим (как нулевой указатель)
- Исключения:
  - Framebuffer с нулевым id - default framebuffer, рисует в окно, привязанное к контексту OpenGL, его нельзя менять
  - В OpenGL ES: Vertex Array с нулевым id - ничем не отличается от других vertex array'ев, но существует (создан) по умолчанию

Объекты OpenGL с нулевым id

- Как правило, объект с нулевым id считается несуществующим (как нулевой указатель)
- Исключения:
  - Framebuffer с нулевым id - default framebuffer, рисует в окно, привязанное к контексту OpenGL, его нельзя менять
  - В OpenGL ES: Vertex Array с нулевым id - ничем не отличается от других vertex array'ев, но существует (создан) по умолчанию
- Функции создания объектов никогда не возвращают нулевой id
- Функции удаления объектов игнорируют нулевой id

Работа с объектами OpenGL

- Почти всегда чтобы работать с объектом, нужно сделать его "текущим"
  - Текущий объект запоминается в контексте OpenGL
  - Если вы работаете с одним контекстом, можно считать, что id текущего объекта - глобальная константа

Работа с объектами OpenGL

- Почти всегда чтобы работать с объектом, нужно сделать его "текущим"
  - Текущий объект запоминается в контексте OpenGL
  - Если вы работаете с одним контекстом, можно считать, что id текущего объекта - глобальная константа
- Некоторые функции не требуют выставления объекта текущим: `glShaderSource`, `glCompileShader`, `glLinkProgram`, `glGetUniformLocation`, ...

Работа с объектами OpenGL

- Почти всегда чтобы работать с объектом, нужно сделать его "текущим"
  - Текущий объект запоминается в контексте OpenGL
  - Если вы работаете с одним контекстом, можно считать, что id текущего объекта - глобальная константа
- Некоторые функции не требуют выставления объекта текущим: `glShaderSource`, `glCompileShader`, `glLinkProgram`, `glGetUniformLocation`, ...
- Некоторые объекты нельзя сделать текущими - шейдеры

Работа с объектами OpenGL

- Сделать программу текущей: `glUseProgram(program)`
  - Функции `glUniform1f`, ... работают с текущей программой

Работа с объектами OpenGL

- Сделать программу текущей: `glUseProgram(program)`
  - Функции `glUniform1f`, ... работают с текущей программой
- Сделать vertex array текущим: `glBindVertexArray(vao)`
  - Функции работы с vertex array используют текущий vertex array

Работа с объектами OpenGL

- Сделать программу текущей: `glUseProgram(program)`
  - Функции `glUniform1f`, ... работают с текущей программой
- Сделать vertex array текущим: `glBindVertexArray(vao)`
  - Функции работы с vertex array используют текущий vertex array
- Функции рисования (`glDrawArrays`) используют текущую шейдерную программу и текущий vertex array

Работа с объектами OpenGL

- Для буферов, текстур, framebuffer'ов и renderbuffer'ов нет одного текущего объекта, но есть текущий объект для конкретного target'а

Работа с объектами OpenGL

- Для буферов, текстур, framebuffer'ов и renderbuffer'ов нет одного текущего объекта, но есть текущий объект для конкретного target'а
- Можно считать, что есть словарь `Target -> Id` текущих объектов

Работа с объектами OpenGL

- Для буферов, текстур, framebuffer'ов и renderbuffer'ов нет одного текущего объекта, но есть текущий объект для конкретного target'а
- Можно считать, что есть словарь `Target -> Id` текущих объектов
- Для каждого вида объектов (буферы, текстуры, ...) есть отдельный словарь и отдельный набор возможных значений Target

Работа с объектами OpenGL

- Для буферов, текстур, framebuffer'ов и renderbuffer'ов нет одного текущего объекта, но есть текущий объект для конкретного target'а
- Можно считать, что есть словарь `Target -> Id` текущих объектов
- Для каждого вида объектов (буферы, текстуры, ...) есть отдельный словарь и отдельный набор возможных значений Target
- Смысл и особенности разных значений Target зависят от вида объекта

Работа с объектами OpenGL

- Для буферов, текстур, framebuffer'ов и renderbuffer'ов нет одного текущего объекта, но есть текущий объект для конкретного target'а
- Можно считать, что есть словарь `Target -> Id` текущих объектов
- Для каждого вида объектов (буферы, текстуры, ...) есть отдельный словарь и отдельный набор возможных значений Target
- Смысл и особенности разных значений Target зависят от вида объекта
- `glBindBuffer(target, id)`
- `glBindTexture(target, id)`
- `glBindRenderbuffer(target, id)`
- `glBindFramebuffer(target, id)`

Буферы

- Могут хранить произвольные данные на GPU

Буферы

- Могут хранить произвольные данные на GPU
- `glGenBuffers/glDeleteBuffers`

Буферы

- Могут хранить произвольные данные на GPU
- `glGenBuffers`/`glDeleteBuffers`
- Возможные значения target для `glBindBuffer`:
  - `GL_ARRAY_BUFFER` (VBO) - массив вершин

Буферы

- Могут хранить произвольные данные на GPU
- `glGenBuffers`/`glDeleteBuffers`
- Возможные значения target для `glBindBuffer`:
  - `GL_ARRAY_BUFFER` (VBO) - массив вершин
  - `GL_ELEMENT_ARRAY_BUFFER` (EBO) - массив индексов вершин

Буферы

- Могут хранить произвольные данные на GPU
- `glGenBuffers`/`glDeleteBuffers`
- Возможные значения target для `glBindBuffer`:
  - `GL_ARRAY_BUFFER` (VBO) - массив вершин
  - `GL_ELEMENT_ARRAY_BUFFER` (EBO) - массив индексов вершин
  - `GL_UNIFORM_BUFFER` (UBO) - массив значений uniform-переменных

Буферы

- Могут хранить произвольные данные на GPU
- `glGenBuffers`/`glDeleteBuffers`
- Возможные значения target для `glBindBuffer`:
  - `GL_ARRAY_BUFFER` (VBO) - массив вершин
  - `GL_ELEMENT_ARRAY_BUFFER` (EBO) - массив индексов вершин
  - `GL_UNIFORM_BUFFER` (UBO) - массив значений uniform-переменных
  - ...и другие

Буферы

- Могут хранить произвольные данные на GPU
- `glGenBuffers`/`glDeleteBuffers`
- Возможные значения target для `glBindBuffer`:
  - `GL_ARRAY_BUFFER` (VBO) - массив вершин
  - `GL_ELEMENT_ARRAY_BUFFER` (EBO) - массив индексов вершин
  - `GL_UNIFORM_BUFFER` (UBO) - массив значений uniform-переменных
  - ...и другие
- Текущий `GL_ELEMENT_ARRAY_BUFFER` хранится не глобально, а в текущем VAO

Буферы: запись данных

- Выделить память на GPU и загрузить данные:

```
glBufferData(GLenum target, GLsizeiptr size,
    const GLvoid * data, GLenum usage)
```

