

# Компьютерная графика

Лекция 14: оптимизация рендеринга, timer queries, batching, instancing, uniform buffers, frustum culling, occlusion culling

2022

# Оптимизация – это сложно

На производительность CPU влияют:

# Оптимизация – это сложно

На производительность CPU влияют:

- ▶ Общая загруженность системы

# Оптимизация – это сложно

На производительность CPU влияют:

- ▶ Общая загрузка системы
- ▶ Количество и паттерн доступов к памяти (cache-friendliness)

# Оптимизация – это сложно

На производительность CPU влияют:

- ▶ Общая загруженность системы
- ▶ Количество и паттерн доступов к памяти (cache-friendliness)
- ▶ Как функции программы лежат в памяти (опять кэш)

# Оптимизация – это сложно

На производительность CPU влияют:

- ▶ Общая загруженность системы
- ▶ Количество и паттерн доступов к памяти (cache-friendliness)
- ▶ Как функции программы лежат в памяти (опять кэш)
- ▶ Branch prediction

# Оптимизация – это сложно

На производительность CPU влияют:

- ▶ Общая загруженность системы
- ▶ Количество и паттерн доступов к памяти (cache-friendliness)
- ▶ Как функции программы лежат в памяти (опять кэш)
- ▶ Branch prediction
- ▶ Оптимизации компилятора

# Оптимизация – это сложно

На производительность CPU влияют:

- ▶ Общая загруженность системы
- ▶ Количество и паттерн доступов к памяти (cache-friendliness)
- ▶ Как функции программы лежат в памяти (опять кэш)
- ▶ Branch prediction
- ▶ Оптимизации компилятора
- ▶ Throttling



# Оптимизация – это сложно

На производительность CPU влияют:

- ▶ Общая загруженность системы
- ▶ Количество и паттерн доступов к памяти (cache-friendliness)
- ▶ Как функции программы лежат в памяти (опять кэш)
- ▶ Branch prediction
- ▶ Оптимизации компилятора
- ▶ Throttling
- ▶ Многое другое

# Оптимизация на GPU – это очень сложно

На производительность GPU влияют:

# Оптимизация на GPU – это очень сложно

На производительность GPU влияют:

- ▶ Асинхронность: операции выполняются отложено

# Оптимизация на GPU – это очень сложно

На производительность GPU влияют:

- ▶ Асинхронность: операции выполняются отложено
- ▶ Параллельность: много операций выполняется одновременно

# Оптимизация на GPU – это очень сложно

На производительность GPU влияют:

- ▶ Асинхронность: операции выполняются отложено
- ▶ Параллельность: много операций выполняется одновременно
- ▶ Много встроенных операций (fixed-function pipeline)

# Оптимизация на GPU – это очень сложно

На производительность GPU влияют:

- ▶ Асинхронность: операции выполняются отложено
- ▶ Параллельность: много операций выполняется одновременно
- ▶ Много встроенных операций (fixed-function pipeline)
- ▶ Сложные операции с памятью (доступ к текстуре: mipmaps + фильтрация, и т.п.)

# Оптимизация на GPU – это очень сложно

На производительность GPU влияют:

- ▶ Асинхронность: операции выполняются отложено
- ▶ Параллельность: много операций выполняется одновременно
- ▶ Много встроенных операций (fixed-function pipeline)
- ▶ Сложные операции с памятью (доступ к текстуре: mipmaps + фильтрация, и т.п.)
- ▶ Оптимизации компилятора шейдеров (зависят от драйвера)

# Оптимизация на GPU – это очень сложно

На производительность GPU влияют:

- ▶ Асинхронность: операции выполняются отложено
- ▶ Параллельность: много операций выполняется одновременно
- ▶ Много встроенных операций (fixed-function pipeline)
- ▶ Сложные операции с памятью (доступ к текстуре: mipmaps + фильтрация, и т.п.)
- ▶ Оптимизации компилятора шейдеров (зависят от драйвера)
- ▶ Многое другое



# Два основных принципа оптимизации

- ▶ Замеряйте то, что оптимизируете

# Два основных принципа оптимизации

- ▶ Замеряйте то, что оптимизируете
  - ▶ N.B Измерение 'на глаз' – тоже измерение

# Два основных принципа оптимизации

- ▶ Замеряйте то, что оптимизируете
  - ▶ N.B Измерение 'на глаз' – тоже измерение
- ▶ Выполняйте меньше операций (любых!)

## Измерение времени работы – неправильный способ

```
while (true) {  
    auto frame_start = clock::now();  
  
    // нарисовали сцену  
    ...  
  
    auto frame_end = clock::now();  
  
    SwapBuffers();  
}
```

## Измерение времени работы – неправильный способ

```
while (true) {  
    auto frame_start = clock::now();  
  
    // нарисовали сцену  
    ...  
  
    auto frame_end = clock::now();  
  
    SwapBuffers();  
}
```

- ▶  $(\text{frame\_end} - \text{frame\_start})$  – сколько времени ушло на то, чтобы **вызвать OpenGL-команды**

## Измерение времени работы – неправильный способ

```
while (true) {  
    auto frame_start = clock::now();  
  
    // нарисовали сцену  
    ...  
  
    auto frame_end = clock::now();  
  
    SwapBuffers();  
}
```

- ▶  $(\text{frame\_end} - \text{frame\_start})$  – сколько времени ушло на то, чтобы **вызвать OpenGL-команды**
- ▶ В реальности драйвер поставил их в очередь, и, возможно, GPU ещё даже не начала их выполнять

## Измерение времени работы – простой способ

```
disableVsync();  
auto last_frame_start = clock::now();  
while (true) {  
    auto frame_start = clock::now();  
    auto frame_time = frame_start - last_frame_start;  
    last_frame_start = frame_start;  
  
    // нарисовали сцену  
    ...  
  
    SwapBuffers();  
}
```

## Измерение времени работы – простой способ

```
disableVsync();  
auto last_frame_start = clock::now();  
while (true) {  
    auto frame_start = clock::now();  
    auto frame_time = frame_start - last_frame_start;  
    last_frame_start = frame_start;  
  
    // нарисовали сцену  
    ...  
  
    SwapBuffers();  
}
```

- ▶ Из-за выключенного vsync видеокарта будет работать ± постоянно



## Измерение времени работы – простой способ

```
disableVsync();  
auto last_frame_start = clock::now();  
while (true) {  
    auto frame_start = clock::now();  
    auto frame_time = frame_start - last_frame_start;  
    last_frame_start = frame_start;  
  
    // нарисовали сцену  
    ...  
  
    SwapBuffers();  
}
```

- ▶ Из-за выключенного vsync видеокарта будет работать  $\pm$  постоянно
- ▶ В итоге мы получим примерное время, тратящееся на рисование одного кадра

## Измерение времени работы: `glFlush` и `glFinish`

- ▶ Многие (старые) tutorиалы по измерению времени кадра советуют вызывать `glFlush` или `glFinish` в конце кадра

## Измерение времени работы: glFlush и glFinish

- ▶ Многие (старые) tutorиалы по измерению времени кадра советуют вызывать `glFlush` или `glFinish` в конце кадра
- ▶ `glFlush` сбрасывает буфер команд (хранящийся внутри драйвера) с CPU на GPU

## Измерение времени работы: `glFlush` и `glFinish`

- ▶ Многие (старые) tutorиалы по измерению времени кадра советуют вызывать `glFlush` или `glFinish` в конце кадра
- ▶ `glFlush` сбрасывает буфер команд (хранящийся внутри драйвера) с CPU на GPU
- ▶ `glFinish` ждёт, пока GPU не завершит обрабатывать все посланные команды

## Измерение времени работы: `glFlush` и `glFinish`

- ▶ Многие (старые) tutorиалы по измерению времени кадра советуют вызывать `glFlush` или `glFinish` в конце кадра
- ▶ `glFlush` сбрасывает буфер команд (хранящийся внутри драйвера) с CPU на GPU
- ▶ `glFinish` ждёт, пока GPU не завершит обрабатывать все посланные команды
- ▶ `SwapBuffers` сама вызывает `glFlush`

## Измерение времени работы: glFlush и glFinish

- ▶ Многие (старые) tutorиалы по измерению времени кадра советуют вызывать glFlush или glFinish в конце кадра
- ▶ glFlush сбрасывает буфер команд (хранящийся внутри драйвера) с CPU на GPU
- ▶ glFinish ждёт, пока GPU не завершит обрабатывать все посланные команды
- ▶ SwapBuffers сама вызывает glFlush
- ▶ glFinish ухудшает производительность: половину времени вы отправляете команды на GPU, а GPU (скорее всего) ничего не делает; половину времени вы ждёте, пока GPU закончит выполнять команды

## Измерение времени работы: glFlush и glFinish

- ▶ Многие (старые) tutorиалы по измерению времени кадра советуют вызывать glFlush или glFinish в конце кадра
- ▶ glFlush сбрасывает буфер команд (хранящийся внутри драйвера) с CPU на GPU
- ▶ glFinish ждёт, пока GPU не завершит обрабатывать все посланные команды
- ▶ SwapBuffers сама вызывает glFlush
- ▶ glFinish ухудшает производительность: половину времени вы отправляете команды на GPU, а GPU (скорее всего) ничего не делает; половину времени вы ждёте, пока GPU закончит выполнять команды
- ▶ ⇒ Лучше не пользоваться этими функциями :)

# Измерение времени работы: FPS vs frame duration

- ▶ FPS (frames per second, количество кадров в секунду) – очень неудобная метрика



# Измерение времени работы: FPS vs frame duration

- ▶ FPS (frames per second, количество кадров в секунду) – очень неудобная метрика
  - ▶ Нелинейна: если кадр рисовался 10 мс, и мы добавили что-то рисующееся 1 мс, и ещё что-то рисующееся 1 мс, то FPS изменялся от 100 до 90.9 до 83.3

# Измерение времени работы: FPS vs frame duration

- ▶ FPS (frames per second, количество кадров в секунду) – очень неудобная метрика
  - ▶ Нелинейна: если кадр рисовался 10 мс, и мы добавили что-то рисующееся 1 мс, и ещё что-то рисующееся 1 мс, то FPS изменялся от 100 до 90.9 до 83.3
- ▶ Обычно при измерениях используют *время* (а не  $1/\text{время}$ ), тратящееся на рисование кадра или конкретного объекта/эффекта (миллисекунды/микросекунды)

