Компьютерная графика

Практика 13: Timer queries, instancing, frustum culling, LOD

2021

Задание 1

Замеряем время рисования кадра с помощью timer queries

- Заводим `std::vector` для ID query объектов и ещё один для запоминания, свободен объект или нет

Задание 1

Замеряем время рисования кадра с помощью timer queries

- Заводим `std::vector` для ID query объектов и ещё один для запоминания, свободен объект или нет
- В начале каждого кадра находим индекс первого свободного query объекта, или, если такого нет, создаём новый и добавляем в массив (и помечаем как свободный)

Задание 1

Замеряем время рисования кадра с помощью timer queries

- Заводим `std::vector` для ID query объектов и ещё один для запоминания, свободен объект или нет
- В начале каждого кадра находим индекс первого свободного query объекта, или, если такого нет, создаём новый и добавляем в массив (и помечаем как свободный)
- Помечаем выбранный query объект как занятый, и вставляем `glBeginQuery(GL_TIME_ELAPSED, id)` в начало кадра (перед `glClear`) и соответствующий `glEndQuery` в конец кадра (перед `SwapBuffers`)

Задание 1

Замеряем время рисования кадра с помощью timer queries

- Заводим `std::vector` для ID query объектов и ещё один для запоминания, свободен объект или нет
- В начале каждого кадра находим индекс первого свободного query объекта, или, если такого нет, создаём новый и добавляем в массив (и помечаем как свободный)
- Помечаем выбранный query объект как занятый, и вставляем `glBeginQuery(GL_TIME_ELAPSED, id)` в начало кадра (перед `glClear`) и соответствующий `glEndQuery` в конец кадра (перед `SwapBuffers`)
- В конце кадра проверяем каждый query объект на то, готовы ли его данные: если готовы – достаём их и логируем (лучше разделить на 10^6 или 10^9 чтобы получить миллисекунды или секунды, соответственно), и помечаем как свободный

Задание 1

Замеряем время рисования кадра с помощью timer queries

- Заводим `std::vector` для ID query объектов и ещё один для запоминания, свободен объект или нет
- В начале каждого кадра находим индекс первого свободного query объекта, или, если такого нет, создаём новый и добавляем в массив (и помечаем как свободный)
- Помечаем выбранный query объект как занятый, и вставляем `glBeginQuery(GL_TIME_ELAPSED, id)` в начало кадра (перед `glClear`) и соответствующий `glEndQuery` в конец кадра (перед `SwapBuffers`)
- В конце кадра проверяем каждый query объект на то, готовы ли его данные: если готовы – достаём их и логируем (лучше разделить на 10^6 или 10^9 чтобы получить миллисекунды или секунды, соответственно), и помечаем как свободный
- В дальнейших заданиях имеет смысл смотреть на это значение и сравнивать с предыдущими заданиями (всё-таки мы занимаемся оптимизацией)

Задание 1

Замеряем время рисования кадра с помощью timer queries

- Заводим `std::vector` для ID query объектов и ещё один для запоминания, свободен объект или нет
- В начале каждого кадра находим индекс первого свободного query объекта, или, если такого нет, создаём новый и добавляем в массив (и помечаем как свободный)
- Помечаем выбранный query объект как занятый, и вставляем `glBeginQuery(GL_TIME_ELAPSED, id)` в начало кадра (перед `glClear`) и соответствующий `glEndQuery` в конец кадра (перед `SwapBuffers`)
- В конце кадра проверяем каждый query объект на то, готовы ли его данные: если готовы – достаём их и логируем (лучше разделить на 10^6 или 10^9 чтобы получить миллисекунды или секунды, соответственно), и помечаем как свободный
- В дальнейших заданиях имеет смысл смотреть на это значение и сравнивать с предыдущими заданиями (всё-таки мы занимаемся оптимизацией)
- В конце программы можно залогировать размер массива `query` объектов: это примерное отставание (в кадрах) GPU от CPU

Задание 2

Рисуем 1024 копии объекта

- В цикле рисуем копии объекта, например, сеткой, с координатами в диапазоне `[-16 .. 16)` с шагом в 1 по X и Z

Задание 2

Рисуем 1024 копии объекта

- В цикле рисуем копии объекта, например, сеткой, с координатами в диапазоне `[-16 .. 16)` с шагом в 1 по X и Z
- Для сдвига можно использовать uniform-переменную `offset` (она уже есть в коде)

Задание 3

Рисуем 1024 копии объекта, используя instancing

- В вершинном шейдере заменяем uniform-переменную `offset` на атрибут вершины (`location = 2`)

Задание 3

Рисуем 1024 копии объекта, используя instancing

- В вершинном шейдере заменяем uniform-переменную `offset` на атрибут вершины (`location = 2`)
- Заводим массив (`std::vector`) сдвигов (`glm::vec3`, 1024 штуки), использовавшихся в предыдущем задании, и заполняем при старте программы