Буферы: запись данных

- Выделить память на GPU и загрузить данные:

```
glBufferData(GLenum target, GLsizeiptr size,
    const GLvoid * data, GLenum usage)
```

  - `target` - `GL_ARRAY_BUFFER` и т.п.
  - `size` - размер данных в байтах
  - `data` - указатель на данные
  - `usage` - подсказка драйверу о том, как данные будут использоваться

Буферы: запись данных

- Выделить память на GPU и загрузить данные:

```
glBufferData(GLenum target, GLsizeiptr size,
    const GLvoid * data, GLenum usage)
```

  - `target` - `GL_ARRAY_BUFFER` и т.п.
  - `size` - размер данных в байтах
  - `data` - указатель на данные
  - `usage` - подсказка драйверу о том, как данные будут использоваться
- Если `data = nullptr`, буфер будет выделен, но данные будут не инициализированы

Буферы: запись данных

- Выделить память на GPU и загрузить данные:
  `glBufferData(GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage)`
  - `target` - `GL_ARRAY_BUFFER` и т.п.
  - `size` - размер данных в байтах
  - `data` - указатель на данные
  - `usage` - подсказка драйверу о том, как данные будут использоваться
- Если `data = nullptr`, буфер будет выделен, но данные будут не инициализированы
- Если буфер уже содержал данные, они заменяются новыми (и происходит реаллокация памяти)

Буферы: запись данных

- Выделить память на GPU и загрузить данные:
  `glBufferData(GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage)`
  - `target` - `GL_ARRAY_BUFFER` и т.п.
  - `size` - размер данных в байтах
  - `data` - указатель на данные
  - `usage` - подсказка драйверу о том, как данные будут использоваться
- Если `data = nullptr`, буфер будет выделен, но данные будут не инициализированы
- Если буфер уже содержал данные, они заменяются новыми (и происходит реаллокация памяти)
- После вызова `glBufferData` с данными по указателю `data` можно делать всё, что угодно (в т.ч. удалить)
- Копирование данных в память GPU тоже происходит асинхронно

Буферы: запись данных

- Выделить память на GPU и загрузить данные:
  `glBufferData(GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage)`
  - `target` - `GL_ARRAY_BUFFER` и т.п.
  - `size` - размер данных в байтах
  - `data` - указатель на данные
  - `usage` - подсказка драйверу о том, как данные будут использоваться
- Если `data = nullptr`, буфер будет выделен, но данные будут не инициализированы
- Если буфер уже содержал данные, они заменяются новыми (и происходит реаллокация памяти)
- После вызова `glBufferData` с данными по указателю `data` можно делать всё, что угодно (в т.ч. удалить)
- Копирование данных в память GPU тоже происходит асинхронно
  - $\Rightarrow$ Драйвер, скорее всего, сначала копирует данные в собственную память

Буферы: параметр usage

GL_STATIC_DRAW	GL_STATIC_READ	GL_STATIC_COPY
GL_DYNAMIC_DRAW	GL_DYNAMIC_READ	GL_DYNAMIC_COPY
GL_STREAM_DRAW	GL_STREAM_READ	GL_STREAM_COPY

Буферы: параметр usage

`GL_STATIC_DRAW`	`GL_STATIC_READ`	`GL_STATIC_COPY`
`GL_DYNAMIC_DRAW`	`GL_DYNAMIC_READ`	`GL_DYNAMIC_COPY`
`GL_STREAM_DRAW`	`GL_STREAM_READ`	`GL_STREAM_COPY`

- Данные будут обновляться
  - `STATIC` - один раз
  - `DYNAMIC` - иногда
  - `STREAM` - почти каждый кадр

Буферы: параметр usage

`GL_STATIC_DRAW`	`GL_STATIC_READ`	`GL_STATIC_COPY`
`GL_DYNAMIC_DRAW`	`GL_DYNAMIC_READ`	`GL_DYNAMIC_COPY`
`GL_STREAM_DRAW`	`GL_STREAM_READ`	`GL_STREAM_COPY`

- Данные будут обновляться
  - `STATIC` - один раз
  - `DYNAMIC` - иногда
  - `STREAM` - почти каждый кадр
- Буфер будет использоваться для:
  - `DRAW` - записи данных в него
  - `READ` - чтения данных из него
  - `COPY` - и записи, и чтения

Буферы: параметр usage

`GL_STATIC_DRAW`	`GL_STATIC_READ`	`GL_STATIC_COPY`
`GL_DYNAMIC_DRAW`	`GL_DYNAMIC_READ`	`GL_DYNAMIC_COPY`
`GL_STREAM_DRAW`	`GL_STREAM_READ`	`GL_STREAM_COPY`

- Данные будут обновляться
  - `STATIC` - один раз
  - `DYNAMIC` - иногда
  - `STREAM` - почти каждый кадр
- Буфер будет использоваться для:
  - `DRAW` - записи данных в него
  - `READ` - чтения данных из него
  - `COPY` - и записи, и чтения
- Это только подсказка драйверу и не влияет на корректность работы

Буферы: запись и чтение данных

- Загрузить данные в часть буфера:

```
glBufferSubData(GLenum target, GLintptr offset,
    GLsizeiptr size, const GLvoid * data)
```

Буферы: запись и чтение данных

- Загрузить данные в часть буфера:

```
glBufferSubData(GLenum target, GLintptr offset,
    GLsizeiptr size, const GLvoid * data)
```

  - Гарантированно не реаллоцирует память GPU
  - Память уже должна быть выделена (`glBufferData`)

Буферы: запись и чтение данных

- Загрузить данные в часть буфера:

```
glBufferSubData(GLenum target, GLintptr offset,
    GLsizeiptr size, const GLvoid * data)
```

  - Гарантированно не реаллоцирует память GPU
  - Память уже должна быть выделена (`glBufferData`)
- Прочитать данные из буфера:

```
glGetBufferSubData(GLenum target, GLintptr offset,
    GLsizeiptr size, GLvoid * data)
```

Буферы: запись и чтение данных

- Загрузить данные в часть буфера:

```
glBufferSubData(GLenum target, GLintptr offset,
    GLsizeiptr size, const GLvoid * data)
```

  - Гарантированно не реаллоцирует память GPU
  - Память уже должна быть выделена (`glBufferData`)
- Прочитать данные из буфера:

```
glGetBufferSubData(GLenum target, GLintptr offset,
    GLsizeiptr size, GLvoid * data)
```

  - К моменту выхода из этой функции данные уже прочитаны
  - $\Rightarrow$ Синхронная функция, блокирующая исполнение

Mapped buffer

- Можно получить виртуальный указатель на буфер или его часть и пользоваться им для чтения/записи

Mapped buffer

- Можно получить виртуальный указатель на буфер или его часть и пользоваться им для чтения/записи
- `glMapBuffer(target, access)` - возвращает mapped указатель
- `access` может принимать значения
  - `GL_READ_ONLY` - по указателю можно читать
  - `GL_WRITE_ONLY` - по указателю можно писать
  - `GL_READ_WRITE` - по указателю можно читать и писать