# Измерение времени работы – правильный способ: timer queries

- ▶ Query objects – объекты OpenGL, позволяющие узнать некоторую статистику с GPU:

# Измерение времени работы – правильный способ: timer queries

- ▶ Query objects – объекты OpenGL, позволяющие узнать некоторую статистику с GPU:
  - ▶ Сколько было нарисовано пикселей

# Измерение времени работы – правильный способ: timer queries

- ▶ Query objects – объекты OpenGL, позволяющие узнать некоторую статистику с GPU:
  - ▶ Сколько было нарисовано пикселей
  - ▶ Сколько сгенерировано примитивов (напр. геометрическим шейдером)

# Измерение времени работы – правильный способ: timer queries

- ▶ Query objects – объекты OpenGL, позволяющие узнать некоторую статистику с GPU:
  - ▶ Сколько было нарисовано пикселей
  - ▶ Сколько сгенерировано примитивов (напр. геометрическим шейдером)
  - ▶ Сколько прошло времени

# Измерение времени работы – правильный способ: timer queries

- ▶ Query objects – объекты OpenGL, позволяющие узнать некоторую статистику с GPU:
  - ▶ Сколько было нарисовано пикселей
  - ▶ Сколько сгенерировано примитивов (напр. геометрическим шейдером)
  - ▶ Сколько прошло времени
- ▶ `glGenQueries/glDeleteQueries`

# Измерение времени работы – правильный способ: timer queries

- ▶ Query objects – объекты OpenGL, позволяющие узнать некоторую статистику с GPU:
  - ▶ Сколько было нарисовано пикселей
  - ▶ Сколько сгенерировано примитивов (напр. геометрическим шейдером)
  - ▶ Сколько прошло времени
- ▶ `glGenQueries/glDeleteQueries`
- ▶ **Нет** `glBindQuery!`



## Измерение времени работы – правильный способ: timer queries

- ▶ `glBeginQuery/glEndQuery` – статистика будет собрана для команд между этими вызовами

## Измерение времени работы – правильный способ: timer queries

- ▶ `glBeginQuery/glEndQuery` – статистика будет собрана для команд между этими вызовами
- ▶ Query одного типа **не могут** быть вложенными

## Измерение времени работы – правильный способ: timer queries

- ▶ glBeginQuery/glEndQuery – статистика будет собрана для команд между этими вызовами
- ▶ Query одного типа **не могут** быть вложенными

```
GLuint query_id;  
glGenQueries(1, &query_id);  
  
...  
  
glBeginQuery(GL_TIME_ELAPSED, query_id);  
  
// что-нибудь рисуем  
  
glEndQuery(GL_TIME_ELAPSED);
```

## Измерение времени работы – правильный способ: timer queries

- ▶ GPU работает асинхронно  $\Rightarrow$  результат query будет готов не сразу

## Измерение времени работы – правильный способ: timer queries

- ▶ GPU работает асинхронно  $\Rightarrow$  результат query будет готов не сразу
- ▶ Узнать, готов ли результат:

```
glGetQueryObjectiv(query_id,  
    GL_QUERY_RESULT_AVAILABLE, &result);
```

## Измерение времени работы – правильный способ: timer queries

- ▶ GPU работает асинхронно  $\Rightarrow$  результат query будет готов не сразу

- ▶ Узнать, готов ли результат:

```
glGetQueryObjectiv(query_id,  
GL_QUERY_RESULT_AVAILABLE, &result);
```

- ▶ Получить результат (блокирует поток, если результат ещё не готов; неявно вызывает glFlush)

```
glGetQueryObjectiv(query_id,  
GL_QUERY_RESULT, &result);
```

## Измерение времени работы – правильный способ: timer queries

- ▶ GPU работает асинхронно  $\Rightarrow$  результат query будет готов не сразу

- ▶ Узнать, готов ли результат:

```
glGetQueryObjectiv(query_id,  
GL_QUERY_RESULT_AVAILABLE, &result);
```

- ▶ Получить результат (блокирует поток, если результат ещё не готов; неявно вызывает glFlush)

```
glGetQueryObjectiv(query_id,  
GL_QUERY_RESULT, &result);
```

- ▶ Время возвращается в **наносекундах**, т.е. знаковый 32-битный тип может представить 2 секунды

## Измерение времени работы – правильный способ: timer queries

- ▶ GPU работает асинхронно  $\Rightarrow$  результат query будет готов не сразу

- ▶ Узнать, готов ли результат:

```
glGetQueryObjectiv(query_id,  
GL_QUERY_RESULT_AVAILABLE, &result);
```

- ▶ Получить результат (блокирует поток, если результат ещё не готов; неявно вызывает glFlush)

```
glGetQueryObjectiv(query_id,  
GL_QUERY_RESULT, &result);
```

- ▶ Время возвращается в **наносекундах**, т.е. знаковый 32-битный тип может представить 2 секунды
- ▶ Если 64-битные и беззнаковые версии этих функций



## Измерение времени работы – правильный способ: пул timer queries

- ▶ Хотим мерять время рисования каждого кадра, но результат для предыдущего кадра может быть не готов к началу следующего кадра

## Измерение времени работы – правильный способ: пул timer queries

- ▶ Хотим мерять время рисования каждого кадра, но результат для предыдущего кадра может быть не готов к началу следующего кадра
- ▶  $\Rightarrow$  Заводим пул (pool) query-объектов:

## Измерение времени работы – правильный способ: пул timer queries

- ▶ Хотим мерять время рисования каждого кадра, но результат для предыдущего кадра может быть не готов к началу следующего кадра
- ▶  $\Rightarrow$  Заводим пул (pool) query-объектов:
  - ▶ Храним расширяемый массив (`std::vector`) query-объектов: ID + пометка, свободен или нет

## Измерение времени работы – правильный способ: пул timer queries

- ▶ Хотим мерять время рисования каждого кадра, но результат для предыдущего кадра может быть не готов к началу следующего кадра
- ▶  $\Rightarrow$  Заводим пул (pool) query-объектов:
  - ▶ Храним расширяемый массив (`std::vector`) query-объектов: ID + пометка, свободен или нет
  - ▶ Когда нам нужен новый query, ищем в массиве свободный объект, если такого нет – добавляем новый

## Измерение времени работы – правильный способ: пул timer queries

- ▶ Хотим мерять время рисования каждого кадра, но результат для предыдущего кадра может быть не готов к началу следующего кадра
- ▶  $\Rightarrow$  Заводим пул (pool) query-объектов:
  - ▶ Храним расширяемый массив (`std::vector`) query-объектов: ID + пометка, свободен или нет
  - ▶ Когда нам нужен новый query, ищем в массиве свободный объект, если такого нет – добавляем новый
  - ▶ В конце рисования кадра проходим по всем несвободным объектам и проверяем: если результат уже готов, обрабатываем его и помечаем объект свободным

## Измерение времени работы – правильный способ: пул timer queries

- ▶ Хотим мерять время рисования каждого кадра, но результат для предыдущего кадра может быть не готов к началу следующего кадра
- ▶  $\Rightarrow$  Заводим пул (pool) query-объектов:
  - ▶ Храним расширяемый массив (`std::vector`) query-объектов: ID + пометка, свободен или нет
  - ▶ Когда нам нужен новый query, ищем в массиве свободный объект, если такого нет – добавляем новый
  - ▶ В конце рисования кадра проходим по всем несвободным объектам и проверяем: если результат уже готов, обрабатываем его и помечаем объект свободным
- ▶ Средний размер пула – на сколько кадров отстаёт GPU от CPU

## Timer queries: ссылки

- ▶ [khronos.org/opengl/wiki/Query\\_Object](https://www.khronos.org/opengl/wiki/Query_Object)
- ▶ Тьюториал по использованию timer queries

# Поиск bottleneck'a

- ▶ Мы померяли и узнали, что тормозит



# Поиск bottleneck'a

- ▶ Мы померяли и узнали, что тормозит
- ▶ OpenGL pipeline включает много компонентов, какой именно тормозит?

# Поиск bottleneck'a

- ▶ Мы померяли и узнали, что тормозит
- ▶ OpenGL pipeline включает много компонентов, какой именно тормозит?
- ▶ Обычно компоненты конвейера влияют на следующие за ними компоненты

# Поиск bottleneck'а

- ▶ Мы померяли и узнали, что тормозит
- ▶ OpenGL pipeline включает много компонентов, какой именно тормозит?
- ▶ Обычно компоненты конвейера влияют на следующие за ними компоненты
  - ▶ Больше вершин  $\Rightarrow$  больше вызовов вершинного шейдера
  - ▶ Больше примитивов  $\Rightarrow$  больше пикселей
  - ▶ Больше пикселей  $\Rightarrow$  больше вызовов фрагментного шейдера
  - ▶ Больше пикселей  $\Rightarrow$  больше операций записи в память

# Поиск bottleneck'а

- ▶ Мы померяли и узнали, что тормозит
- ▶ OpenGL pipeline включает много компонентов, какой именно тормозит?
- ▶ Обычно компоненты конвейера влияют на следующие за ними компоненты
  - ▶ Больше вершин  $\Rightarrow$  больше вызовов вершинного шейдера
  - ▶ Больше примитивов  $\Rightarrow$  больше пикселей
  - ▶ Больше пикселей  $\Rightarrow$  больше вызовов фрагментного шейдера
  - ▶ Больше пикселей  $\Rightarrow$  больше операций записи в память
- ▶  $\Rightarrow$  Удобно искать bottleneck начиная с *конца* конвейера

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)

## Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый фрагментный шейдер

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей
  - ▶ Стало лучше?  $\Rightarrow$  Слишком много растеризации и операций записи в память



# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей
  - ▶ Стало лучше?  $\Rightarrow$  Слишком много растеризации и операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей
  - ▶ Стало лучше?  $\Rightarrow$  Слишком много растеризации и операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый вершинный шейдер

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей
  - ▶ Стало лучше?  $\Rightarrow$  Слишком много растеризации и операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый вершинный шейдер
- ▶ Уменьшим число вершин (параметр `count` в `glDraw*`)

