

Компьютерная графика

Лекция 3: Объекты OpenGL, буферы, атрибуты 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 формат и хранение вершин

Объекты OpenGL

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

Объекты OpenGL

- ▶ Шейдеры, шейдерные программы – программируемая часть конвейера
- ▶ Буферы – хранят данные на GPU
- ▶ Vertex Array – описывают формат и хранение вершин
- ▶ Текстуры – изображения в памяти GPU, которые можно читать из шейдера, и в которые можно рисовать
- ▶ Renderbuffer – буферы, в которые можно рисовать
- ▶ Framebuffer – содержит настройки рисования в текстуры и renderbuffer'ы
- ▶ И другие

Создание/удаление объектов OpenGL

- ▶ В коде объект представляется идентификатором типа `GLuint`
 - ▶ Id уникален среди объектов одного типа (шейдеры, программы, ...)

Создание/удаление объектов OpenGL

- ▶ В коде 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

- ▶ В коде 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

- ▶ В коде 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 настраивать

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 настраивать
 - ▶ 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 настраивать
 - ▶ 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 (т.е. удалить нулевой объект – не ошибка)

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 настраивать
 - ▶ 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 (т.е. удалить нулевой объект – не ошибка)
- ▶ Нулевой id можно использовать как маркер отсутствия объекта

Удаление объектов

- ▶ По-хорошему, объекты нужно удалять, когда они перестают быть нужны (все движки и серьёзные обёртки над OpenGL это делают)

Удаление объектов

- ▶ По-хорошему, объекты нужно удалять, когда они перестают быть нужны (все движки и серьёзные обёртки над OpenGL это делают)
- ▶ На практике, они удаляются автоматически в конце работы программы (при удалении контекста OpenGL), и мы будем этим пользоваться в течение курса

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

Текущий объект

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

Текущий объект

- ▶ 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

Текущий объект

- ▶ 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` и др.) используют текущую шейдерную программу и текущий vertex array

Текущий объект

- ▶ `glBindBuffer(target, id)`
- ▶ `glBindTexture(target, id)`
- ▶ `glBindRenderbuffer(target, id)`
- ▶ `glBindFramebuffer(target, id)`

Текущий объект

- ▶ `glBindBuffer(target, id)`
- ▶ `glBindTexture(target, id)`
- ▶ `glBindRenderbuffer(target, id)`
- ▶ `glBindFramebuffer(target, id)`
- ▶ Что такое `target`?

Текущий объект: target

- ▶ Для каждого вида объектов (буферы, текстуры, ...) есть отдельный набор возможных значений target

Текущий объект: target

- ▶ Для каждого вида объектов (буферы, текстуры, ...) есть отдельный набор возможных значений target
- ▶ Для буферов, текстур, framebuffer'ов и renderbuffer'ов нет одного текущего объекта, но есть текущий объект для каждого конкретного target'a

Текущий объект: target

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

Текущий объект: target

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

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/IBO) – 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/IBO) – массив индексов вершин
 - ▶ `GL_UNIFORM_BUFFER` (UBO) – массив для uniform-блоков

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/IBO) – массив индексов вершин
 - ▶ `GL_UNIFORM_BUFFER` (UBO) – массив для uniform-блоков
 - ▶ ...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/IBO) – массив индексов вершин
 - ▶ `GL_UNIFORM_BUFFER` (UBO) – массив для uniform-блоков
 - ▶ ...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
 - ▶ ⇒ Драйвер, возможно, 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
- ▶ ⇒ 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 особый *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 особый *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 особый *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 особый *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

Mapped buffer

- ▶ Можно получить виртуальный указатель на буфер или его часть и пользоваться им для чтения/записи
- ▶ `glMapBuffer(target, access)` – возвращает особый *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
- ▶ `glUnmapBuffer` может вернуть `GL_FALSE`, если что-то пошло не так, и память буфера при этом остаётся непроинициализированной

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](https://www.khronos.org/opengl/wiki/Buffer_Object)
- ▶ songho.ca/opengl/gl_vbo.html