Mapped buffer

- Можно получить виртуальный указатель на буфер или его часть и пользоваться им для чтения/записи
- `glMapBuffer(target, access)` - возвращает mapped указатель
- `access` может принимать значения
  - `GL_READ_ONLY` - по указателю можно читать
  - `GL_WRITE_ONLY` - по указателю можно писать
  - `GL_READ_WRITE` - по указателю можно читать и писать
- `glUnmapBuffer(target)`

Mapped buffer

- Можно получить виртуальный указатель на буфер или его часть и пользоваться им для чтения/записи
- `glMapBuffer(target, access)` - возвращает mapped указатель
- `access` может принимать значения
  - `GL_READ_ONLY` - по указателю можно читать
  - `GL_WRITE_ONLY` - по указателю можно писать
  - `GL_READ_WRITE` - по указателю можно читать и писать
- `glUnmapBuffer(target)`
- Между `glMapBuffer` и `glUnmapBuffer` работать с буфером (загружать данные другими методами, использовать данные для рисования) нельзя

Mapped buffer

- Можно получить виртуальный указатель на буфер или его часть и пользоваться им для чтения/записи
- `glMapBuffer(target, access)` - возвращает mapped указатель
- `access` может принимать значения
  - `GL_READ_ONLY` - по указателю можно читать
  - `GL_WRITE_ONLY` - по указателю можно писать
  - `GL_READ_WRITE` - по указателю можно читать и писать
- `glUnmapBuffer(target)`
- Между `glMapBuffer` и `glUnmapBuffer` работать с буфером (загружать данные другими методами, использовать данные для рисования) нельзя
- После `glUnmapBuffer` mapped указатель использовать нельзя

Буферы: типичный пример использования

```
GLuint vbo;
glGenBuffers(1, &vbo);

std::vector<vertex> vertices;
...

glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER,
    vertices.size() * sizeof(vertices[0]),
    vertices.data(), GL_STATIC_DRAW);
```

Буферы: ссылки

- khronos.org/opengl/wiki/Buffer_Object
- songho.ca/opengl/gl_vbo.html

Аттрибуты вершин: шейдер

```
#version 330 core

layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;
layout (location = 2) in vec4 color;
layout (location = 3) in int materialID;

void main() {
    ...
}
```

Аттрибуты вершин

- Индекс: `0`, `1`, ... `glGet(GL_MAX_VERTEX_ATTRIBS)`
  - `glGet(GL_MAX_VERTEX_ATTRIBS)` ≥ 16

Аттрибуты вершин

- Индекс: `0, 1, ...` `glGet(GL_MAX_VERTEX_ATTRIBS)`
  - `glGet(GL_MAX_VERTEX_ATTRIBS)` ≥ 16
- Включен/выключен
  - `glEnableVertexAttribArray(index)` - включить аттрибут
  - Выключены по умолчанию
  - Хранится в vertex array

Аттрибуты вершин: хранение

Параметры хранения данных аттрибута:

Аттрибуты вершин: хранение

Параметры хранения данных аттрибута:

- Формат:
  - Количество компонент - 1, 2, 3, 4

Аттрибуты вершин: хранение

Параметры хранения данных аттрибута:

- Формат:
  - Количество компонент - 1, 2, 3, 4
  - Тип - `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_DOUBLE`, `GL_INT_2_10_10_10_REV`, `GL_UNSIGNED_INT_2_10_10_10_REV`

Аттрибуты вершин: хранение

Параметры хранения данных аттрибута:

- Формат:
  - Количество компонент - 1, 2, 3, 4
  - Тип - `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_DOUBLE`, `GL_INT_2_10_10_10_REV`, `GL_UNSIGNED_INT_2_10_10_10_REV`
  - Нормированный / не нормированный

Аттрибуты вершин: хранение

Параметры хранения данных аттрибута:

- Формат:
  - Количество компонент - 1, 2, 3, 4
  - Тип - `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_DOUBLE`, `GL_INT_2_10_10_10_REV`, `GL_UNSIGNED_INT_2_10_10_10_REV`
  - Нормированный / не нормированный
- Расположение данных:
  - Адрес начала данных (на GPU)
  - Расстояние (в байтах) между значениями этого аттрибута, соответствующими соседним вершинам

Аттрибуты вершин: хранение

Параметры хранения данных аттрибута:

- Формат:
  - Количество компонент - 1, 2, 3, 4
  - Тип - `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, `GL_UNSIGNED_INT`, `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_DOUBLE`, `GL_INT_2_10_10_10_REV`, `GL_UNSIGNED_INT_2_10_10_10_REV`
  - Нормированный / не нормированный
- Расположение данных:
  - Адрес начала данных (на GPU)
  - Расстояние (в байтах) между значениями этого аттрибута, соответствующими соседним вершинам
- Всё это запоминается в vertex array

Аттрибуты вершин: нормированность

- Как интерпретировать значение аттрибута, который в шейдере объявлен как floating-point (`float`, `vec3`, и т.п.), но в данных он хранится как целочисленный?

Аттрибуты вершин: нормированность

- Как интерпретировать значение аттрибута, который в шейдере объявлен как floating-point (`float`, `vec3`, и т.п.), но в данных он хранится как целочисленный?
- 2 варианта:
  - Не нормированный: передавать, как есть, делая преобразование `int -> float`

Аттрибуты вершин: нормированность

- Как интерпретировать значение аттрибута, который в шейдере объявлен как floating-point (`float`, `vec3`, и т.п.), но в данных он хранится как целочисленный?
- 2 варианта:
  - Не нормированный: передавать, как есть, делая преобразование `int -> float`
  - Нормированный: преобразовывать диапазон целочисленных значений в $[-1, 1]$ (для знаковых) или $[0, 1]$ (для беззнаковых) `unsigned short: x -> x / 65535` `short: x -> (2 x + 1) / 65535`

Аттрибуты вершин: нормированность

- Как интерпретировать значение аттрибута, который в шейдере объявлен как floating-point (`float`, `vec3`, и т.п.), но в данных он хранится как целочисленный?
- 2 варианта:
  - Не нормированный: передавать, как есть, делая преобразование `int -> float`
  - Нормированный: преобразовывать диапазон целочисленных значений в $[-1, 1]$ (для знаковых) или $[0, 1]$ (для беззнаковых) `unsigned short: x -> x / 65535` `short: x -> (2 x + 1) / 65535`
- Полезно для передачи цвета: `0 .. 255` превращаются в стандартные для OpenGL `0.0 .. 1.0`