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей
  - ▶ Стало лучше? ⇒ Слишком много растеризации и операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый вершинный шейдер
- ▶ Уменьшим число вершин (параметр count в `glDraw*`)
  - ▶ Стало лучше? ⇒ Слишком много вершин

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей
  - ▶ Стало лучше? ⇒ Слишком много растеризации и операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый вершинный шейдер
- ▶ Уменьшим число вершин (параметр count в `glDraw*`)
  - ▶ Стало лучше? ⇒ Слишком много вершин
- ▶ Ничего не помогло ⇒ CPU-bound

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей
  - ▶ Стало лучше? ⇒ Слишком много растеризации и операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый вершинный шейдер
- ▶ Уменьшим число вершин (параметр count в `glDraw*`)
  - ▶ Стало лучше? ⇒ Слишком много вершин
- ▶ Ничего не помогло ⇒ CPU-bound
  - ▶ Слишком много OpenGL-вызовов

# Поиск bottleneck'a

- ▶ Упростим до предела фрагментный шейдер (напр. выведем фиксированный цвет)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый фрагментный шейдер
- ▶ Уменьшим размер окна до чего-нибудь в духе 50x50 пикселей
  - ▶ Стало лучше? ⇒ Слишком много растеризации и операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый вершинный шейдер
- ▶ Уменьшим число вершин (параметр count в `glDraw*`)
  - ▶ Стало лучше? ⇒ Слишком много вершин
- ▶ Ничего не помогло ⇒ CPU-bound
  - ▶ Слишком много OpenGL-вызовов
  - ▶ Слишком много других операций на CPU

# Оптимизация шейдеров

- ▶ Выполняем меньше операций



# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем (по возможности) вызова сложных функций (`sin`, `exp`, `pow`)

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем (по возможности) вызова сложных функций (`sin`, `exp`, `pow`)
- ▶ Реорганизуем вычисления (напр. `exp(a+b+c+d)` вместо `exp(a)*exp(b)*exp(c)*exp(d)`)

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем (по возможности) вызова сложных функций ( $\sin$ ,  $\exp$ ,  $\log$ )
- ▶ Реорганизуем вычисления (напр.  $\exp(a+b+c+d)$  вместо  $\exp(a) * \exp(b) * \exp(c) * \exp(d)$ )
- ▶ Предпочитаем что-нибудь (в константный массив в шейдере или в текстуру)

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем (по возможности) вызова сложных функций ( $\sin$ ,  $\exp$ ,  $\log$ )
- ▶ Реорганизуем вычисления (напр.  $\exp(a+b+c+d)$  вместо  $\exp(a) * \exp(b) * \exp(c) * \exp(d)$ )
- ▶ Предпочитаем что-нибудь (в константный массив в шейдере или в текстуру)
- ▶ Меньше читаем из текстур

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем (по возможности) вызова сложных функций (`sin`, `exp`, `pow`)
- ▶ Реорганизуем вычисления (напр. `exp(a+b+c+d)` вместо `exp(a)*exp(b)*exp(c)*exp(d)`)
- ▶ Предпочитаем что-нибудь (в константный массив в шейдере или в текстуру)
- ▶ Меньше читаем из текстур
- ▶ Читаем из текстур меньшего размера (лучше утилизируется текстурный кэш)

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем (по возможности) вызова сложных функций ( $\sin$ ,  $\exp$ ,  $\log$ )
- ▶ Реорганизуем вычисления (напр.  $\exp(a+b+c+d)$  вместо  $\exp(a) * \exp(b) * \exp(c) * \exp(d)$ )
- ▶ Предпочитаем что-нибудь (в константный массив в шейдере или в текстуру)
- ▶ Меньше читаем из текстур
- ▶ Читаем из текстур меньшего размера (лучше утилизируется текстурный кэш)
- ▶ Используем  $\text{mipmap}$ 'ы (лучше утилизируется текстурный кэш)

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем (по возможности) вызова сложных функций ( $\sin$ ,  $\exp$ ,  $\log$ )
- ▶ Реорганизуем вычисления (напр.  $\exp(a+b+c+d)$  вместо  $\exp(a) * \exp(b) * \exp(c) * \exp(d)$ )
- ▶ Предпочитаем что-нибудь (в константный массив в шейдере или в текстуру)
- ▶ Меньше читаем из текстур
- ▶ Читаем из текстур меньшего размера (лучше утилизируется текстурный кэш)
- ▶ Используем mipmaps (лучше утилизируется текстурный кэш)
- ▶ Делаем так, чтобы близкие пиксели читали близкие части текстуры (cache coherency; лучше утилизируется текстурный кэш)

# Оптимизация вершин

- ▶ Используем индексированный рендеринг (меньше данных нужно прочитать из памяти; лучше используется вершинный кэш)



# Оптимизация вершин

- ▶ Используем индексированный рендеринг (меньше данных нужно прочесть из памяти; лучше используется вершинный кэш)
- ▶ Используем примитивы, группирующие вершины – line strip, triangle strip, triangle fan, etc (те же причины)

# Оптимизация вершин

- ▶ Используем индексированный рендеринг (меньше данных нужно прочитать из памяти; лучше используется вершинный кэш)
- ▶ Используем примитивы, группирующие вершины – line strip, triangle strip, triangle fan, etc (те же причины)
- ▶ Используем меньше атрибутов, не храним ненужные атрибуты, сжимаем атрибуты (напр. GL\_BYTE для нормалей, GL\_UNSIGNED\_BYTE для текстурных координат)

# Оптимизация вершин

- ▶ Используем индексированный рендеринг (меньше данных нужно прочитать из памяти; лучше используется вершинный кэш)
- ▶ Используем примитивы, группирующие вершины – line strip, triangle strip, triangle fan, etc (те же причины)
- ▶ Используем меньше атрибутов, не храним ненужные атрибуты, сжимаем атрибуты (напр. GL\_BYTE для нормалей, GL\_UNSIGNED\_BYTE для текстурных координат)
- ▶ Используем LOD (Level Of Detail) – упрощённые версии 3D моделей

# Оптимизация количества OpenGL-вызовов

- ▶ **Batching:** группируем объекты по используемому шейдеру, текстуре, другим настройкам (меньше переключения состояния OpenGL  $\Rightarrow$  меньше OpenGL-вызовов)

# Оптимизация количества OpenGL-вызовов

- ▶ **Batching:** группируем объекты по используемому шейдеру, текстуре, другим настройкам (меньше переключения состояния OpenGL  $\Rightarrow$  меньше OpenGL-вызовов)
- ▶ **Instancing:** рисуем много объектов одним OpenGL-вызовом

# Оптимизация количества OpenGL-вызовов

- ▶ **Batching:** группируем объекты по используемому шейдеру, текстуре, другим настройкам (меньше переключения состояния OpenGL  $\Rightarrow$  меньше OpenGL-вызовов)
- ▶ **Instancing:** рисуем много объектов одним OpenGL-вызовом
- ▶ **Uniform buffers:** передаём uniform-переменные не по одной, а записываем их в буффер (вместо большого количества вызовов `glUniform*` один вызов `glBufferData`)

# Оптимизация количества OpenGL-вызовов

- ▶ **Batching:** группируем объекты по используемому шейдеру, текстуре, другим настройкам (меньше переключения состояния OpenGL  $\Rightarrow$  меньше OpenGL-вызовов)
- ▶ **Instancing:** рисуем много объектов одним OpenGL-вызовом
- ▶ **Uniform buffers:** передаём uniform-переменные не по одной, а записываем их в буффер (вместо большого количества вызовов `glUniform*` один вызов `glBufferData`)
- ▶ **Indirect rendering:** переносим вычисления того, что нужно нарисовать, на GPU (OpenGL 4.0 + compute shaders)

# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру



# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру
  - ▶ Occlusion culling: не рисуем то, что закрыто другими объектами

# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру
  - ▶ Occlusion culling: не рисуем то, что закрыто другими объектами
- ▶ Рисуем с уменьшенным разрешением
  - ▶ Обычно используется для эффектов (blur, SSAO и т.п.)

# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру
  - ▶ Occlusion culling: не рисуем то, что закрыто другими объектами
- ▶ Рисуем с уменьшенным разрешением
  - ▶ Обычно используется для эффектов (blur, SSAO и т.п.)
- ▶ Переводим рисование в отдельный поток
  - ▶ Освобождает основной (UI) поток

# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру
  - ▶ Occlusion culling: не рисуем то, что закрыто другими объектами
- ▶ Рисуем с уменьшенным разрешением
  - ▶ Обычно используется для эффектов (blur, SSAO и т.п.)
- ▶ Переводим рисование в отдельный поток
  - ▶ Освобождает основной (UI) поток
  - ▶ Позволяет делать полезную работу, пока render-поток ждёт VSync

# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру
  - ▶ Occlusion culling: не рисуем то, что закрыто другими объектами
- ▶ Рисуем с уменьшенным разрешением
  - ▶ Обычно используется для эффектов (blur, SSAO и т.п.)
- ▶ Переводим рисование в отдельный поток
  - ▶ Освобождает основной (UI) поток
  - ▶ Позволяет делать полезную работу, пока render-поток ждёт VSync
  - ▶ Сильно усложняет код

# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру
  - ▶ Occlusion culling: не рисуем то, что закрыто другими объектами
- ▶ Рисуем с уменьшенным разрешением
  - ▶ Обычно используется для эффектов (blur, SSAO и т.п.)
- ▶ Переводим рисование в отдельный поток
  - ▶ Освобождает основной (UI) поток
  - ▶ Позволяет делать полезную работу, пока render-поток ждёт VSync
  - ▶ Сильно усложняет код
  - ▶ Все OpenGL-вызовы нужно делать из render-потока

# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру
  - ▶ Occlusion culling: не рисуем то, что закрыто другими объектами
- ▶ Рисуем с уменьшенным разрешением
  - ▶ Обычно используется для эффектов (blur, SSAO и т.п.)
- ▶ Переводим рисование в отдельный поток
  - ▶ Освобождает основной (UI) поток
  - ▶ Позволяет делать полезную работу, пока render-поток ждёт VSync
  - ▶ Сильно усложняет код
  - ▶ Все OpenGL-вызовы нужно делать из render-потока
  - ▶ Применяется только в крайних случаях

## Оптимизация: ссылки

- ▶ [khronos.org/opengl/wiki/Performance](http://khronos.org/opengl/wiki/Performance)
- ▶ [opengl.org/pipeline/article/vol003\\_8](http://opengl.org/pipeline/article/vol003_8)
- ▶ Доклад с GDC 2003, всё ещё актуальный



# LOD (level of detail)

- ▶ Когда модель далеко от камеры, рисуем упрощённую модель объекта вместо детализированной

# LOD (level of detail)

- ▶ Когда модель далеко от камеры, рисуем упрощённую модель объекта вместо детализированной
- ▶ Таких уровней детализации может быть несколько, вплоть до почти непрерывного изменения детализации (Unreal 5 Nanite)

# LOD (level of detail)

- ▶ Когда модель далеко от камеры, рисуем упрощённую модель объекта вместо детализированной
- ▶ Таких уровней детализации может быть несколько, вплоть до почти непрерывного изменения детализации (Unreal 5 Nanite)
- ▶ Упрощённая модель может иметь отдельные VAO/VBO/EBO, а может лежать вместе с основной моделью (рисование конкретного LOD'а сводится к передаче правильных `first` и `count` в `glDrawArrays` и т.п.)

