

# Компьютерная графика

Лекция 15: оптимизация рендеринга, timer queries, batching, instancing, uniform buffers, frustum culling, occlusion culling

2021

# 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)
- ▶ Помещаются ли данные в кэш

# Оптимизация – это сложно

На производительность (CPU) влияют:

- ▶ Общая загруженность системы
- ▶ Количество и паттерн доступов к памяти (cache-friendliness)
- ▶ Помещаются ли данные в кэш
- ▶ Branch prediction

# Оптимизация – это сложно

На производительность (CPU) влияют:

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



# Оптимизация на GPU – это очень сложно

- ▶ Асинхронность

# Оптимизация на GPU – это очень сложно

- ▶ Асинхронность
- ▶ Параллельность

# Оптимизация на GPU – это очень сложно

- ▶ Асинхронность
- ▶ Параллельность
- ▶ Много встроенных операций (fixed-function pipeline)

# Оптимизация на GPU – это очень сложно

- ▶ Асинхронность
- ▶ Параллельность
- ▶ Много встроенных операций (fixed-function pipeline)
- ▶ Сложные операции с памятью (доступ к текстуре: mipmaps + фильтрация)

# Оптимизация на GPU – это очень сложно

- ▶ Асинхронность
- ▶ Параллельность
- ▶ Много встроенных операций (fixed-function pipeline)
- ▶ Сложные операции с памятью (доступ к текстуре: mipmaps + фильтрация)
- ▶ Многое другое

## 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();  
}
```

- ▶ `frame_end - 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();  
}
```

- ▶ `frame_end - frame_start` – сколько времени ушло на то, чтобы **вызвать OpenGL-команды**
- ▶ В реальности драйвер поставил их в очередь, и скорее всего GPU ещё 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: 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 используют 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 (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 (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 (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 (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
- ▶ **Не могут** быть вложенными