Атрибуты вершин

- ▶ С точки зрения OpenGL, вершины характеризуются набором *атрибутов*

Атрибуты вершин

- ▶ С точки зрения OpenGL, вершины характеризуются набором *атрибутов*
- ▶ Каждый атрибут имеет свой тип и размерность

Атрибуты вершин

- ▶ С точки зрения OpenGL, вершины характеризуются набором *атрибутов*
- ▶ Каждый атрибут имеет свой тип и размерность
- ▶ Все вершины в рамках одного вызова команды рисования (`glDrawArrays` и др.) имеют один набор атрибутов

Атрибуты вершин

- ▶ С точки зрения OpenGL, вершины характеризуются набором *атрибутов*
- ▶ Каждый атрибут имеет свой тип и размерность
- ▶ Все вершины в рамках одного вызова команды рисования (`glDrawArrays` и др.) имеют один набор атрибутов
- ▶ Все настройки атрибутов конкретного набора вершин хранятся в VAO

Атрибуты вершин: шейдер

```
#version 330 core
```

```
//          index      type  name
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() {
    ...
}
```

Атрибуты вершин

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

Атрибуты вершин

- ▶ Индекс: 0, 1, ... `glGet(GL_MAX_VERTEX_ATTRIBS)` - 1
 - ▶ `glGet(GL_MAX_VERTEX_ATTRIBS) ≥ 16`
- ▶ Включен/выключен
 - ▶ `glEnableVertexAttribArray(index)` – включить атрибут
 - ▶ Выключены по умолчанию
 - ▶ Состояние хранится в vertex array

Атрибуты вершин

- ▶ Индекс: 0, 1, ... `glGet(GL_MAX_VERTEX_ATTRIBS)` - 1
 - ▶ `glGet(GL_MAX_VERTEX_ATTRIBS) ≥ 16`
- ▶ Включен/выключен
 - ▶ `glEnableVertexAttribArray(index)` – включить атрибут
 - ▶ Выключены по умолчанию
 - ▶ Состояние хранится в vertex array
- ▶ Если у вершины есть атрибут, не соответствующий атрибуту в шейдере, он просто игнорируется

Атрибуты вершин

- ▶ Индекс: 0, 1, ... `glGet(GL_MAX_VERTEX_ATTRIBS)` - 1
 - ▶ `glGet(GL_MAX_VERTEX_ATTRIBS) ≥ 16`
- ▶ Включен/выключен
 - ▶ `glEnableVertexAttribArray(index)` – включить атрибут
 - ▶ Выключены по умолчанию
 - ▶ Состояние хранится в vertex array
- ▶ Если у вершины есть атрибут, не соответствующий атрибуту в шейдере, он просто игнорируется
- ▶ Если в шейдере есть атрибут, не соответствующий атрибуту у вершин, он принимает значение по умолчанию:
 - ▶ float: 0.0
 - ▶ vec2: (0.0, 0.0)
 - ▶ vec3: (0.0, 0.0, 0.0)
 - ▶ vec4: (0.0, 0.0, 0.0, 1.0)

Атрибуты вершин: хранение

Параметры хранения данных атрибута:

Атрибуты вершин: хранение

Параметры хранения данных атрибута:

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

Атрибуты вершин: хранение

Параметры хранения данных атрибута:

- ▶ 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

Атрибуты вершин: хранение

Параметры хранения данных атрибута:

- ▶ 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

Атрибуты вершин: хранение

Параметры хранения данных атрибута:

- ▶ 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 атрибута, соответствующими соседним вершинам

Атрибуты вершин: хранение

Параметры хранения данных атрибута:

- ▶ 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 атрибута, соответствующими соседним вершинам
- ▶ Всё это запоминается в vertex array

Атрибуты вершин: нормированность

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

Атрибуты вершин: нормированность

- ▶ Как интерпретировать значение атрибута, 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`

Атрибуты вершин: нормированность

- ▶ Как интерпретировать значение атрибута, 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`

Атрибуты вершин: нормированность

- ▶ Как интерпретировать значение атрибута, 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