# LOD (level of detail)

- ▶ Когда модель далеко от камеры, рисуем упрощённую модель объекта вместо детализированной
- ▶ Таких уровней детализации может быть несколько, вплоть до почти непрерывного изменения детализации (Unreal 5 Nanite)
- ▶ Упрощённая модель может иметь отдельные VAO/VBO/EBO, а может лежать вместе с основной моделью (рисование конкретного LOD'а сводится к передаче правильных `first` и `count` в `glDrawArrays` и т.п.)
- ▶ Автоматическая генерация упрощённой модели – предмет активных исследований
  - ▶ Большинство современных подходов используют `edge collapse`: пара вершин, соединённых ребром, схлопывается в одну вершину

# LOD (level of detail)

- ▶ Когда модель далеко от камеры, рисуем упрощённую модель объекта вместо детализированной
- ▶ Таких уровней детализации может быть несколько, вплоть до почти непрерывного изменения детализации (Unreal 5 Nanite)
- ▶ Упрощённая модель может иметь отдельные VAO/VBO/EBO, а может лежать вместе с основной моделью (рисование конкретного LOD'а сводится к передаче правильных `first` и `count` в `glDrawArrays` и т.п.)
- ▶ Автоматическая генерация упрощённой модели – предмет активных исследований
  - ▶ Большинство современных подходов используют `edge collapse`: пара вершин, соединённых ребром, схлопывается в одну вершину
- ▶ Конкретный уровень детализации выбирается на основе желаемого видимого размера треугольников

# LOD (level of detail)

- ▶ Когда модель далеко от камеры, рисуем упрощённую модель объекта вместо детализированной
- ▶ Таких уровней детализации может быть несколько, вплоть до почти непрерывного изменения детализации (Unreal 5 Nanite)
- ▶ Упрощённая модель может иметь отдельные VAO/VBO/EBO, а может лежать вместе с основной моделью (рисование конкретного LOD'а сводится к передаче правильных `first` и `count` в `glDrawArrays` и т.п.)
- ▶ Автоматическая генерация упрощённой модели – предмет активных исследований
  - ▶ Большинство современных подходов используют `edge collapse`: пара вершин, соединённых ребром, схлопывается в одну вершину
- ▶ Конкретный уровень детализации выбирается на основе желаемого видимого размера треугольников
- ▶ Выбор уровня детализации можно перенести на GPU (`indirect rendering`)

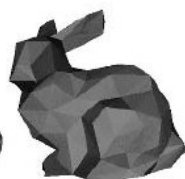
## LOD (level of detail)



69,451  
triangles



2,502  
triangles



251  
triangles



76  
triangles

# LOD (level of detail)

- ▶ Концепция LOD не ограничивается 3D-моделями



# LOD (level of detail)

- ▶ Концепция LOD не ограничивается 3D-моделями
- ▶ Например, хотим нарисовать огромный лес

# LOD (level of detail)

- ▶ Концепция LOD не ограничивается 3D-моделями
- ▶ Например, хотим нарисовать огромный лес
  - ▶ Близкие деревья используют детализированную модель

# LOD (level of detail)

- ▶ Концепция LOD не ограничивается 3D-моделями
- ▶ Например, хотим нарисовать огромный лес
  - ▶ Близкие деревья используют детализированную модель
  - ▶ Деревья подальше используют упрощённую модель

# LOD (level of detail)

- ▶ Концепция LOD не ограничивается 3D-моделями
- ▶ Например, хотим нарисовать огромный лес
  - ▶ Близкие деревья используют детализированную модель
  - ▶ Деревья подальше используют упрощённую модель
  - ▶ Далёкие деревья рисуются как billboard'ы с картой нормалей

# LOD (level of detail)

- ▶ Концепция LOD не ограничивается 3D-моделями
- ▶ Например, хотим нарисовать огромный лес
  - ▶ Близкие деревья используют детализированную модель
  - ▶ Деревья подальше используют упрощённую модель
  - ▶ Далёкие деревья рисуются как billboard'ы с картой нормалей
  - ▶ Очень далёкий лес рисуется сплошной поверхностью кроны деревьев с правильно подобранной BRDF

# Batching

- ▶ Группируем вместе объекты, использующие один шейдер, текстуру, настройки, и т.п., чтобы уменьшить количество OpenGL-вызовов для настроек рисования

# Batching

- ▶ Группируем вместе объекты, использующие один шейдер, текстуру, настройки, и т.п., чтобы уменьшить количество OpenGL-вызовов для настроек рисования
- ▶ Может быть неявным: объекты, использующие один шейдер/материал/etc сами по себе лежат в одном месте

# Batching

- ▶ Группируем вместе объекты, использующие один шейдер, текстуру, настройки, и т.п., чтобы уменьшить количество OpenGL-вызовов для настроек рисования
- ▶ Может быть неявным: объекты, использующие один шейдер/материал/etc сами по себе лежат в одном месте
- ▶ Может быть явным: движок рендеринга получает список объектов и сам их сортирует



# Batching

- ▶ Группируем вместе объекты, использующие один шейдер, текстуру, настройки, и т.п., чтобы уменьшить количество OpenGL-вызовов для настроек рисования
- ▶ Может быть неявным: объекты, использующие один шейдер/материал/etc сами по себе лежат в одном месте
- ▶ Может быть явным: движок рендеринга получает список объектов и сам их сортирует
- ▶ Конкретная реализация сильно зависит от ситуации:

# Batching

- ▶ Группируем вместе объекты, использующие один шейдер, текстуру, настройки, и т.п., чтобы уменьшить количество OpenGL-вызовов для настроек рисования
- ▶ Может быть неявным: объекты, использующие один шейдер/материал/etc сами по себе лежат в одном месте
- ▶ Может быть явным: движок рендеринга получает список объектов и сам их сортирует
- ▶ Конкретная реализация сильно зависит от ситуации:
  - ▶ Карты, ландшафт: один batch – один *тайл* (квадратный кусочек) карты

# Batching

- ▶ Группируем вместе объекты, использующие один шейдер, текстуру, настройки, и т.п., чтобы уменьшить количество OpenGL-вызовов для настроек рисования
- ▶ Может быть неявным: объекты, использующие один шейдер/материал/etc сами по себе лежат в одном месте
- ▶ Может быть явным: движок рендеринга получает список объектов и сам их сортирует
- ▶ Конкретная реализация сильно зависит от ситуации:
  - ▶ Карты, ландшафт: один batch – один *тайл* (квадратный кусочек) карты
  - ▶ Шутер: по batch'у на каждую небольшую группу зданий + на каждого персонажа + один batch на все пули

# Batching

- ▶ Группируем вместе объекты, использующие один шейдер, текстуру, настройки, и т.п., чтобы уменьшить количество OpenGL-вызовов для настроек рисования
- ▶ Может быть неявным: объекты, использующие один шейдер/материал/etc сами по себе лежат в одном месте
- ▶ Может быть явным: движок рендеринга получает список объектов и сам их сортирует
- ▶ Конкретная реализация сильно зависит от ситуации:
  - ▶ Карты, ландшафт: один batch – один *тайл* (квадратный кусочек) карты
  - ▶ Шутер: по batch'у на каждую небольшую группу зданий + на каждого персонажа + один batch на все пули
  - ▶ Визуализация графика функции: один batch на всё

## Instanced rendering (instancing)

- ▶ Позволяет рисовать несколько копий (instances) объекта одним OpenGL-вызовом

## Instanced rendering (instancing)

- ▶ Позволяет рисовать несколько копий (instances) объекта одним OpenGL-вызовом
- ▶ Обычно атрибуты вершин берутся в соответствии с индексом вершины: `offset + stride * vertexID`

# Instanced rendering (instancing)

- ▶ Позволяет рисовать несколько копий (instances) объекта одним OpenGL-вызовом
- ▶ Обычно атрибуты вершин берутся в соответствии с индексом вершины:  $\text{offset} + \text{stride} * \text{vertexID}$
- ▶ При использовании instancing конкретный атрибут вычисляется из номера instance:  
 $\text{offset} + \text{stride} * (\text{instanceID} / \text{divisor})$   
(целочисленное деление)

# Instanced rendering (instancing)

- ▶ Позволяет рисовать несколько копий (instances) объекта одним OpenGL-вызовом
- ▶ Обычно атрибуты вершин берутся в соответствии с индексом вершины:  $\text{offset} + \text{stride} * \text{vertexID}$
- ▶ При использовании instancing конкретный атрибут вычисляется из номера instance:  
 $\text{offset} + \text{stride} * (\text{instanceID} / \text{divisor})$   
(целочисленное деление)
- ▶ Включить instancing для конкретного атрибута: `glVertexAttribDivisor(index, divisor)`:
  - ▶ `divisor = 0`: атрибут не использует instancing и использует номер вершины
  - ▶ `divisor != 0`: атрибут использует instancing и использует номер instance (по формуле выше)



# Instanced rendering (instancing)

- ▶ Вызвать instanced rendering: `glDrawArraysInstanced`
  - ▶ Параметры – те же, что у `glDrawArrays`, плюс последний параметр – количество instance'ов (значения `instanceID` будут в диапазоне `0 .. instance_count-1`)