Аттрибуты вершин: нормированность

- Как интерпретировать значение аттрибута, который в шейдере объявлен как floating-point (`float`, `vec3`, и т.п.), но в данных он хранится как целочисленный?
- 2 варианта:
  - Не нормированный: передавать, как есть, делая преобразование `int -> float`
  - Нормированный: преобразовывать диапазон целочисленных значений в $[-1, 1]$ (для знаковых) или $[0, 1]$ (для беззнаковых) `unsigned short: x -> x / 65535` `short: x -> (2 x + 1) / 65535`
- Полезно для передачи цвета: `0 .. 255` превращаются в стандартные для OpenGL `0.0 .. 1.0`
- Полезно для компактного хранения аттрибутов

Аттрибуты вершин: нормированность

- Как интерпретировать значение аттрибута, который в шейдере объявлен как floating-point (`float`, `vec3`, и т.п.), но в данных он хранится как целочисленный?
- 2 варианта:
  - Не нормированный: передавать, как есть, делая преобразование `int -> float`
  - Нормированный: преобразовывать диапазон целочисленных значений в $[-1, 1]$ (для знаковых) или $[0, 1]$ (для беззнаковых) `unsigned short: x -> x / 65535` `short: x -> (2 x + 1) / 65535`
- Полезно для передачи цвета: `0 .. 255` превращаются в стандартные для OpenGL `0.0 .. 1.0`
- Полезно для компактного хранения аттрибутов
- Имеет смысл только для floating-point аттрибутов

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

- `index` - индекс аттрибута

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

- `index` - индекс аттрибута
- `size` - размерность (число компонент)

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

- `index` - индекс аттрибута
- `size` - размерность (число компонент)
- `type` - тип компонент

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

- `index` - индекс аттрибута
- `size` - размерность (число компонент)
- `type` - тип компонент
- `normalized` - нормированный или нет

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

- `index` - индекс аттрибута
- `size` - размерность (число компонент)
- `type` - тип компонент
- `normalized` - нормированный или нет
- `stride` - расстояние между соседними значениями

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

- `index` - индекс аттрибута
- `size` - размерность (число компонент)
- `type` - тип компонент
- `normalized` - нормированный или нет
- `stride` - расстояние между соседними значениями
- `pointer` - указатель на данные

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

- `index` - индекс аттрибута
- `size` - размерность (число компонент)
- `type` - тип компонент
- `normalized` - нормированный или нет
- `stride` - расстояние между соседними значениями
- `pointer` - указатель на данные
  - На самом деле, `pointer` - сдвиг относительно начала памяти текущего `GL_ARRAY_BUFFER`
  - Например, если данные этого аттрибута начинаются на 12ом байте текущего `GL_ARRAY_BUFFER`, нужно передать `(const void *)(12)`

Аттрибуты вершин: floating-point

```
glVertexAttribPointer(GLuint index, GLint size,
    GLenum type, GLboolean normalized,
    GLsizei stride, const GLvoid * pointer)
```

- `index` - индекс аттрибута
- `size` - размерность (число компонент)
- `type` - тип компонент
- `normalized` - нормированный или нет
- `stride` - расстояние между соседними значениями
- `pointer` - указатель на данные
  - На самом деле, `pointer` - сдвиг относительно начала памяти текущего `GL_ARRAY_BUFFER`
  - Например, если данные этого аттрибута начинаются на 12ом байте текущего `GL_ARRAY_BUFFER`, нужно передать `(const void *)(12)`
  - Разные аттрибуты могут лежать в разных буферах

Аттрибуты вершин: целочисленные

```
glVertexAttribIPointer(GLuint index, GLint size,
    GLenum type,
    GLsizei stride, const GLvoid * pointer)
```

- Всё то же самое, но
  - Нет параметра `normalized`
  - `type` может быть только целочисленным

Аттрибуты вершин: stride

- Пусть `start` - смещение, указанное в параметре `pointer` функции `glVertexAttribPointer`
- Тогда соответстующий аттрибут вершины с номером `i` будет взят по смещению `start + stride * i`

Аттрибуты вершин: stride

- Пусть `start` - смещение, указанное в параметре `pointer` функции `glVertexAttribPointer`
- Тогда соответстующий аттрибут вершины с номером `i` будет взят по смещению `start + stride * i`
  - Нулевая вершина: `start`
  - Первая вершина: `start + stride`
  - Вторая вершина: `start + 2 * stride`
  - И т.д.

Аттрибуты вершин: stride

- Пусть `start` - смещение, указанное в параметре `pointer` функции `glVertexAttribPointer`
- Тогда соответстующий аттрибут вершины с номером `i` будет взят по смещению `start + stride * i`
  - Нулевая вершина: `start`
  - Первая вершина: `start + stride`
  - Вторая вершина: `start + 2 * stride`
  - И т.д.
- Если `stride = 0`, `stride` считается равным размеру аттрибута `stride = size * sizeof(type)`
  - Например, для `size = 3` и `type = GL_UNSIGNED_INT`, `stride` будет вычислен как
    `stride = 3 * sizeof(unsigned int) = 3 * 4 = 12`

Аттрибуты вершин: stride

- Пусть `start` - смещение, указанное в параметре `pointer` функции `glVertexAttribPointer`
- Тогда соответстующий аттрибут вершины с номером `i` будет взят по смещению `start + stride * i`
  - Нулевая вершина: `start`
  - Первая вершина: `start + stride`
  - Вторая вершина: `start + 2 * stride`
  - И т.д.
- Если `stride = 0`, `stride` считается равным размеру аттрибута `stride = size * sizeof(type)`
  - Например, для `size = 3` и `type = GL_UNSIGNED_INT`, `stride` будет вычислен как
    `stride = 3 * sizeof(unsigned int) = 3 * 4 = 12`
- Нужен для гибкости хранения аттрибутов

Аттрибуты вершин: пример array-of-structs, один буфер

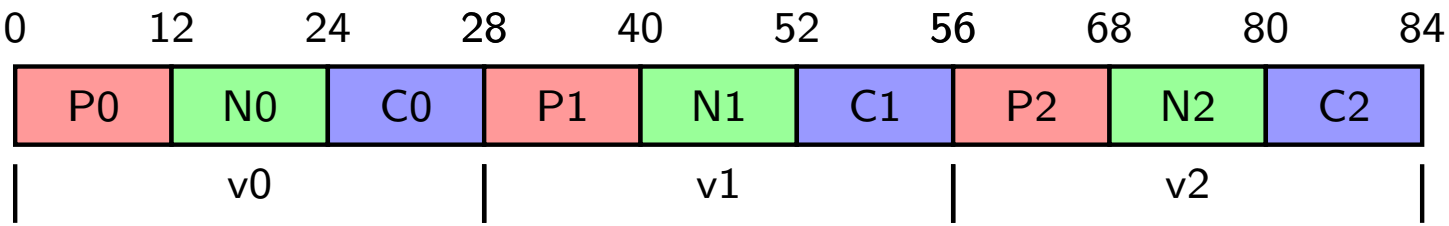
```
struct vertex {
    float position[3];
    float normal[3];
    unsigned char color[4];
};
vertex vertices[N];

glBindBuffer(GL_ARRAY_BUFFER, vbo);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
    sizeof(vertex), (void*)(0));

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
    sizeof(vertex), (void*)(12));

glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 4, GL_UNSIGNED_BYTE, GL_TRUE,
    sizeof(vertex), (void*)(24));
```