Атрибуты вершин: нормированность

- ▶ Как интерпретировать значение атрибута, 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 атрибутов

Атрибуты вершин: нормированность

- ▶ Как интерпретировать значение атрибута, 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 атрибутов
- ▶ Имеет смысл только для floating-point атрибутов

Атрибуты вершин

- ▶ Перед настройкой конкретного атрибута нужно сделать текущим VAO, который вы хотите настроить

Атрибуты вершин

- ▶ Перед настройкой конкретного атрибута нужно сделать текущим VAO, который вы хотите настроить
- ▶ Если атрибут объявлен в шейдере как floating-point (`vec3` и т.п.), нужно использовать функцию `glVertexAttribPointer`, но храниться он *может* как целочисленный
- ▶ Если атрибут объявлен в шейдере как целочисленный (`ivec3`, `uvec4` и т.п.), нужно использовать функцию `glVertexAttribIPointer`, и храниться он *должен* как целочисленный

Атрибуты вершин: floating-point

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

Атрибуты вершин: floating-point

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

- ▶ `index` – индекс атрибута (`layout (location = index)` в шейдере)

Атрибуты вершин: floating-point

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

- ▶ index – индекс атрибута (layout (location = index) в шейдере)
- ▶ size – размерность (число компонент)

Атрибуты вершин: floating-point

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

- ▶ index – индекс атрибута (layout (location = index) в шейдере)
- ▶ size – размерность (число компонент)
- ▶ type – тип каждой компоненты

Атрибуты вершин: floating-point

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

- ▶ `index` – индекс атрибута (`layout (location = index)` в шейдере)
- ▶ `size` – размерность (число компонент)
- ▶ `type` – тип каждой компоненты
- ▶ `normalized` – нормированный или нет

Атрибуты вершин: floating-point

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

- ▶ `index` – индекс атрибута (`layout (location = index)` в шейдере)
- ▶ `size` – размерность (число компонент)
- ▶ `type` – тип каждой компоненты
- ▶ `normalized` – нормированный или нет
- ▶ `stride` – расстояние между соседними значениями (в байтах)

Атрибуты вершин: floating-point

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

- ▶ `index` – индекс атрибута (`layout (location = index)` в шейдере)
- ▶ `size` – размерность (число компонент)
- ▶ `type` – тип каждой компоненты
- ▶ `normalized` – нормированный или нет
- ▶ `stride` – расстояние между соседними значениями (в байтах)
- ▶ `pointer` – указатель на данные

Атрибуты вершин: floating-point

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

- ▶ `index` – индекс атрибута (`layout (location = index)` в шейдере)
- ▶ `size` – размерность (число компонент)
- ▶ `type` – тип каждой компоненты
- ▶ `normalized` – нормированный или нет
- ▶ `stride` – расстояние между соседними значениями (в байтах)
- ▶ `pointer` – указатель на данные
 - ▶ На самом деле, `pointer` – сдвиг в байтах относительно начала памяти текущего `GL_ARRAY_BUFFER`
 - ▶ Например, если данные этого атрибута начинаются на 12-ом байте текущего `GL_ARRAY_BUFFER`, нужно передать `(void *) (12)`

Атрибуты вершин: floating-point

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

- ▶ `index` – индекс атрибута (`layout (location = index)` в шейдере)
- ▶ `size` – размерность (число компонент)
- ▶ `type` – тип каждой компоненты
- ▶ `normalized` – нормированный или нет
- ▶ `stride` – расстояние между соседними значениями (в байтах)
- ▶ `pointer` – указатель на данные
 - ▶ На самом деле, `pointer` – сдвиг в байтах относительно начала памяти текущего `GL_ARRAY_BUFFER`
 - ▶ Например, если данные этого атрибута начинаются на 12-ом байте текущего `GL_ARRAY_BUFFER`, нужно передать `(void *) (12)`
 - ▶ Разные атрибуты могут лежать в разных буферах

Атрибуты 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

Атрибуты вершин: stride

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

Атрибуты вершин: stride

- ▶ Пусть `start` – смещение, указанное в параметре `pointer` функции `glVertexAttribPointer`
- ▶ Тогда соответствующий атрибут 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.