# Instanced rendering (instancing)

- ▶ Вызвать instanced rendering: `glDrawArraysInstanced`
  - ▶ Параметры – те же, что у `glDrawArrays`, плюс последний параметр – количество instance'ов (значения `instanceID` будут в диапазоне `0 .. instance_count-1`)
- ▶ Аналогично есть `glDrawElementsInstanced`

# Instanced rendering (instancing)

- ▶ Вызвать instanced rendering: `glDrawArraysInstanced`
  - ▶ Параметры – те же, что у `glDrawArrays`, плюс последний параметр – количество instance'ов (значения `instanceID` будут в диапазоне `0 .. instance_count-1`)
- ▶ Аналогично есть `glDrawElementsInstanced`
- ▶ Типичный пример использования:
  - ▶ 3D модель хранится в одном VBO/EBO, для её атрибутов `divisor = 0`
  - ▶ Атрибуты для каждого instance (напр. положение и поворот) хранятся в другом VBO, для них `divisor = 1`

# Instanced rendering (instancing)

- ▶ Вызвать instanced rendering: `glDrawArraysInstanced`
  - ▶ Параметры – те же, что у `glDrawArrays`, плюс последний параметр – количество instance'ов (значения `instanceID` будут в диапазоне `0 .. instance_count-1`)
- ▶ Аналогично есть `glDrawElementsInstanced`
- ▶ Типичный пример использования:
  - ▶ 3D модель хранится в одном VBO/EBO, для её атрибутов `divisor = 0`
  - ▶ Атрибуты для каждого instance (напр. положение и поворот) хранятся в другом VBO, для них `divisor = 1`
- ▶ В вершинном шейдере никаких изменений, все атрибуты равноправны

# Instanced rendering (instancing): пример шейдера

```
uniform mat4 camera_transform;

// divisor = 0
layout (location = 0) in vec3 in_position;
layout (location = 1) in vec3 in_normal;

// divisor = 1
layout (location = 2) in mat4 in_instance_transform;

out vec3 normal;

void main()
{
    gl_Position = camera_transform * in_instance_transform * in_position;

    normal = vec3(in_instance_transform) * in_normal;
}
```

# Instancing: производительность

- ▶ Instancing сам по себе не бесплатный, и имеет некоторый overhead

# Instancing: производительность

- ▶ Instancing сам по себе не бесплатный, и имеет некоторый overhead
- ▶ Обычно instancing становится быстрее чем цикл вызовов обычного рисования (по одному на каждый instance), когда число instance'ов достигает сотен или тысяч

## Instancing: ссылки

- ▶ [khronos.org/opengl/wiki/Vertex\\_Rendering#Instancing](https://www.khronos.org/opengl/wiki/Vertex_Rendering#Instancing)
- ▶ [learnopengl.com/Advanced-OpenGL/Instancing](https://learnopengl.com/Advanced-OpenGL/Instancing)
- ▶ [ogldev.org/www/tutorial33/tutorial33.html](https://ogldev.org/www/tutorial33/tutorial33.html)
- ▶ [habr.com/ru/post/352962](https://habr.com/ru/post/352962)



# Uniform buffers

- ▶ Позволяет использовать в качестве uniform-переменных данные из буфера

# Uniform buffers

- ▶ Позволяет использовать в качестве uniform-переменных данные из буфера
- ▶ Специальный тип (target) для буферов:  
GL\_UNIFORM\_BUFFER (создание и загрузка данных – так же, как для других буферов)

# Uniform buffers

- ▶ Позволяет использовать в качестве uniform-переменных данные из буфера
- ▶ Специальный тип (target) для буферов: `GL_UNIFORM_BUFFER` (создание и загрузка данных – так же, как для других буферов)
- ▶ В шейдере – т.н. *buffer-backed interface block*

# Uniform buffers

- ▶ Позволяет использовать в качестве uniform-переменных данные из буфера
- ▶ Специальный тип (target) для буферов: `GL_UNIFORM_BUFFER` (создание и загрузка данных – так же, как для других буферов)
- ▶ В шейдере – т.н. *buffer-backed interface block*
- ▶ Нужно быть внимательным с memory layout данных в буфере (конкретные правила описаны в спецификации)

# Uniform buffers

- ▶ Позволяет использовать в качестве uniform-переменных данные из буфера
- ▶ Специальный тип (target) для буферов:  
GL\_UNIFORM\_BUFFER (создание и загрузка данных – так же, как для других буферов)
- ▶ В шейдере – т.н. *buffer-backed interface block*
- ▶ Нужно быть внимательным с memory layout данных в буфере (конкретные правила описаны в спецификации)
- ▶ Каждый interface block нужно привязать к *binding index*:  
glGetUniformBlockIndex + glUniformBlockBinding (в OpenGL 4.2 можно задать прямо в шейдере)
- ▶ Uniform buffer нужно привязать к тому же *binding index*:  
glBindBufferBase или glBindBufferRange

## Uniform buffers: пример шейдера

```
#define MAX_BONES 256

layout (std140) uniform Bones
{
    mat4x3 bones[MAX_BONES];
};
```

# Uniform buffers

- ▶ OpenGL гарантирует поддержку как минимум 1024 компонент uniform-переменных (напр. `vec4` это 4 компоненты)

# Uniform buffers

- ▶ OpenGL гарантирует поддержку как минимум 1024 компонент uniform-переменных (напр. `vec4` это 4 компоненты)
- ▶ Для uniform buffers гарантируется максимальный размер как минимум в 16Kb  $\Rightarrow$  позволяют передать больше данных в шейдер



## Uniform buffers: ссылки

- ▶ [khronos.org/opengl/wiki/Uniform\\_Buffer\\_Object](https://www.khronos.org/opengl/wiki/Uniform_Buffer_Object)
- ▶ [khronos.org/opengl/wiki/Interface\\_Block\\_\(GLSL\)#Buffer\\_backed](https://www.khronos.org/opengl/wiki/Interface_Block_(GLSL)#Buffer_backed)
- ▶ [learnopengl.com/Advanced-OpenGL/Advanced-GLSL](https://learnopengl.com/Advanced-OpenGL/Advanced-GLSL)

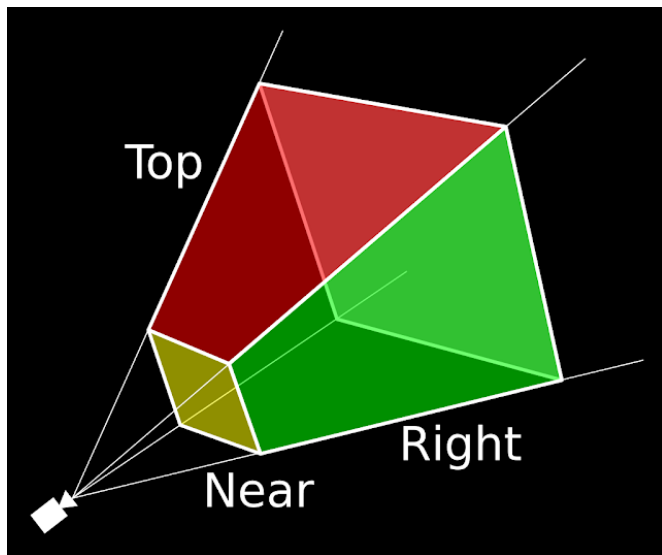
# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру

# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)

## Усечённая пирамида (frustum)



# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью

## Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью
- ▶ Оболочка должна:
  - ▶ Как можно плотнее прилегать к объекту

# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью
- ▶ Оболочка должна:
  - ▶ Как можно плотнее прилегать к объекту
  - ▶ Быть дешёвой для вычисления

# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью
- ▶ Оболочка должна:
  - ▶ Как можно плотнее прилегать к объекту
  - ▶ Быть дешёвой для вычисления
  - ▶ Иметь поменьше вершин (упрощает поиск пересечений)



# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью
- ▶ Оболочка должна:
  - ▶ Как можно плотнее прилегать к объекту
  - ▶ Быть дешёвой для вычисления
  - ▶ Иметь поменьше вершин (упрощает поиск пересечений)
  - ▶ Быть выпуклой (упрощает поиск пересечений)

# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью
- ▶ Оболочка должна:
  - ▶ Как можно плотнее прилегать к объекту
  - ▶ Быть дешёвой для вычисления
  - ▶ Иметь поменьше вершин (упрощает поиск пересечений)
  - ▶ Быть выпуклой (упрощает поиск пересечений)
- ▶ Варианты оболочки:
  - ▶ Выпуклая оболочка – плотно прилегает, но дорого вычислять и много вершин

# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью
- ▶ Оболочка должна:
  - ▶ Как можно плотнее прилегать к объекту
  - ▶ Быть дешёвой для вычисления
  - ▶ Иметь поменьше вершин (упрощает поиск пересечений)
  - ▶ Быть выпуклой (упрощает поиск пересечений)
- ▶ Варианты оболочки:
  - ▶ Выпуклая оболочка – плотно прилегает, но дорого вычислять и много вершин
  - ▶ Ограничивающая сфера/эллипсоид – может неплотно прилегать, легко вычислять, средний алгоритм пересечения (сложнее для эллипсоидов)

# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью
- ▶ Оболочка должна:
  - ▶ Как можно плотнее прилегать к объекту
  - ▶ Быть дешёвой для вычисления
  - ▶ Иметь поменьше вершин (упрощает поиск пересечений)
  - ▶ Быть выпуклой (упрощает поиск пересечений)
- ▶ Варианты оболочки:
  - ▶ Выпуклая оболочка – плотно прилегает, но дорого вычислять и много вершин
  - ▶ Ограничивающая сфера/эллипсоид – может неплотно прилегать, легко вычислять, средний алгоритм пересечения (сложнее для эллипсоидов)
  - ▶ AABB (axis-aligned bounding box) – легко вычислять (но нужно пересчитывать каждый кадр), не всегда плотно прилегает, легко искать пересечения