Аттрибуты вершин: пример struct-of-arrays, один буфер

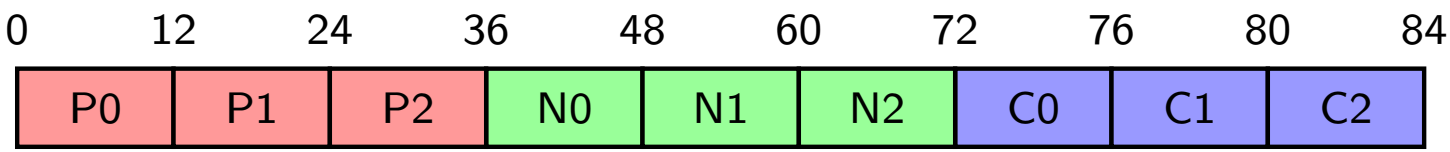
```
float positions[3 * N];
float normal[3 * N];
unsigned char color[4 * N];

glBindBuffer(GL_ARRAY_BUFFER, vbo);

glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
    0, (void*)(0));

glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
    0, (void*)(12 * N));

glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 4, GL_UNSIGNED_BYTE, GL_TRUE,
    0, (void*)(24 * N));
```



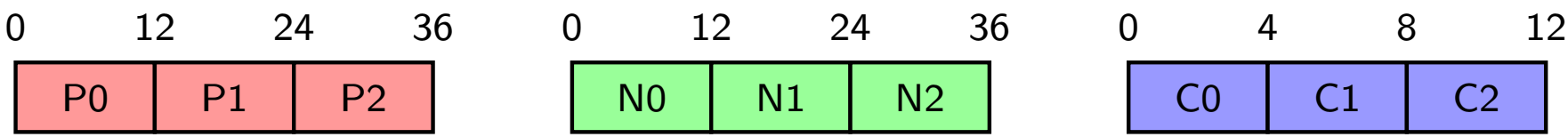
Аттрибуты вершин: пример struct-of-arrays, три буфера

```
float positions[3 * N];
float normal[3 * N];
unsigned char color[4 * N];

glBindBuffer(GL_ARRAY_BUFFER, position_vbo);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE,
    0, (void*)(0));

glBindBuffer(GL_ARRAY_BUFFER, normal_vbo);
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE,
    0, (void*)(0));

glBindBuffer(GL_ARRAY_BUFFER, color_vbo);
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 4, GL_UNSIGNED_BYTE, GL_TRUE,
    0, (void*)(0));
```



Аттрибуты вершин: как хранить?

- Если не можете выбрать, используйте array-of-structs

Аттрибуты вершин: как хранить?

- Если не можете выбрать, используйте array-of-structs
- Если часть аттрибутов постоянны, а другую часть нужно иногда обновлять - разделите эти части по двум разным VBO

Аттрибуты вершин: пример полностью

```
// Инициализация:
program = createProgram()
vertices = generateVertices()
vbo = createVBO(vertices)
vao = createVAO()
setupVAO(vao, vbo)

// Рендеринг:
glUseProgram(program)
glBindVertexArray(vao)
glDrawArrays(GL_TRIANGLES, 0, count)
```

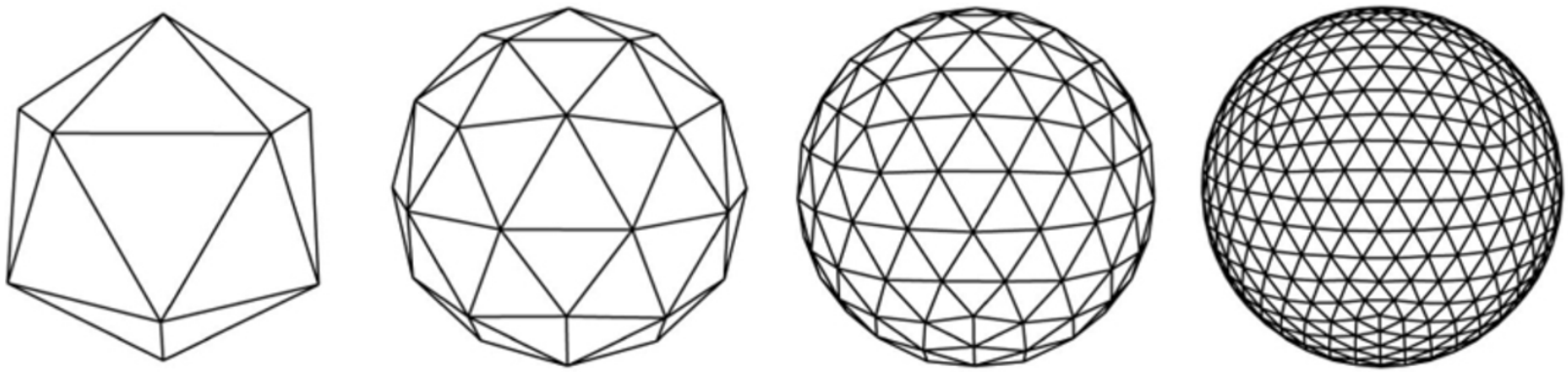
Аттрибуты вершин: пример полностью

```
// Инициализация:
program = createProgram()
vertices = generateVertices()
vbo = createVBO(vertices)
vao = createVAO()
setupVAO(vao, vbo)

// Рендеринг:
glUseProgram(program)
glBindVertexArray(vao)
glDrawArrays(GL_TRIANGLES, 0, count)
```

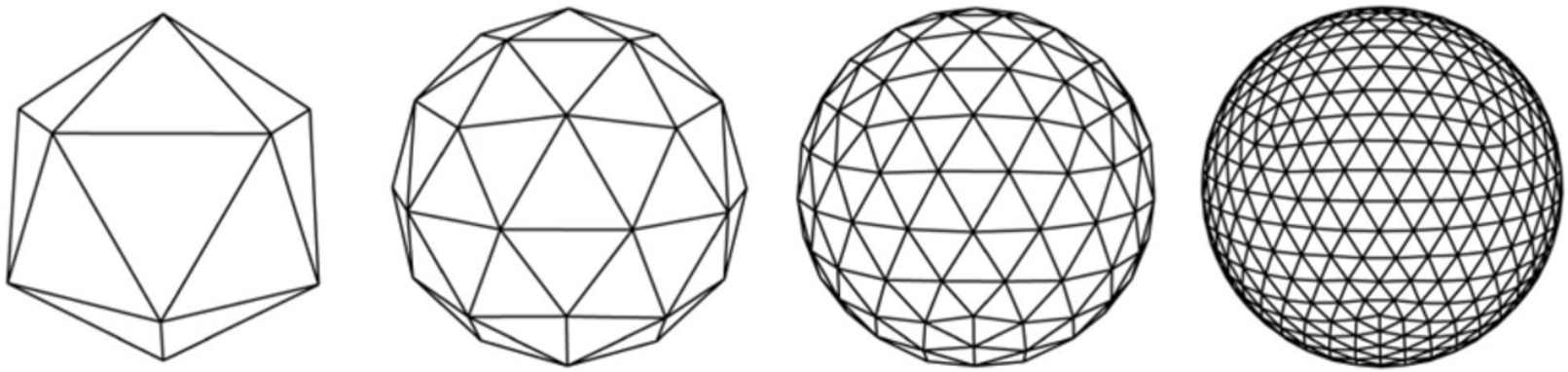
- khronos.org/opengl/wiki/Vertex_Specification