Атрибуты вершин: stride

- ▶ Пусть `start` – смещение, указанное в параметре `pointer` функции `glVertexAttribPointer`
- ▶ Тогда соответствующий атрибут 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 атрибута `stride = size * sizeof(type)`
 - ▶ Например, для `size = 3` и `type = GL_UNSIGNED_INT`, `stride` будет вычислен как
$$\text{stride} = 3 * \text{sizeof}(\text{unsigned int}) = 3 * 4 = 12$$

Атрибуты вершин: stride

- ▶ Пусть `start` – смещение, указанное в параметре `pointer` функции `glVertexAttribPointer`
- ▶ Тогда соответствующий атрибут 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 атрибута `stride = size * sizeof(type)`
 - ▶ Например, для `size = 3` и `type = GL_UNSIGNED_INT`, `stride` будет вычислен как
$$\text{stride} = 3 * \text{sizeof}(\text{unsigned int}) = 3 * 4 = 12$$
 - ▶ Можно использовать, если значения конкретного атрибута идут в памяти подряд, друг за другом, без пропусков

Атрибуты вершин: stride

- ▶ Пусть `start` – смещение, указанное в параметре `pointer` функции `glVertexAttribPointer`
- ▶ Тогда соответствующий атрибут 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 атрибута `stride = size * sizeof(type)`
 - ▶ Например, для `size = 3` и `type = GL_UNSIGNED_INT`, `stride` будет вычислен как
$$\text{stride} = 3 * \text{sizeof}(\text{unsigned int}) = 3 * 4 = 12$$
 - ▶ Можно использовать, если значения конкретного атрибута идут в памяти подряд, друг за другом, без пропусков
- ▶ Нужен для гибкости хранения атрибутов

Атрибуты вершин: размерность

- ▶ Размерность атрибута, указанная в `glVertexAttribPointer` (т.е. реальное количество хранимых в памяти компонент), не обязана совпадать с размерностью, указанной в шейдере

Атрибуты вершин: размерность

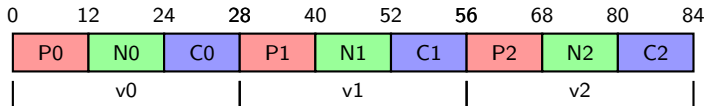
- ▶ Размерность атрибута, указанная в `glVertexAttribPointer` (т.е. реальное количество хранимых в памяти компонент), не обязана совпадать с размерностью, указанной в шейдере
- ▶ Если компонент больше, чем надо, они отбрасываются

Атрибуты вершин: размерность

- ▶ Размерность атрибута, указанная в `glVertexAttribPointer` (т.е. реальное количество хранимых в памяти компонент), не обязана совпадать с размерностью, указанной в шейдере
- ▶ Если компонент больше, чем надо, они отбрасываются
- ▶ Если компонент меньше, чем надо, они дополняются нулём (первые 3 компоненты) или единицей (4ая компонента)

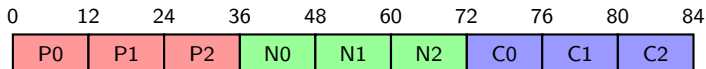
Атрибуты вершин: пример array-of-structs, один буфер

```
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));
```



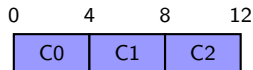
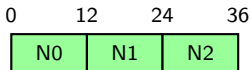
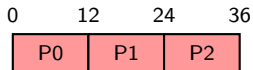
Аттрибуты вершин: пример struct-of-arrays, три буфера

```
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 атрибутов постоянны, а другую часть нужно иногда обновлять – разделите эти части по двум разным VBO

Аттрибуты вершин: пример полностью

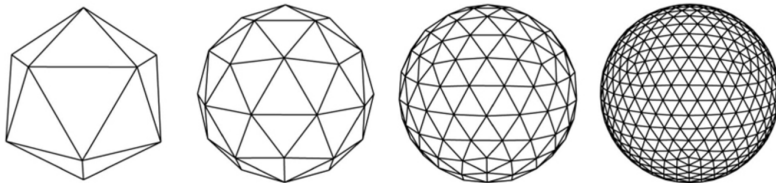
```
// Инициализация:  
program = createProgram()  
vertices = generateVertices()  
vbo = createVBO(vertices)  
vao = createVAO()  
setupAttributes(vao, vbo) // glVertexAttribPointer, ...  
  