# Frustum culling

- ▶ Давайте не рисовать то, что заведомо не попадёт в камеру
- ▶ Видимая область камеры – параллелепипед (ортографическая проекция) или усечённая пирамида (перспективная проекция)
- ▶ Для каждого объекта нужно посчитать какую-нибудь оболочку (bounding volume) и пересечь её с видимой областью
- ▶ Оболочка должна:
  - ▶ Как можно плотнее прилегать к объекту
  - ▶ Быть дешёвой для вычисления
  - ▶ Иметь поменьше вершин (упрощает поиск пересечений)
  - ▶ Быть выпуклой (упрощает поиск пересечений)
- ▶ Варианты оболочки:
  - ▶ Выпуклая оболочка – плотно прилегает, но дорого вычислять и много вершин
  - ▶ Ограничивающая сфера/эллипсоид – может неплотно прилегать, легко вычислять, средний алгоритм пересечения (сложнее для эллипсоидов)
  - ▶ AABB (axis-aligned bounding box) – легко вычислять (но нужно пересчитывать каждый кадр), не всегда плотно прилегает, легко искать пересечения
  - ▶ OBB (oriented bounding box) – легко вычислять (предподсчитали для модели, поворачиваем вместе с моделью), плотно прилегает, легко искать пересечения

## Frustum culling: SAT

- ▶ Для детектирования пересечения выпуклых многогранников (frustum, AABB, OBB, etc.) обычно используется алгоритм, основанный на SAT – Separating Axis Theorem (в математике известна как HST – Hyperplane Separation Theorem)

## Frustum culling: SAT

- ▶ Для детектирования пересечения выпуклых многогранников (frustum, AABB, OBB, etc.) обычно используется алгоритм, основанный на SAT – Separating Axis Theorem (в математике известна как HST – Hyperplane Separation Theorem)
- ▶ Пусть  $A$  и  $B$  – два непересекающихся замкнутых выпуклых подмножества Евклидова пространства (есть версия для Банаховых пространств). Тогда:

## Frustum culling: SAT

- ▶ Для детектирования пересечения выпуклых многогранников (frustum, AABB, OBB, etc.) обычно используется алгоритм, основанный на SAT – Separating Axis Theorem (в математике известна как HST – Hyperplane Separation Theorem)
- ▶ Пусть  $A$  и  $B$  – два непересекающихся замкнутых выпуклых подмножества Евклидова пространства (есть версия для Банаховых пространств). Тогда:
  - ▶ HST: существует гиперплоскость (separating hyperplane), проходящая между этими подмножествами



## Frustum culling: SAT

- ▶ Для детектирования пересечения выпуклых многогранников (frustum, AABB, OBB, etc.) обычно используется алгоритм, основанный на SAT – Separating Axis Theorem (в математике известна как HST – Hyperplane Separation Theorem)
- ▶ Пусть  $A$  и  $B$  – два непересекающихся замкнутых выпуклых подмножества Евклидова пространства (есть версия для Банаховых пространств). Тогда:
  - ▶ HST: существует гиперплоскость (separating hyperplane), проходящая между этими подмножествами
  - ▶ SAT: существует прямая (separating axis), такая, что проекции  $A$  и  $B$  на эту прямую не пересекаются

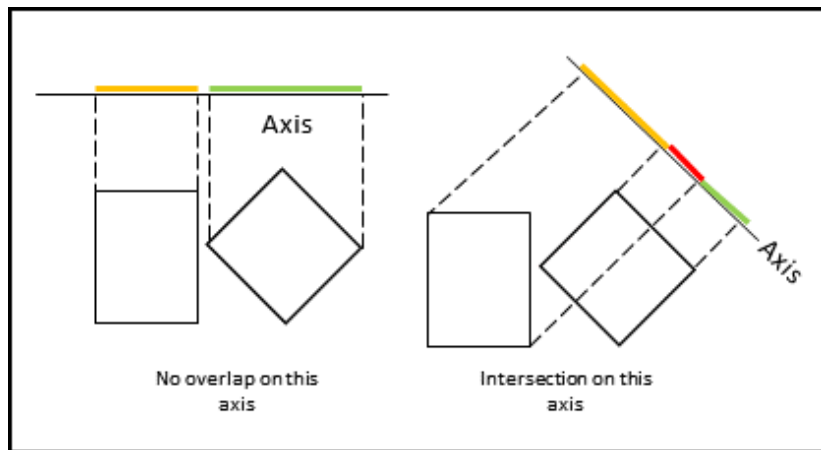
## Frustum culling: SAT

- ▶ Для детектирования пересечения выпуклых многогранников (frustum, AABB, OBB, etc.) обычно используется алгоритм, основанный на SAT – Separating Axis Theorem (в математике известна как HST – Hyperplane Separation Theorem)
- ▶ Пусть A и B – два непересекающихся замкнутых выпуклых подмножества Евклидова пространства (есть версия для Банаховых пространств). Тогда:
  - ▶ HST: существует гиперплоскость (separating hyperplane), проходящая между этими подмножествами
  - ▶ SAT: существует прямая (separating axis), такая, что проекции A и B на эту прямую не пересекаются
  - ▶ HST  $\Leftrightarrow$  SAT: гиперплоскость  $\Leftrightarrow$  перпендикулярная ей прямая

## Frustum culling: SAT

- ▶ Для детектирования пересечения выпуклых многогранников (frustum, AABB, OBB, etc.) обычно используется алгоритм, основанный на SAT – Separating Axis Theorem (в математике известна как HST – Hyperplane Separation Theorem)
- ▶ Пусть A и B – два непересекающихся замкнутых выпуклых подмножества Евклидова пространства (есть версия для Банаховых пространств). Тогда:
  - ▶ HST: существует гиперплоскость (separating hyperplane), проходящая между этими подмножествами
  - ▶ SAT: существует прямая (separating axis), такая, что проекции A и B на эту прямую не пересекаются
  - ▶  $HST \Leftrightarrow SAT$ : гиперплоскость  $\Leftrightarrow$  перпендикулярная ей прямая
- ▶ N.B.: на этой же теореме основывается метод SVM – Support Vector Machine
- ▶ N.B.: тот же алгоритм используется для детектирования столкновений в физических движках

# Frustum culling: SAT



## Frustum culling: вычисление проекции на прямую

- ▶ Проекция выпуклого замкнутого ограниченного множества на прямую – отрезок

## Frustum culling: вычисление проекции на прямую

- ▶ Проекция выпуклого замкнутого ограниченного множества на прямую – отрезок
- ▶ Нужно выбрать некоторую точку  $o$  на прямой, тогда проекция точки  $p$  на прямую вычисляется как  $(p - o) \cdot n$  (где  $n$  – вектор направления прямой)

## Frustum culling: вычисление проекции на прямую

- ▶ Проекция выпуклого замкнутого ограниченного множества на прямую – отрезок
- ▶ Нужно выбрать некоторую точку  $o$  на прямой, тогда проекция точки  $p$  на прямую вычисляется как  $(p - o) \cdot n$  (где  $n$  – вектор направления прямой)
- ▶ Выбор другой точки  $o$  или замена вектора  $n$  на коллинеарный приведёт к сдвигу и масштабированию проекций  $\Rightarrow$  непересекающиеся проекции останутся непересекающимися

## Frustum culling: вычисление проекции на прямую

- ▶ Проекция выпуклого замкнутого ограниченного множества на прямую – отрезок
- ▶ Нужно выбрать некоторую точку  $o$  на прямой, тогда проекция точки  $p$  на прямую вычисляется как  $(p - o) \cdot n$  (где  $n$  – вектор направления прямой)
- ▶ Выбор другой точки  $o$  или замена вектора  $n$  на коллинеарный приведёт к сдвигу и масштабированию проекций  $\Rightarrow$  непересекающиеся проекции останутся непересекающимися
- ▶  $\Rightarrow$  нам не важна начальная точка, можно вычислять  $p \cdot n$  (интерпретируя точку  $p$  как радиус-вектор из начала координат)



# Frustum culling: вычисление проекции на прямую

Псевдокод вычисления проекции выпуклого множества на прямую:

```
float vmin = inf, vmax = -inf;
for (p : vertices) {
    float v = dot(p, n);
    vmin = min(vmin, v);
    vmax = max(vmax, v);
}
```

# Frustum culling: вычисление пересечения проекций

- ▶ Число принадлежит отрезку  $[v_{min}, v_{max}]$ , если  $v_{min} \leq v \leq v_{max}$

# Frustum culling: вычисление пересечения проекций

- ▶ Число принадлежит отрезку  $[v_{min}, v_{max}]$ , если  $v_{min} \leq v \leq v_{max}$
- ▶ Число принадлежит пересечению двух отрезков, если выполняются два уравнения

$$\begin{cases} v_{min}^1 \leq v \leq v_{max}^1 \\ v_{min}^2 \leq v \leq v_{max}^2 \end{cases} \quad (1)$$

# Frustum culling: вычисление пересечения проекций

- ▶ Число принадлежит отрезку  $[v_{min}, v_{max}]$ , если  $v_{min} \leq v \leq v_{max}$
- ▶ Число принадлежит пересечению двух отрезков, если выполняются два уравнения

$$\begin{cases} v_{min}^1 \leq v \leq v_{max}^1 \\ v_{min}^2 \leq v \leq v_{max}^2 \end{cases} \quad (1)$$

- ▶ Чтобы эта система имела решения (т.е. отрезки пересекались), нужно

$$\begin{cases} v_{min}^1 \leq v_{max}^2 \\ v_{min}^2 \leq v_{max}^1 \end{cases} \quad (2)$$

# Frustum culling: вычисление пересечения проекций

Псевдокод детектирования пересечения двух отрезков:

```
return v1min <= v2max && v2min <= v1max;
```

# Frustum culling: SAT

- ▶ Если проекции объектов на любые прямые пересекаются, то объекты пересекаются

# Frustum culling: SAT

- ▶ Если проекции объектов на любые прямые пересекаются, то объекты пересекаются
- ▶ Если проекции объектов хотя бы на одну прямую не пересекаются, то объекты не пересекаются

# Frustum culling: SAT

- ▶ Если проекции объектов на любые прямые пересекаются, то объекты пересекаются
- ▶ Если проекции объектов хотя бы на одну прямую не пересекаются, то объекты не пересекаются
- ▶ Возможных прямых бесконечно много  $\Rightarrow$  нужно выбрать конечное число прямых



## Frustum culling: SAT, 2D

- ▶ Будем мысленно сдвигать объекты вдоль separating axis друг к другу до первого пересечения