Задание 3

Рисуем 1024 копии объекта, используя instancing

- В вершинном шейдере заменяем uniform-переменную `offset` на атрибут вершины (`location = 2`)
- Заводим массив (`std::vector`) сдвигов (`glm::vec3`, 1024 штуки), использовавшихся в предыдущем задании, и заполняем при старте программы
- Заводим VBO для этих сдвигов и загружаем в него данные (так же, как в обычный VBO)

Задание 3

Рисуем 1024 копии объекта, используя instancing

- В вершинном шейдере заменяем uniform-переменную `offset` на атрибут вершины (`location = 2`)
- Заводим массив (`std::vector`) сдвигов (`glm::vec3`, 1024 штуки), использовавшихся в предыдущем задании, и заполняем при старте программы
- Заводим VBO для этих сдвигов и загружаем в него данные (так же, как в обычный VBO)
- Настраиваем атрибут с `index = 2`, беря данные из нового VBO, и делаем `glVertexAttribDivisor(2, 1)` (чтобы этот атрибут менялся один раз на instance, а не на вершину)

Задание 3

Рисуем 1024 копии объекта, используя instancing

- В вершинном шейдере заменяем uniform-переменную `offset` на атрибут вершины (`location = 2`)
- Заводим массив (`std::vector`) сдвигов (`glm::vec3`, 1024 штуки), использовавшихся в предыдущем задании, и заполняем при старте программы
- Заводим VBO для этих сдвигов и загружаем в него данные (так же, как в обычный VBO)
- Настраиваем атрибут с `index = 2`, беря данные из нового VBO, и делаем `glVertexAttribDivisor(2, 1)` (чтобы этот атрибут менялся один раз на instance, а не на вершину)
- Вместо цикла по 1024 объектам делаем один вызов `glDrawElementsIsntanced`

Задание 4

Добавляем frustum culling

- Возвращаем цикл из задания 2 и uniform-переменную `offset` (атрибут можно убрать, он нам уже не нужен)
  - (код настройки атрибута, загрузки его данных и instanced рисования оставляем, но комментируем – чтобы можно было потом проверить)

Задание 4

Добавляем frustum culling

- Возвращаем цикл из задания 2 и uniform-переменную `offset` (атрибут можно убрать, он нам уже не нужен)
  - (код настройки атрибута, загрузки его данных и instanced рисования оставляем, но комментируем – чтобы можно было потом проверить)
- На старте, после загрузки модели вычисляем её bounding box функцией `bbox` (из `mesh_utils.hpp`) – получаем координаты минимальной и максимальной точек bounding box'а

Задание 4

Добавляем frustum culling

- Возвращаем цикл из задания 2 и uniform-переменную `offset` (атрибут можно убрать, он нам уже не нужен)
  - (код настройки атрибута, загрузки его данных и instanced рисования оставляем, но комментируем – чтобы можно было потом проверить)
- На старте, после загрузки модели вычисляем её bounding box функцией `bbox` (из `mesh_utils.hpp`) – получаем координаты минимальной и максимальной точек bounding box'а
- Каждый кадр создаём объект `frustum` (из `frustum.hpp`), передавая ему матрицу `projection*view`

Задание 4

Добавляем frustum culling

- Возвращаем цикл из задания 2 и uniform-переменную `offset` (атрибут можно убрать, он нам уже не нужен)
  - (код настройки атрибута, загрузки его данных и instanced рисования оставляем, но комментируем – чтобы можно было потом проверить)
- На старте, после загрузки модели вычисляем её bounding box функцией `bbox` (из `mesh_utils.hpp`) – получаем координаты минимальной и максимальной точек bounding box'а
- Каждый кадр создаём объект `frustum` (из `frustum.hpp`), передавая ему матрицу `projection*view`
- В цикле прохода по всем объектам, для каждого объекта создаём объект типа `aabb` (из `aabb.hpp`), учитывая bbox модели и offset текущего объекта

Задание 4

Добавляем frustum culling

- Возвращаем цикл из задания 2 и uniform-переменную `offset` (атрибут можно убрать, он нам уже не нужен)
  - (код настройки атрибута, загрузки его данных и instanced рисования оставляем, но комментируем – чтобы можно было потом проверить)
- На старте, после загрузки модели вычисляем её bounding box функцией `bbox` (из `mesh_utils.hpp`) – получаем координаты минимальной и максимальной точек bounding box'а
- Каждый кадр создаём объект `frustum` (из `frustum.hpp`), передавая ему матрицу `projection*view`
- В цикле прохода по всем объектам, для каждого объекта создаём объект типа `aabb` (из `aabb.hpp`), учитывая bbox модели и offset текущего объекта
- Если `aabb` объекта не пересекает `frustum` (функция `intersect` из `intersect.hpp`), пропускаем рисование этого объекта