// Рендеринг:  
glUseProgram(program)  
glBindVertexArray(vao)  
glDrawArrays(GL_TRIANGLES, 0, count)
```

Аттрибуты вершин: пример полностью

```
// Инициализация:  
program = createProgram()  
vertices = generateVertices()  
vbo = createVBO(vertices)  
vao = createVAO()  
setupAttributes(vao, vbo) // glVertexAttribPointer, ...  
  
// Рендеринг:  
glUseProgram(program)  
glBindVertexArray(vao)  
glDrawArrays(GL_TRIANGLES, 0, count)
```

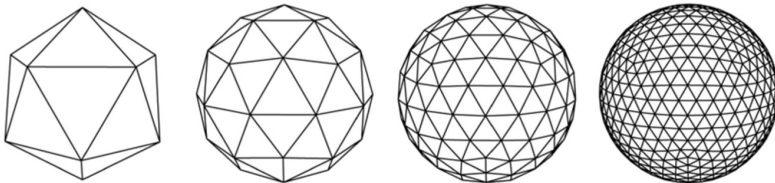
► [khronos.org/opengl/wiki/Vertex_Specification](https://www.khronos.org/opengl/wiki/Vertex_Specification)

Индексированный рендеринг



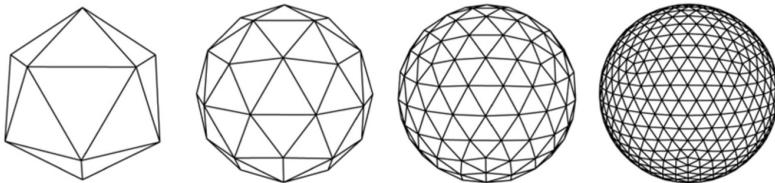
- ▶ В моделях одна вершина часто входит в несколько треугольников (в среднем, 6)

Индексированный рендеринг



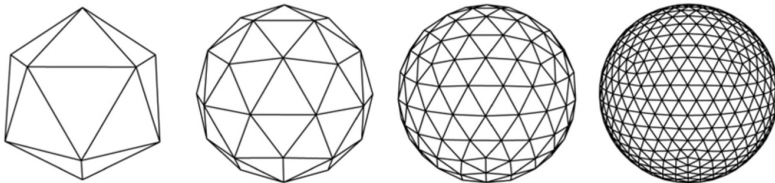
- ▶ В моделях одна вершина часто входит в несколько треугольников (в среднем, 6)
- ▶ Если использовать `GL_TRIANGLES`, придётся копировать каждую вершину несколько раз

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):`

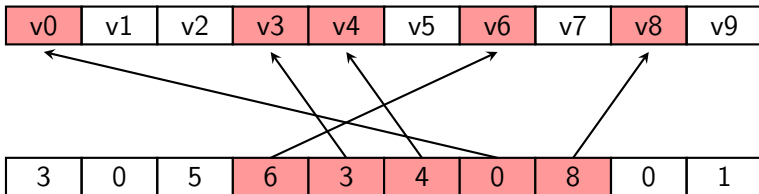
v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
----	----	----	----	----	----	----	----	----	----

Индексированный рендеринг

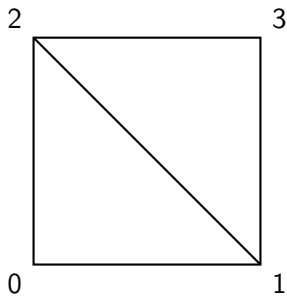
- ▶ `glDrawArrays(mode, 3, 5):`

v0	v1	v2	v3	v4	v5	v6	v7	v8	v9
----	----	----	----	----	----	----	----	----	----

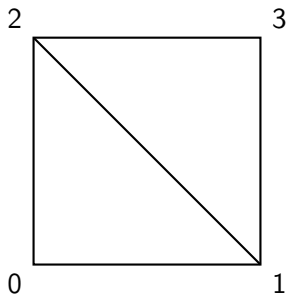
- ▶ `glDrawElements(mode, ...):`



Индексированный рендеринг: квадрат

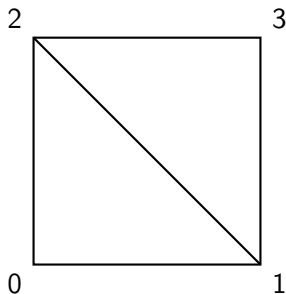


Индексированный рендеринг: квадрат



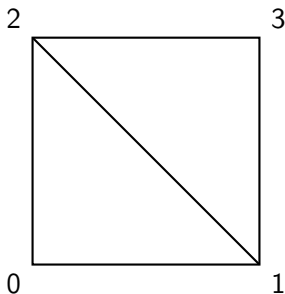
- ▶ Без индексов, `GL_TRIANGLES`:
 - ▶ Вершины: `v0`, `v1`, `v2`, `v2`, `v1`, `v3`

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 атрибуты...
// загружаем вершины в vbo...
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo)
// загружаем индексы в 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]` эквивалентно
`[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] эквивалентно