## Frustum culling: SAT, 2D

- ▶ Будем мысленно сдвигать объекты вдоль separating axis друг к другу до первого пересечения
- ▶ Три варианта пересечения:
  - ▶ Ребро-ребро
  - ▶ Ребро-вершина
  - ▶ Вершина-вершина

## Frustum culling: SAT, 2D

- ▶ Будем мысленно сдвигать объекты вдоль separating axis друг к другу до первого пересечения
- ▶ Три варианта пересечения:
  - ▶ Ребро-ребро
  - ▶ Ребро-вершина
  - ▶ Вершина-вершина
- ▶ Во всех трёх случаях в качестве направления separating axis можно взять нормаль к какому-нибудь ребру

## Frustum culling: SAT, 2D

- ▶ Будем мысленно сдвигать объекты вдоль separating axis друг к другу до первого пересечения
- ▶ Три варианта пересечения:
  - ▶ Ребро-ребро
  - ▶ Ребро-вершина
  - ▶ Вершина-вершина
- ▶ Во всех трёх случаях в качестве направления separating axis можно взять нормаль к какому-нибудь ребру
- ▶  $\Rightarrow$  SAT для выпуклых многоугольников: в качестве множества направлений для separating axis берём множество нормалей к рёбрам обоих объектов

# SAT, 2D

```
bool intersect_along(Body b1, Body b2, vec3 n) {  
    auto p1 = project(b1.vertices, n);  
    auto p2 = project(b2.vertices, n);  
    return intersect(p1, p2);  
}  
  
bool intersect(Body b1, Body b2) {  
    for (n : b1.edge_normals) {  
        if (!intersect_along(b1, b2, n))  
            return false;  
    }  
  
    for (n : b2.edge_normals) {  
        if (!intersect_along(b1, b2, n))  
            return false;  
    }  
  
    return true;  
}
```

## Frustum culling: SAT, 3D

- ▶ Для 3D можно рассмотреть ту же идею, получится больше случаев

## Frustum culling: SAT, 3D

- ▶ Для 3D можно рассмотреть ту же идею, получится больше случаев
- ▶ SAT для выпуклых многогранников: в качестве множества направлений для separating axis берём
  - ▶ Множество нормалей к граням обоих объектов
  - ▶ + множество попарных векторных произведений  $e_1 \times e_2$ , где  $e_1$  и  $e_2$  – рёбра первого и второго многогранников, соответственно (для всех пар рёбер)

## Frustum culling: SAT, 3D

- ▶ Для 3D можно рассмотреть ту же идею, получится больше случаев
- ▶ SAT для выпуклых многогранников: в качестве множества направлений для separating axis берём
  - ▶ Множество нормалей к граням обоих объектов
  - ▶ + множество попарных векторных произведений  $e_1 \times e_2$ , где  $e_1$  и  $e_2$  – рёбра первого и второго многогранников, соответственно (для всех пар рёбер)
- ▶ N.B.: нас интересуют только направления с точностью до умножения на константу, так что во многих случаях необязательно рассматривать все грани и рёбра



## Frustum culling: SAT, 3D

- ▶ Для 3D можно рассмотреть ту же идею, получится больше случаев
- ▶ SAT для выпуклых многогранников: в качестве множества направлений для separating axis берём
  - ▶ Множество нормалей к граням обоих объектов
  - ▶ + множество попарных векторных произведений  $e_1 \times e_2$ , где  $e_1$  и  $e_2$  – рёбра первого и второго многогранников, соответственно (для всех пар рёбер)
- ▶ N.B.: нас интересуют только направления с точностью до умножения на константу, так что во многих случаях необязательно рассматривать все грани и рёбра
  - ▶ Например, у куба/параллелепипеда только три неколлинеарных нормали и три неколлинеарных ребра

## Frustum culling: SAT, 3D

```
bool intersect_along(Body b1, Body b2, vec3 n) {
    auto p1 = project(b1.vertices, n);
    auto p2 = project(b2.vertices, n);
    return intersect(p1, p2);
}

bool intersect(Body b1, Body b2) {
    for (n : b1.face_normals) {
        if (!intersect_along(b1, b2, n))
            return false;
    }

    for (n : b2.face_normals) {
        if (!intersect_along(b1, b2, n))
            return false;
    }

    for (e1 : b1.edges) {
        for (e2 : b2.edges) {
            vec3 n = cross(e1, e2);
            if (!intersect_along(b1, b2, n))
                return false;
        }
    }

    return true;
}
```

## Frustum culling: вычисление camera frustum

- ▶ В независимости от проекции, комбинаторно видимая область – куб, т.е. имеет 8 вершин, 12 рёбер, и 6 граней, соединённых как в кубе

## Frustum culling: вычисление camera frustum

- ▶ В независимости от проекции, комбинаторно видимая область – куб, т.е. имеет 8 вершин, 12 рёбер, и 6 граней, соединённых как в кубе
- ▶ Достаточно вычислить координаты 8-ми вершин

## Frustum culling: вычисление camera frustum

- ▶ В независимости от проекции, комбинаторно видимая область – куб, т.е. имеет 8 вершин, 12 рёбер, и 6 граней, соединённых как в кубе
- ▶ Достаточно вычислить координаты 8-ми вершин
- ▶ Можно вычислить напрямую из параметров и свойств камеры (напр. используя углы и тригонометрию)  $\Rightarrow$  сложный ad-hoc алгоритм

## Frustum culling: вычисление camera frustum

- ▶ В независимости от проекции, комбинаторно видимая область – куб, т.е. имеет 8 вершин, 12 рёбер, и 6 граней, соединённых как в кубе
- ▶ Достаточно вычислить координаты 8-ми вершин
- ▶ Можно вычислить напрямую из параметров и свойств камеры (напр. используя углы и тригонометрию)  $\Rightarrow$  сложный ad-hoc алгоритм
- ▶ Можно вычислить из матрицы  $T$  (view + projection):

## Frustum culling: вычисление camera frustum

- ▶ В независимости от проекции, комбинаторно видимая область – куб, т.е. имеет 8 вершин, 12 рёбер, и 6 граней, соединённых как в кубе
- ▶ Достаточно вычислить координаты 8-ми вершин
- ▶ Можно вычислить напрямую из параметров и свойств камеры (напр. используя углы и тригонометрию)  $\Rightarrow$  сложный ad-hoc алгоритм
- ▶ Можно вычислить из матрицы  $T$  (view + projection):
  - ▶ Координаты вершин видимой области в NDC (normalized device coordinates):  $(\pm 1, \pm 1, \pm 1)$

## Frustum culling: вычисление camera frustum

- ▶ В независимости от проекции, комбинаторно видимая область – куб, т.е. имеет 8 вершин, 12 рёбер, и 6 граней, соединённых как в кубе
- ▶ Достаточно вычислить координаты 8-ми вершин
- ▶ Можно вычислить напрямую из параметров и свойств камеры (напр. используя углы и тригонометрию)  $\Rightarrow$  сложный ad-hoc алгоритм
- ▶ Можно вычислить из матрицы  $T$  (view + projection):
  - ▶ Координаты вершин видимой области в NDC (normalized device coordinates):  $(\pm 1, \pm 1, \pm 1)$
  - ▶ Координаты вершин видимой области в сцене:  
 $\text{Proj}(T^{-1} \cdot (\pm 1, \pm 1, \pm 1, 1))$



# Frustum culling

- ▶ В идеале, хочется отсечь как можно больше

# Frustum culling

- ▶ В идеале, хочется отсечь как можно больше
- ▶ Отсекать на уровне треугольников неэффективно: GPU нарисует треугольник быстрее, чем мы проверим, что его не нужно рисовать

# Frustum culling

- ▶ В идеале, хочется отсечь как можно больше
- ▶ Отсекать на уровне треугольников неэффективно: GPU нарисует треугольник быстрее, чем мы проверим, что его не нужно рисовать
- ▶ Отсекать только на уровне огромных групп объектов неэффективно: будем рисовать все объекты, попавшие в группу, если хотя бы один объект виден

# Frustum culling

- ▶ В идеале, хочется отсечь как можно больше
- ▶ Отсекать на уровне треугольников неэффективно: GPU нарисует треугольник быстрее, чем мы проверим, что его не нужно рисовать
- ▶ Отсекать только на уровне огромных групп объектов неэффективно: будем рисовать все объекты, попавшие в группу, если хотя бы один объект виден
- ▶ Нужен какой-то баланс между стоимостью самого frustum culling'а и эффективностью отсека

# Frustum culling

- ▶ Объекты можно как-то группировать, чтобы отсекать сразу большие группы объектов, не попадающие в камеру:

# Frustum culling

- ▶ Объекты можно как-то группировать, чтобы отсекать сразу большие группы объектов, не попадающие в камеру:
  - ▶ Деревья (BVH – bounding volume hierarchy, octree, R-tree, ...)

