

# Компьютерная графика

Практика 14: Timer queries, instancing, frustum culling,  
LOD

---

2021



- В этой практике имеет смысл делать Release сборку проекта

# 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; делить нужно во floating-point), и помечаем как свободный
- N.B.: на некоторых GPU у timer queries очень большая погрешность, и могут получиться даже отрицательные числа
- 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 1

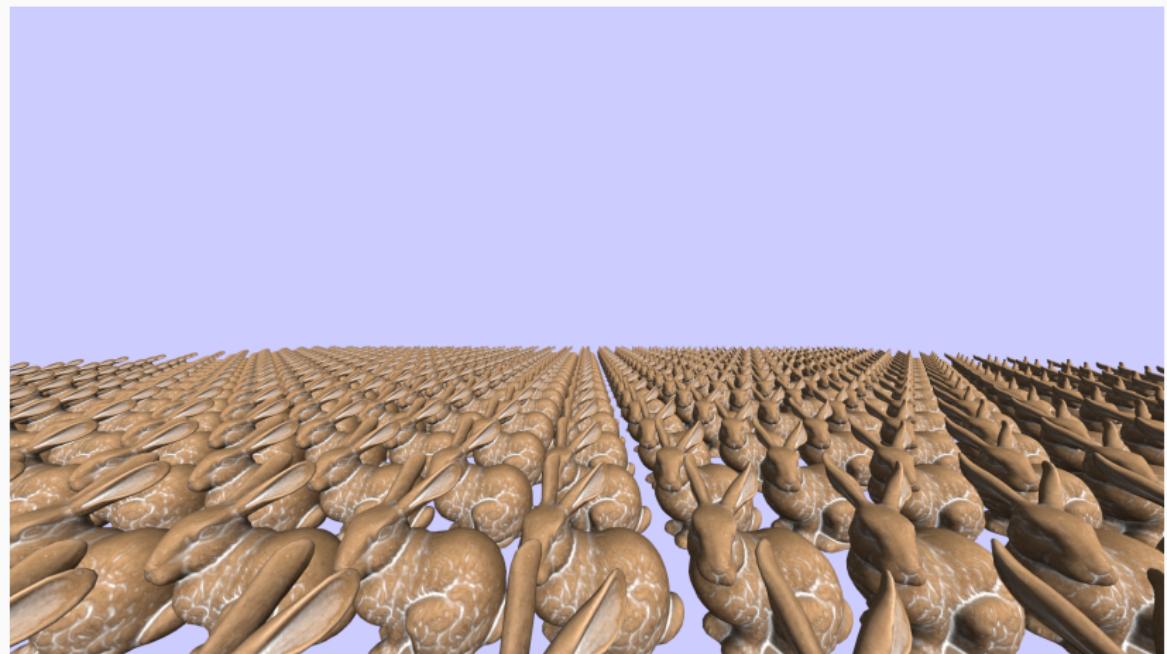


## Задание 2

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 model-матрицу (она уже есть в коде, нужно только обновлять значение через `glUniformMatrix4fv`)