Индексированный рендеринг



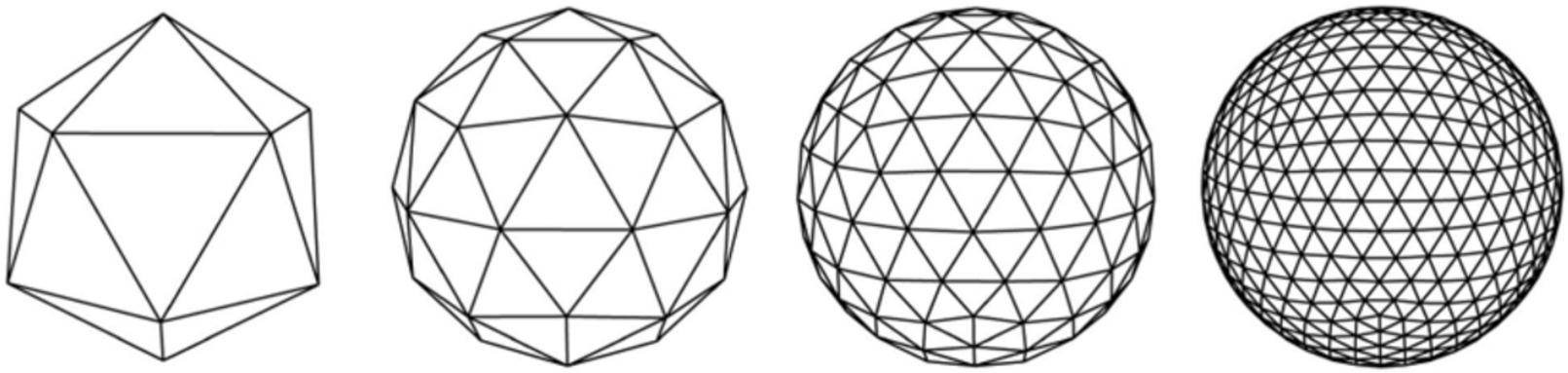
- В моделях одна вершина часто входит в несколько треугольников (в среднем, 6)

Индексированный рендеринг



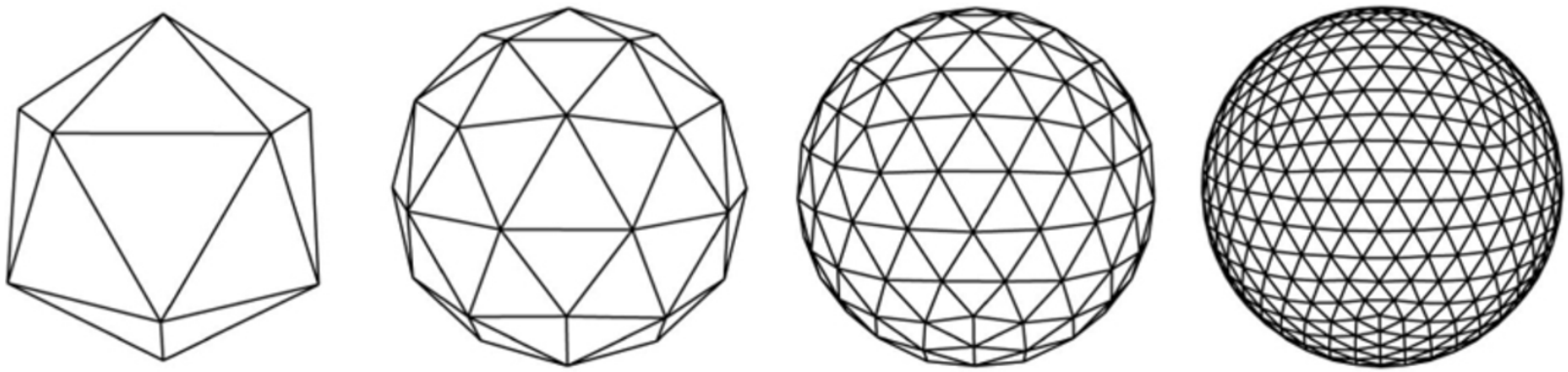
- В моделях одна вершина часто входит в несколько треугольников (в среднем, 6)
- Если использовать `GL_TRIANGLES`, придётся копировать каждую вершину несколько раз

Индексированный рендеринг



- В моделях одна вершина часто входит в несколько треугольников (в среднем, 6)
- Если использовать `GL_TRIANGLES`, придётся копировать каждую вершину несколько раз
- `GL_TRIANGLE_STRIP` и т.п. только частично решают проблему

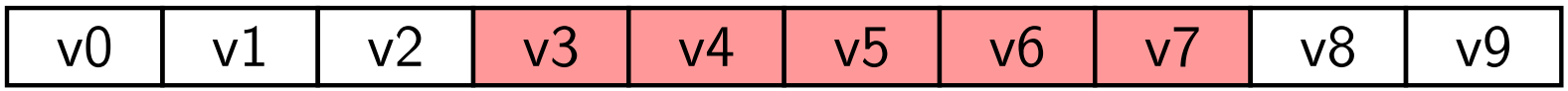
Индексированный рендеринг



- В моделях одна вершина часто входит в несколько треугольников (в среднем, 6)
- Если использовать `GL_TRIANGLES`, придётся копировать каждую вершину несколько раз
- `GL_TRIANGLE_STRIP` и т.п. только частично решают проблему
- $\Rightarrow$ Индексированный рендеринг

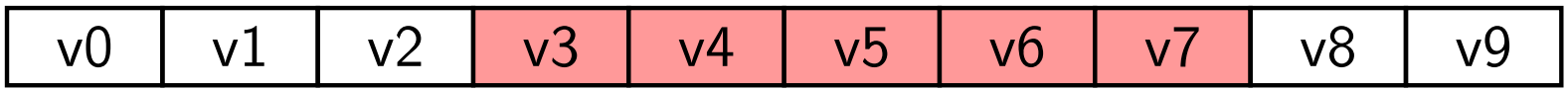
Индексированный рендеринг

- `glDrawArrays(mode, 3, 5)`:

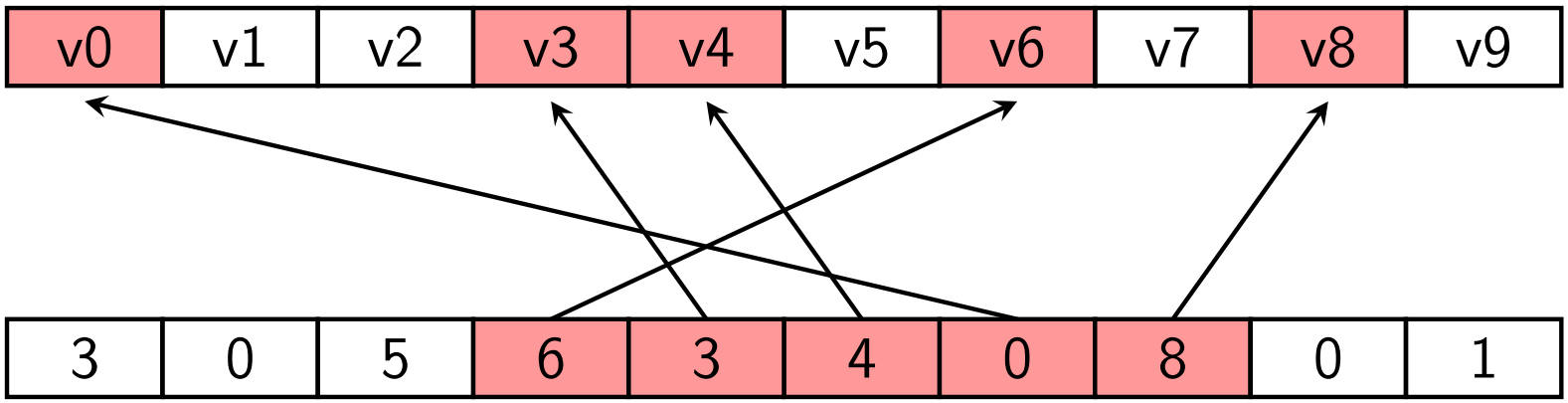


Индексированный рендеринг

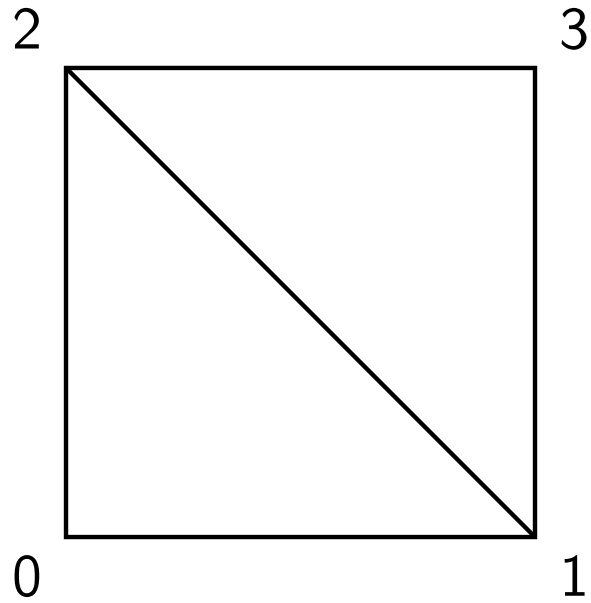
- `glDrawArrays(mode, 3, 5)`:



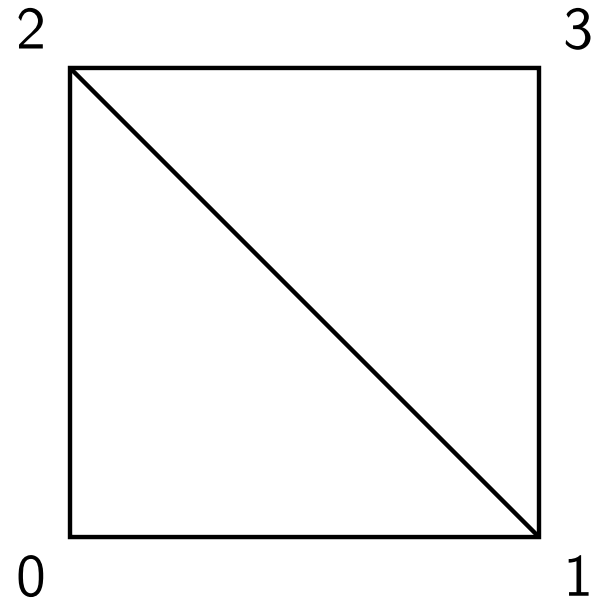
- `glDrawElements(mode, ...)`:



Индексированный рендеринг: квадрат

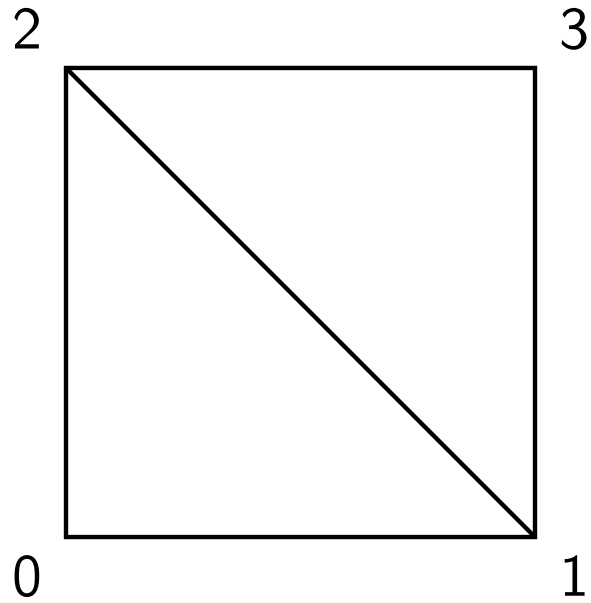


Индексированный рендеринг: квадрат



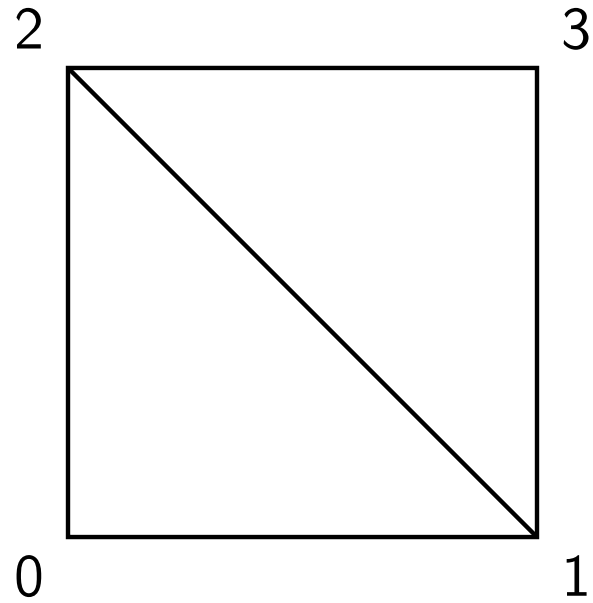
- Без индексов, `GL_TRIANGLES`:
  - Вершины: `v0, v1, v2, v2, v1, v3`