Задание 4

Добавляем frustum culling

- Возвращаем цикл из задания 2 и uniform-переменную `offset` (атрибут можно убрать, он нам уже не нужен)
  - (код настройки атрибута, загрузки его данных и instanced рисования оставляем, но комментируем – чтобы можно было потом проверить)
- На старте, после загрузки модели вычисляем её bounding box функцией `bbox` (из `mesh_utils.hpp`) – получаем координаты минимальной и максимальной точек bounding box'а
- Каждый кадр создаём объект `frustum` (из `frustum.hpp`), передавая ему матрицу `projection*view`
- В цикле прохода по всем объектам, для каждого объекта создаём объект типа `aabb` (из `aabb.hpp`), учитывая bbox модели и offset текущего объекта
- Если `aabb` объекта не пересекает `frustum` (функция `intersect` из `intersect.hpp`), пропускаем рисование этого объекта
- Логируем число нарисованных объектов (если вы не меняли начальные параметры камеры, на старте будет 495 объектов)

Задание 5 (начало)

Добавляем LODs

- Вместо одного файла с моделью загружаем все шесть LOD'ов: `bunny0.obj .. bunny5.obj`

Задание 5 (начало)

Добавляем LODs

- Вместо одного файла с моделью загружаем все шесть LOD'ов: `bunny0.obj .. bunny5.obj`
- Храним все LOD'ы в одном наборе VAO+VBO+EBO:
  - Настройка VAO никак не меняется
  - VBO – последовательно записанные наборы вершин всех LOD'ов, от 0 до 5
  - EBO – последовательно записанные наборы индексов всех LOD'ов, от 0 до 5, но значения индексов нужно сдвинуть: например, индексы `bunny1.obj` начинаются с нуля, но тогда они будут ссылаться на вершины `bunny0.obj`, ведь они идут первыми в VBO, так что к индексам нужно прибавить количество вершин в `bunny0.obj`; аналогично, к индексам `bunny2.obj` нужно прибавить количество вершин из `bunny0.obj` и `bunny1.obj`, и т.д.

Задание 5 (продолжение)

Добавляем LODs

- Нам нужно знать, где начинаются индексы для каждого LOD'а и сколько их: удобно завести массив размера `N+1` (где `N` – количество LOD'ов, у нас `N = 6`), где `a[i]` – количество вершин в сумме во всех LOD'ах с номерами, меньшими `i` (в т.ч. `a[0] = 0` и `a[N]` – суммарное количество вершин во всех LOD'ах), тогда индексы `i`-ого LOD'а начинаются с `a[i]`, и их `a[i+1] - a[i]` штук
  - N.B.: последний параметр `glDrawElements` – не номер индекса, а сдвиг в байтах, приведённый к типу `void*`!

Задание 5 (продолжение)

Добавляем LODs

- Нам нужно знать, где начинаются индексы для каждого LOD'а и сколько их: удобно завести массив размера `N+1` (где `N` – количество LOD'ов, у нас `N = 6`), где `a[i]` – количество вершин в сумме во всех LOD'ах с номерами, меньшими `i` (в т.ч. `a[0] = 0` и `a[N]` – суммарное количество вершин во всех LOD'ах), тогда индексы `i`-ого LOD'а начинаются с `a[i]`, и их `a[i+1] - a[i]` штук
  - N.B.: последний параметр `glDrawElements` – не номер индекса, а сдвиг в байтах, приведённый к типу `void*`!
- При рендеринге вычисляем номер LOD, который мы будем использовать (например, как расстояние до камеры, делённое на фиксированное значение; координаты камеры можно вычислить из матрицы `view`)

Задание 5 (продолжение)

Добавляем LODs

- Нам нужно знать, где начинаются индексы для каждого LOD'а и сколько их: удобно завести массив размера `N+1` (где `N` – количество LOD'ов, у нас `N = 6`), где `a[i]` – количество вершин в сумме во всех LOD'ах с номерами, меньшими `i` (в т.ч. `a[0] = 0` и `a[N]` – суммарное количество вершин во всех LOD'ах), тогда индексы `i`-ого LOD'а начинаются с `a[i]`, и их `a[i+1] - a[i]` штук
  - N.B.: последний параметр `glDrawElements` – не номер индекса, а сдвиг в байтах, приведённый к типу `void*`!
- При рендеринге вычисляем номер LOD, который мы будем использовать (например, как расстояние до камеры, делённое на фиксированное значение; координаты камеры можно вычислить из матрицы `view`)
- Меняем вызов `glDrawElements`, чтобы рисовался только выбранный LOD