## Задание 2

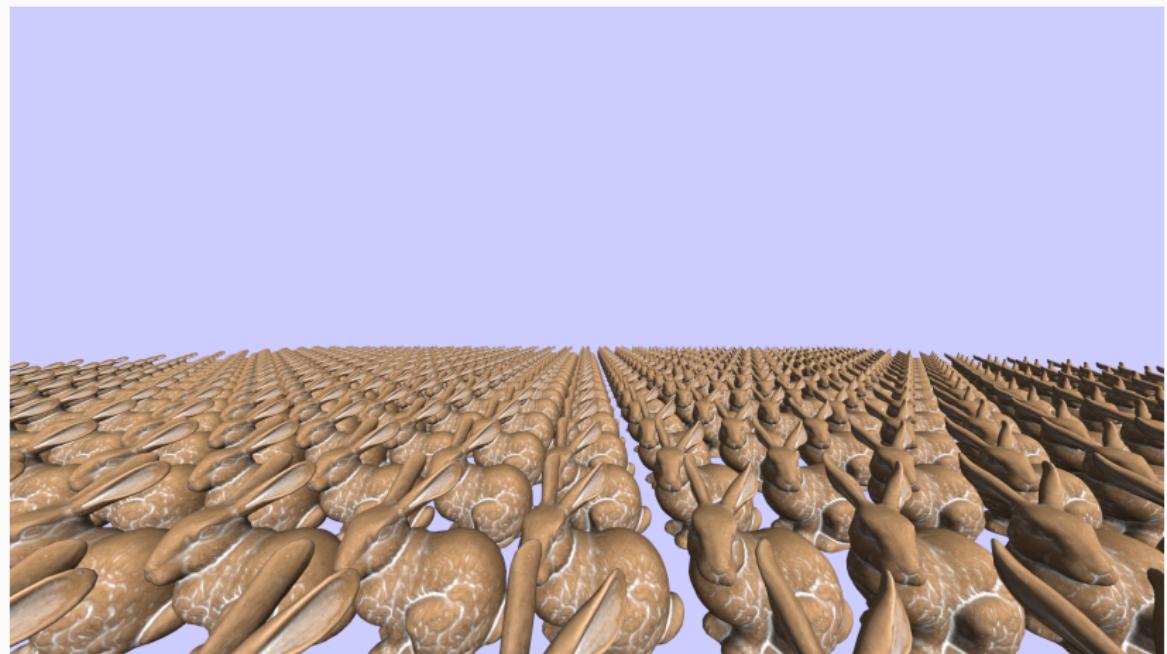


## Задание 3

Рисуем 1024 копии объекта, используя instancing

- В вершинном шейдере заводим новый атрибут вершины для позиции instance (`vec3, location = 3`) и прибавляем к `in_position` перед применением матриц
- Заводим массив `std::vector` сдвигов `glm::vec3`, использовавшихся в предыдущем задании, и заполняем при старте программы
- Заводим VBO для этих сдвигов и загружаем в него этот массив (так же, как в обычный VBO)
- Настраиваем атрибут с `index = 3`, беря данные из нового VBO, и делаем `glVertexAttribDivisor(3, 1)` (чтобы этот атрибут менялся один раз на instance, а не на вершину)
  - В коде есть несколько VAO, настроить нужно каждый!
  - Перед `glVertexAttribPointer` нужно будет сделать текущим ваш новый VBO
- Вместо цикла по 1024 объектам делаем один вызов `glDrawElementsInstanced`
- Model-матрицу меняем обратно на единичную

## Задание 3

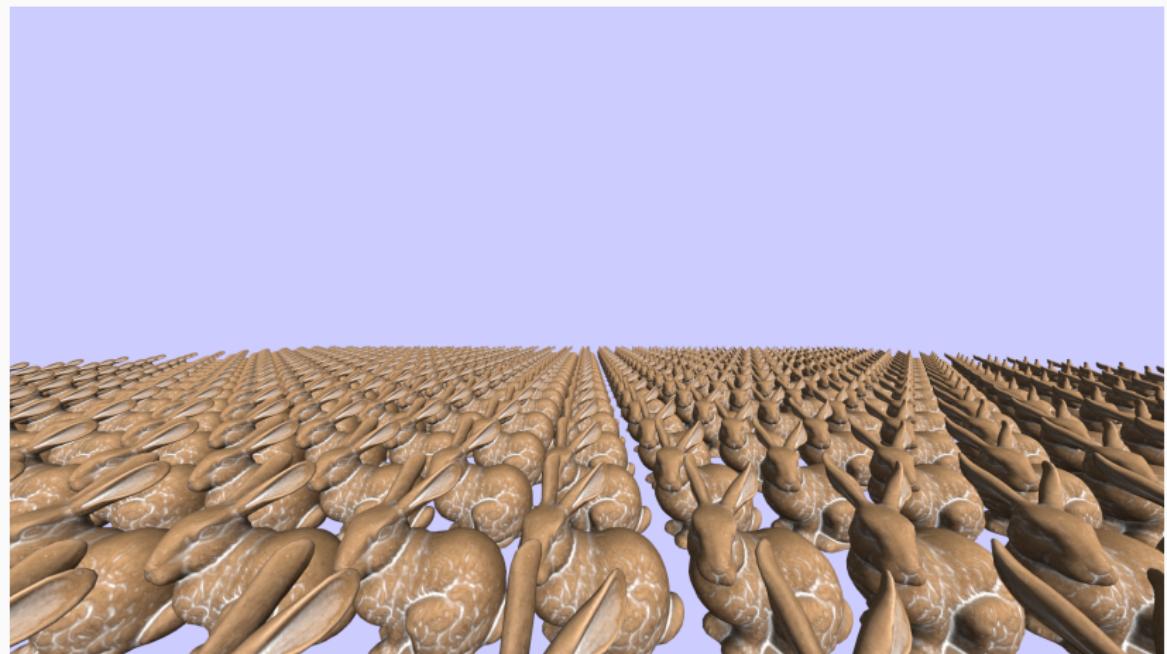


## Задание 4

### Добавляем frustum culling

- Каждый кадр создаём объект `frustum` (из `frustum.hpp`), передавая ему матрицу `projection * view`
- Вместо задания массива `instance` на старте, мы пересоздаём его на каждый кадр
- В цикле прохода по всем объектам (который теперь снова в цикле рисования) создаём объект типа `aabb` (из `aabb.hpp`), используя `bbox` модели (`input_model.meshes[0].min/max`) и прибавляя `offset` текущего объекта
- Если `aabb` объекта пересекает `frustum` (функция `intersect` из `intersect.hpp`), добавляем его в массив `instance'ов`
- Обновляем VBO с позициями `instance'ов`
- Логируем число нарисованных объектов (если вы не меняли начальные параметры камеры, на старте будет 500 объектов)

## Задание 4



## Задание 5\*

### Добавляем LOD

- Вместо одного массива `instance` заводим 6 штук – по одному на каждый уровень детализации
- Для каждого объекта вычисляем его номер LOD (например, как расстояние до камеры, делённое на фиксированное значение) и кладём в соответствующий массив
- После `frustum culling`'а делаем цикл по всем LOD [`0..5`], где
  - Загружаем массив `instance` для данного LOD в ваш VBO для данных `instance`
  - Рисуем модель `input_model.meshes[lod]` используя VAO `vaos[lod]`

## Задание 5\*