Индексированный рендеринг: квадрат



- Без индексов, `GL_TRIANGLES`:
  - Вершины: `v0, v1, v2, v2, v1, v3`
- Без индексов, `GL_TRIANGLE_STRIP`:
  - Вершины: `v0, v1, v2, v3`

Индексированный рендеринг: квадрат



- Без индексов, `GL_TRIANGLES`:
  - Вершины: `v0, v1, v2, v2, v1, v3`
- Без индексов, `GL_TRIANGLE_STRIP`:
  - Вершины: `v0, v1, v2, v3`
- С индексами, `GL_TRIANGLES`:
  - Вершины: `v0, v1, v2, v3`
  - Индексы: `0, 1, 2, 2, 1, 3`

Индексированный рендеринг

- Индексы хранятся в `GL_ELEMENT_ARRAY_BUFFER`
- id текущего `GL_ELEMENT_ARRAY_BUFFER` хранится в текущем VAO

Индексированный рендеринг

- Индексы хранятся в `GL_ELEMENT_ARRAY_BUFFER`
- id текущего `GL_ELEMENT_ARRAY_BUFFER` хранится в текущем VAO

```
glDrawElements(GLenum mode, GLsizei count,
    GLenum type, const GLvoid * indices)
```

Индексированный рендеринг

- Индексы хранятся в `GL_ELEMENT_ARRAY_BUFFER`
- id текущего `GL_ELEMENT_ARRAY_BUFFER` хранится в текущем VAO

```
glDrawElements(GLenum mode, GLsizei count,
    GLenum type, const GLvoid * indices)
```

- `mode` - как в `glDrawArrays`: `GL_TRIANGLES`, ...

Индексированный рендеринг

- Индексы хранятся в `GL_ELEMENT_ARRAY_BUFFER`
- id текущего `GL_ELEMENT_ARRAY_BUFFER` хранится в текущем VAO

```
glDrawElements(GLenum mode, GLsizei count,
    GLenum type, const GLvoid * indices)
```

- `mode` - как в `glDrawArrays`: `GL_TRIANGLES`, ...
- `count` - количество индексов

Индексированный рендеринг

- Индексы хранятся в `GL_ELEMENT_ARRAY_BUFFER`
- id текущего `GL_ELEMENT_ARRAY_BUFFER` хранится в текущем VAO

```
glDrawElements(GLenum mode, GLsizei count,
    GLenum type, const GLvoid * indices)
```

- `mode` - как в `glDrawArrays`: `GL_TRIANGLES`, ...
- `count` - количество индексов
- `type` - тип индексов: `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, `GL_UNSIGNED_INT`

Индексированный рендеринг

- Индексы хранятся в `GL_ELEMENT_ARRAY_BUFFER`
- id текущего `GL_ELEMENT_ARRAY_BUFFER` хранится в текущем VAO

```
glDrawElements(GLenum mode, GLsizei count,
    GLenum type, const GLvoid * indices)
```

- `mode` - как в `glDrawArrays`: `GL_TRIANGLES`, ...
- `count` - количество индексов
- `type` - тип индексов: `GL_UNSIGNED_BYTE`, `GL_UNSIGNED_SHORT`, `GL_UNSIGNED_INT`
- `indices` - смещение в текущем `GL_ELEMENT_ARRAY_BUFFER` до места, где начинаются индексы

Индексированный рендеринг: пример

```
// Инициализация:
program = createProgram()
vertices, indices = generateMesh()
vbo = createBuffer(vertices)
ebo = createBuffer(indices)
vao = createVAO()

glBindVertexArray(vao)
glBindBuffer(GL_ARRAY_BUFFER, vbo)
// настраиваем аттрибуты
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)

// Рендеринг:
glUseProgram(program)
glBindVertexArray(vao)
glDrawElements(GL_TRIANGLES, count,
    GL_UNSIGNED_SHORT, (void*)(0))
```

Primitive restart

- При использовании `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` все входные вершины образуют один line strip, line loop, triangle strip, triangle fan соответственно

Primitive restart

- При использовании `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` все входные вершины образуют один line strip, line loop, triangle strip, triangle fan соответственно
- Что, если мы хотим одну последовательность вершин разбить на несколько примитивов?

Primitive restart

- При использовании `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` все входные вершины образуют один line strip, line loop, triangle strip, triangle fan соответственно
- Что, если мы хотим одну последовательность вершин разбить на несколько примитивов?
- Можно сделать несколько вызовов `glDraw*`, но это медленно

Primitive restart

- При использовании `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` все входные вершины образуют один line strip, line loop, triangle strip, triangle fan соответственно
- Что, если мы хотим одну последовательность вершин разбить на несколько примитивов?
- Можно сделать несколько вызовов `glDraw*`, но это медленно
- Специальное значение индекса вершины означает что со следующей вершины нужно начать новый примитив:

  `GL_TRIANGLE_STRIP:`
  `[0, 1, 2, 3, 65535, 4, 5, 6, 7]`
  `[0, 1, 2, 3], [4, 5, 6, 7]`

Primitive restart

- При использовании `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` все входные вершины образуют один line strip, line loop, triangle strip, triangle fan соответственно
- Что, если мы хотим одну последовательность вершин разбить на несколько примитивов?
- Можно сделать несколько вызовов `glDraw*`, но это медленно
- Специальное значение индекса вершины означает что со следующей вершины нужно начать новый примитив:

  `GL_TRIANGLE_STRIP:`
  `[0, 1, 2, 3, 65535, 4, 5, 6, 7]`
  `[0, 1, 2, 3], [4, 5, 6, 7]`
- `glEnable(GL_PRIMITIVE_RESTART)`

Primitive restart

- При использовании `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_TRIANGLE_STRIP`, `GL_TRIANGLE_FAN` все входные вершины образуют один line strip, line loop, triangle strip, triangle fan соответственно
- Что, если мы хотим одну последовательность вершин разбить на несколько примитивов?
- Можно сделать несколько вызовов `glDraw*`, но это медленно
- Специальное значение индекса вершины означает что со следующей вершины нужно начать новый примитив:

  `GL_TRIANGLE_STRIP:`
  `[0, 1, 2, 3, 65535, 4, 5, 6, 7]`
  `[0, 1, 2, 3], [4, 5, 6, 7]`
- `glEnable(GL_PRIMITIVE_RESTART)`
- `glPrimitiveRestartIndex(index)`

Индексированный рендеринг: ссылки

- khronos.org/opengl/wiki/Vertex_Rendering#Basic_Drawing
- opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing

VAO, VBO, EBO: схема