## Измерение времени работы – правильный способ: timer queries

- ▶ glBeginQuery/glEndQuery – статистика будет собрана для команд между этими вызовами
- ▶ **Не могут** быть вложенными

```
GLuint query_id;  
glGenQueries(1, &query_id);  
  
...  
  
glBegin(GL_TIME_ELAPSED, query_id);  
  
// что-нибудь рисуем  
  
glEnd(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 + 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 + 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 + 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 + 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 знаем, что что-то тормозит

# Поиск bottleneck'a

- ▶ Мы знаем, что что-то тормозит
- ▶ OpenGL pipeline включает много компонентов, какой именно тормозит?

# Поиск bottleneck'a

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

# Поиск bottleneck'а

- ▶ Мы знаем, что что-то 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 знаем, что что-то 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
- ▶ Удобно искать bottleneck 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 операций записи в память

# Поиск bottleneck'а

- ▶ 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 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 операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ 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
  - ▶ Стало лучше?  $\Rightarrow$  Слишком много операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше?  $\Rightarrow$  Слишком тяжёлый фрагментный 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 операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый фрагментный 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 операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый фрагментный 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 операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый фрагментный 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 операций записи в память
- ▶ Упростим до предела вершинный шейдер (напр. вернём фиксированные координаты)
  - ▶ Стало лучше? ⇒ Слишком тяжёлый фрагментный 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 вызова сложных функций (`sin`, `exp`, `pow`)

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем 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 вызова сложных функций ( $\sin$ ,  $\exp$ ,  $\text{pow}$ )
- ▶ 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)

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем 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

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем вызова сложных функций ( $\sin$ ,  $\exp$ ,  $\text{pow}$ )
- ▶ 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 вызова сложных функций ( $\sin$ ,  $\exp$ ,  $\text{pow}$ )
- ▶ 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)
- ▶ Близкие пиксели читают близкие части текстуры (лучше утилизируется текстурный кэш)

# Оптимизация шейдеров

- ▶ Выполняем меньше операций
- ▶ Избегаем вызова сложных функций ( $\sin$ ,  $\exp$ ,  $\text{pow}$ )
- ▶ 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)
- ▶ Близкие пиксели читают близкие части текстуры (лучше утилизируется текстурный кэш)
- ▶ Используем `matmul`'ы



# Оптимизация числа вершин

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

# Оптимизация числа 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 числа 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 LOD (level of detail)

# 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, чего не видно (закрыто другими объектами)

# Оптимизация чего угодно

- ▶ Рисуем поменьше
  - ▶ Frustum culling: не рисуем то, что не попадёт в камеру
  - ▶ Occlusion culling: не рисуем то, чего не видно (закрыто другими объектами)
- ▶ 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, чего не видно (закрыто другими объектами)
- ▶ 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, чего не видно (закрыто другими объектами)
- ▶ 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, чего не видно (закрыто другими объектами)
- ▶ 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, чего не видно (закрыто другими объектами)
- ▶ 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 вместо детализированной



# LOD (level of detail)

- ▶ Когда модель далеко от камеры, рисуем упрощённую модель 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 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 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 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 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`)

# Batching

- ▶ Может быть неявным: объекты, использующие один шейдер/материал/etc сами по себе лежат в одном месте

# Batching

- ▶ 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

## 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)

- ▶ 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)
- ▶ 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)

- ▶ 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)
- ▶ 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`

## 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: 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)

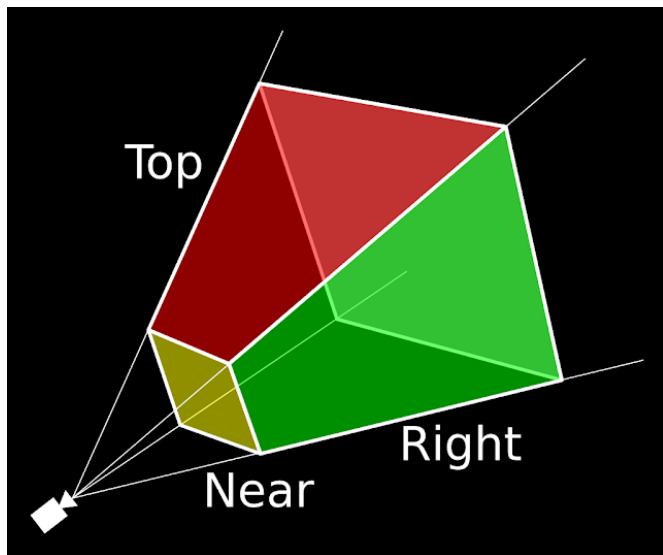
# Frustum culling

- ▶ 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 – легко вычислять, довольно нетривиальный алгоритм 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 – легко вычислять, довольно нетривиальный алгоритм пересечения
  - ▶ 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 – легко вычислять, довольно нетривиальный алгоритм пересечения
  - ▶ 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

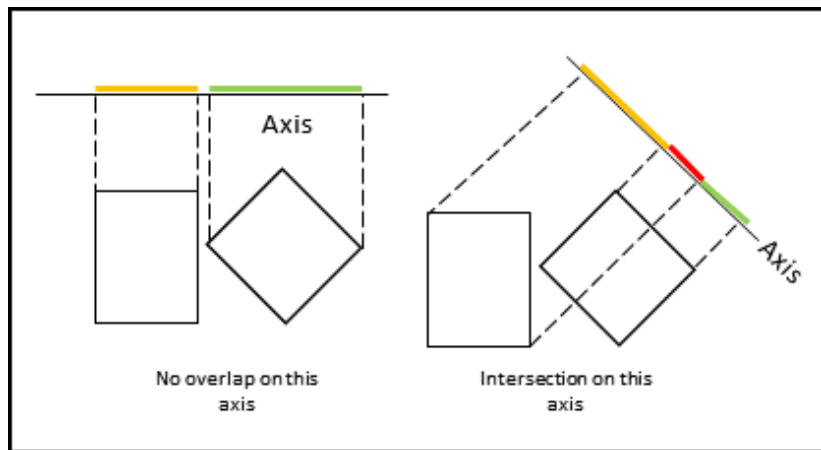
## Frustum culling: SAT

- ▶ 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:

```
if (v1min <= v2max && v2min <= v1max)
    return true;
else
    return false;
```



# Frustum culling: SAT

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

# Frustum culling: SAT

- ▶ Если проекции объектов на любые прямые пересекаются, то объекты пересекаются, иначе – не пересекаются
- ▶ Возможных прямых бесконечно много, мы можем проверить только конечное их число

## 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
- ▶ SAT для выпуклых многоугольников: в качестве множества направлений для separating axis берём множество нормалей к рёбрам обоих объектов

## Frustum culling: SAT, 3D

- ▶ Можно рассмотреть ту же идею, но в 3D больше случаев

## Frustum culling: SAT, 3D

- ▶ Можно рассмотреть ту же идею, но в 3D 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

- ▶ Можно рассмотреть ту же идею, но в 3D 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

- ▶ Можно рассмотреть ту же идею, но в 3D 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 true;
    }

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

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

    return false;
}
```

## 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  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  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  $\Rightarrow$  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

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

# 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, ...)

# 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: 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