# Frustum culling

- ▶ Объекты можно как-то группировать, чтобы отсекал сразу большие группы объектов, не попадающие в камеру:
  - ▶ Деревья (BVH – bounding volume hierarchy, octree, R-tree, ...)
  - ▶ Сетки/bins: группируем объекты в ячейки квадратной/кубической сетки, отсекаем сразу целые ячейки

# Frustum culling

- ▶ Объекты можно как-то группировать, чтобы отсекал сразу большие группы объектов, не попадающие в камеру:
  - ▶ Деревья (BVH – bounding volume hierarchy, octree, R-tree, ...)
  - ▶ Сетки/bins: группируем объекты в ячейки квадратной/кубической сетки, отсекаем сразу целые ячейки
- ▶ Можно перевести отсечение на GPU (indirect rendering)



## Frustum culling + batching

- ▶ Для уменьшения числа OpenGL-вызовов хочется хранить вместе (один VAO/VBO/EBO) как можно больше объектов

## Frustum culling + batching

- ▶ Для уменьшения числа OpenGL-вызовов хочется хранить вместе (один VAO/VBO/EBO) как можно больше объектов
- ▶ Для поддержки динамических сцен хочется хранить разные объекты по отдельности (проще добавлять/удалять объекты, обновлять буферы)

## Frustum culling + batching

- ▶ Для уменьшения числа OpenGL-вызовов хочется хранить вместе (один VAO/VBO/EBO) как можно больше объектов
- ▶ Для поддержки динамических сцен хочется хранить разные объекты по отдельности (проще добавлять/удалять объекты, обновлять буферы)
- ▶ Для frustum culling хочется хранить объекты по отдельности (чтобы по отдельности их отсекал и рисовать / не рисовать)

## Frustum culling + batching

- ▶ Для уменьшения числа OpenGL-вызовов хочется хранить вместе (один VAO/VBO/EBO) как можно больше объектов
- ▶ Для поддержки динамических сцен хочется хранить разные объекты по отдельности (проще добавлять/удалять объекты, обновлять буферы)
- ▶ Для frustum culling хочется хранить объекты по отдельности (чтобы по отдельности их отсекал и рисовать / не рисовать)
- ▶ Противоречащие друг другу требования

# Frustum culling + batching

- ▶ Для уменьшения числа OpenGL-вызовов хочется хранить вместе (один VAO/VBO/EBO) как можно больше объектов
- ▶ Для поддержки динамических сцен хочется хранить разные объекты по отдельности (проще добавлять/удалять объекты, обновлять буферы)
- ▶ Для frustum culling хочется хранить объекты по отдельности (чтобы по отдельности их отсекал и рисовать / не рисовать)
- ▶ Противоречащие друг другу требования
- ▶ Как найти баланс между ними – зависит от ситуации

## Frustum culling: ссылки

- ▶ Есть много способов написать этот алгоритм; есть вариации, оптимизированные для параллелограммов/кубов
- ▶ Часто в туториалах описывают **неправильный/неполный алгоритм**, выдающий false positive пересечения – не страшно для frustum culling, но неэффективно
- ▶ [en.wikipedia.org/wiki/Hyperplane\\_separation\\_theorem](https://en.wikipedia.org/wiki/Hyperplane_separation_theorem)
- ▶ Статья с разбором SAT и выводом алгоритмов для 2D и 3D
- ▶ Большая статья с разбором разных вариантов bounding volume

# Occlusion culling

- ▶ Не будем рисовать объекты (ocludees), закрытые другими объектами (occluders)

# Occlusion culling

- ▶ Не будем рисовать объекты (ocludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций



## Occlusion culling

- ▶ Не будем рисовать объекты (ocludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)

## Occlusion culling

- ▶ Не будем рисовать объекты (ocludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости

## Occlusion culling

- ▶ Не будем рисовать объекты (ocludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости
  - ▶ Можно нарисовать их обычным способом, и затем полагаться на early depth test (z pre-pass)

## Occlusion culling

- ▶ Не будем рисовать объекты (occludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости
  - ▶ Можно нарисовать их обычным способом, и затем полагаться на early depth test (z pre-pass)
- ▶ Можно использовать буфер глубины с предыдущего кадра

## Occlusion culling

- ▶ Не будем рисовать объекты (occludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости
  - ▶ Можно нарисовать их обычным способом, и затем полагаться на early depth test (z pre-pass)
- ▶ Можно использовать буфер глубины с предыдущего кадра
  - ▶ Нужно запомнить матрицу проекции предыдущего кадра

# Occlusion culling

- ▶ Не будем рисовать объекты (occludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости
  - ▶ Можно нарисовать их обычным способом, и затем полагаться на early depth test (z pre-pass)
- ▶ Можно использовать буфер глубины с предыдущего кадра
  - ▶ Нужно запомнить матрицу проекции предыдущего кадра
  - ▶ Может давать неточный результат

## Occlusion culling

- ▶ Не будем рисовать объекты (occludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости
  - ▶ Можно нарисовать их обычным способом, и затем полагаться на early depth test (z pre-pass)
- ▶ Можно использовать буфер глубины с предыдущего кадра
  - ▶ Нужно запомнить матрицу проекции предыдущего кадра
  - ▶ Может давать неточный результат
- ▶ Лучше построить max-mipmaps по буферу глубины (HiZ – hierarchical Z): специальным шейдером строить mipmaps, вычисляя максимум среди группы 2x2 пикселей

## Occlusion culling

- ▶ Не будем рисовать объекты (occludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости
  - ▶ Можно нарисовать их обычным способом, и затем полагаться на early depth test (z pre-pass)
- ▶ Можно использовать буфер глубины с предыдущего кадра
  - ▶ Нужно запомнить матрицу проекции предыдущего кадра
  - ▶ Может давать неточный результат
- ▶ Лучше построить max-mipmaps по буферу глубины (HiZ – hierarchical Z): специальным шейдером строить mipmaps, вычисляя максимум среди группы 2x2 пикселей
  - ▶ По размеру объекта (occludee) вычисляем mipmap-уровень: если в Z-буфере значение меньше, чем минимальная глубина нашего объекта, то объект не будет видно



# Occlusion culling

- ▶ Не будем рисовать объекты (occludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости
  - ▶ Можно нарисовать их обычным способом, и затем полагаться на early depth test (z pre-pass)
- ▶ Можно использовать буфер глубины с предыдущего кадра
  - ▶ Нужно запомнить матрицу проекции предыдущего кадра
  - ▶ Может давать неточный результат
- ▶ Лучше построить max-mipmaps по буферу глубины (HiZ – hierarchical Z): специальным шейдером строить mipmaps, вычисляя максимум среди группы 2x2 пикселей
  - ▶ По размеру объекта (occludee) вычисляем mipmap-уровень: если в Z-буфере значение меньше, чем минимальная глубина нашего объекта, то объект не будет видно
- ▶ Можно сгенерировать HiZ по списку известных occluders

# Occlusion culling

- ▶ Не будем рисовать объекты (occludees), закрытые другими объектами (occluders)
- ▶ **Очень** много вариаций
- ▶ Если есть заранее известные occluders (здания, ландшафт)
  - ▶ Можно растеризовать их на CPU в буфер глубины низкого разрешения, и использовать его для проверки видимости
  - ▶ Можно нарисовать их обычным способом, и затем полагаться на early depth test (z pre-pass)
- ▶ Можно использовать буфер глубины с предыдущего кадра
  - ▶ Нужно запомнить матрицу проекции предыдущего кадра
  - ▶ Может давать неточный результат
- ▶ Лучше построить max-mipmaps по буферу глубины (HiZ – hierarchical Z): специальным шейдером строить mipmaps, вычисляя максимум среди группы 2x2 пикселей
  - ▶ По размеру объекта (occludee) вычисляем mipmap-уровень: если в Z-буфере значение меньше, чем минимальная глубина нашего объекта, то объект не будет видно
- ▶ Можно сгенерировать HiZ по списку известных occluders
- ▶ Можно запомнить список видимых объектов с прошлого кадра и использовать их как occluders

# Occlusion culling

- ▶ Главное – избежать *pipeline stall*, т.е. блокирующего ожидания окончания работы GPU на CPU

# Occlusion culling

- ▶ Главное – избежать *pipeline stall*, т.е. блокирующего ожидания окончания работы GPU на CPU
- ▶ Если Z-буфер для occlusion culling (возможно, в виде HiZ) построен на CPU, проверку тоже можно осуществлять на CPU

# Occlusion culling

- ▶ Главное – избежать *pipeline stall*, т.е. блокирующего ожидания окончания работы GPU на CPU
- ▶ Если Z-буфер для occlusion culling (возможно, в виде HiZ) построен на CPU, проверку тоже можно осуществлять на CPU
- ▶ Если Z-буфер построен на GPU (взят с прошлого кадра, или построен по известным occluders), проверку хочется делать на GPU

# Occlusion culling

- ▶ Главное – избежать *pipeline stall*, т.е. блокирующего ожидания окончания работы GPU на CPU
- ▶ Если Z-буфер для occlusion culling (возможно, в виде HiZ) построен на CPU, проверку тоже можно осуществлять на CPU
- ▶ Если Z-буфер построен на GPU (взят с прошлого кадра, или построен по известным occluders), проверку хочется делать на GPU
  - ▶ Можно использовать compute shaders + indirect rendering (OpenGL 4.0)

# Occlusion culling

- ▶ Главное – избежать *pipeline stall*, т.е. блокирующего ожидания окончания работы GPU на CPU
- ▶ Если Z-буфер для occlusion culling (возможно, в виде HiZ) построен на CPU, проверку тоже можно осуществлять на CPU
- ▶ Если Z-буфер построен на GPU (взят с прошлого кадра, или построен по известным occluders), проверку хочется делать на GPU
  - ▶ Можно использовать compute shaders + indirect rendering (OpenGL 4.0)
  - ▶ Можно использовать occlusion queries + conditional rendering

## Occlusion culling: conditional rendering

- ▶ Выключаем рисование в z-буфер и в цветовой буфер (нас интересует только то, **будет ли нарисован объект**)



## Occlusion culling: conditional rendering

- ▶ Выключаем рисование в z-буфер и в цветовой буфер (нас интересует только то, **будет ли нарисован** объект)
- ▶ Внутри  
`glBeginQuery(GL_ANY_SAMPLES_PASSED, query_id)`  
рисуем дешёвую аппроксимацию объекта (напр. bounding box)

## Occlusion culling: conditional rendering

- ▶ Выключаем рисование в z-буфер и в цветовой буфер (нас интересует только то, **будет ли нарисован** объект)
- ▶ Внутри  
`glBeginQuery(GL_ANY_SAMPLES_PASSED, query_id)`  
рисуем дешёвую аппроксимацию объекта (напр. bounding box)
- ▶ Обратно включаем рисование в z-буфер и в цветовой буфер

## Occlusion culling: conditional rendering

- ▶ Выключаем рисование в z-буфер и в цветовой буфер (нас интересует только то, **будет ли нарисован** объект)
- ▶ Внутри  
`glBeginQuery(GL_ANY_SAMPLES_PASSED, query_id)`  
рисуем дешёвую аппроксимацию объекта (напр. bounding box)
- ▶ Обратно включаем рисование в z-буфер и в цветовой буфер
- ▶ Внутри пары  
`glBeginConditionalRender(query_id, GL_QUERY_NO_WAIT)`  
и `glEndConditionalRender` рисуем сам объект

## Occlusion culling: ссылки

- ▶ Статья про conditional rendering на OpenGL wiki
- ▶ Статья про много вариантов реализации occlusion culling
- ▶ Ещё одна статья
- ▶ И ещё одна статья
- ▶ И ещё одна, очень большая, статья