

# Компьютерная графика

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

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)
- ▶ Много 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'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)

# 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?  $\Rightarrow$  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*`)
  - ▶ Стало лучше?  $\Rightarrow$  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 ( $\sin$ ,  $\exp$ ,  $\text{pow}$ )
- ▶ 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 ( $\sin$ ,  $\exp$ ,  $\log$ )
- ▶ 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 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 прочитать 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, что 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, что 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, что 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, что 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, что 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, что 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