[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] эквивалентно
[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](https://www.khronos.org/opengl/wiki/Vertex_Rendering#Basic_Drawing)
- ▶ opengl-tutorial.org/intermediate-tutorials/tutorial-9-vbo-indexing

VAO, VBO, EBO: схема

VBO1:

P0	N0	P1	N1
----	----	----	----

 ...

VBO2:

C0	C1
----	----

 ...

EBO:

0	1	2	2	1	3
---	---	---	---	---	---

 ...

VAO:	<table border="1"><tr><td>Indices:</td><td>EBO</td></tr><tr><td>Attrib 0:</td><td>VBO1, offset, stride</td></tr><tr><td>Attrib 1:</td><td>VBO1, offset, stride</td></tr><tr><td>Attrib 2:</td><td>VBO2, offset, stride</td></tr></table>	Indices:	EBO	Attrib 0:	VBO1, offset, stride	Attrib 1:	VBO1, offset, stride	Attrib 2:	VBO2, offset, stride
Indices:	EBO								
Attrib 0:	VBO1, offset, stride								
Attrib 1:	VBO1, offset, stride								
Attrib 2:	VBO2, offset, stride								

Рендеринг нескольких объектов

- ▶ Обычно: для каждого объекта свой набор VAO+VBO(+EBO)
- ▶ Для обновления данных нужны только VBO (и, возможно, EBO)
- ▶ Для рисования нужен только VAO

Рендеринг нескольких объектов

Инициализация:

```
for (o in objects) {  
    glGenVertexArrays(1, &o.vao)  
    glGenBuffers(1, &o.vbo)  
    glGenBuffers(1, &o.ebo)  
  
    glBindVertexArray(o.vao)  
    glBindBuffer(GL_ARRAY_BUFFER, o.vbo)  
    for (i in attribs) {  
        glEnableVertexAttribArray(i)  
        glVertexAttribPointer(i, ...)  
    }  
  
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,  
        o.ebo) // хранится в o.vao  
}
```


Рендеринг нескольких объектов

Загрузка данных:

```
for (o in objects) {  
    glBindBuffer(GL_ARRAY_BUFFER, o.vbo)  
    glBufferData(GL_ARRAY_BUFFER, o.vertices)  
  
    glBindBuffer(GL_ARRAY_BUFFER, o.ebo)  
    glBufferData(GL_ARRAY_BUFFER, o.indices)  
}
```

Рендеринг нескольких объектов

Рендеринг:

```
glUseProgram(program)
glUniform(...)
for (o in objects) {
    glBindVertexArray(o.vao)
    glDrawElements(o.indexCount)
}
```