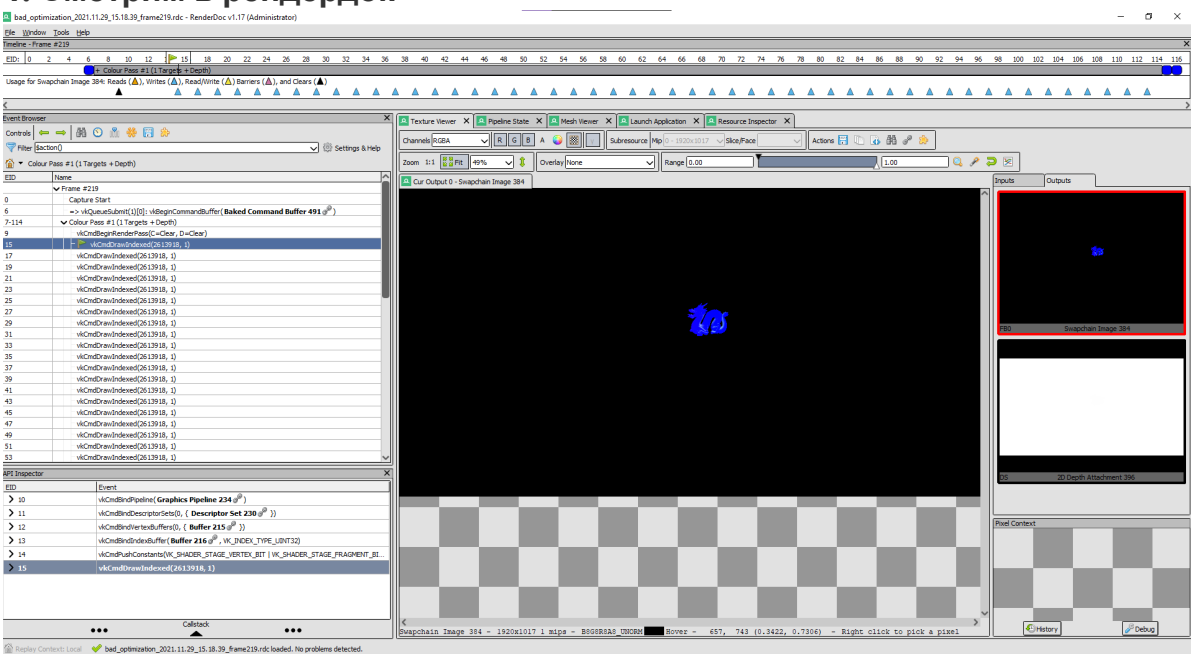
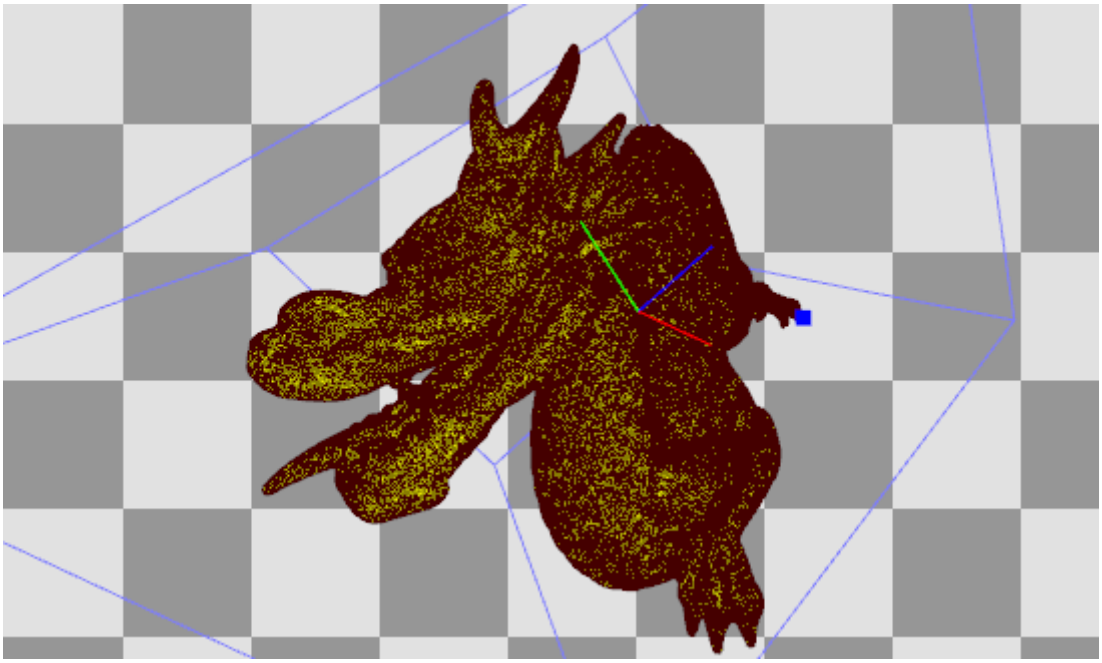
Оптимизация драконов

1. Смотрим в рендердок

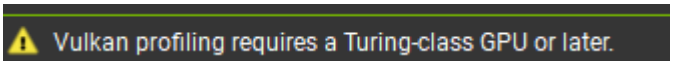


Видим классический случай "wireframe не отличается от нормальной модели". 2_613_918 вершин. Ну и куча абсолютно одинаковых draw call-ов. Пока основные рекомендации - lod + instancing. Ну и куллинг внутренних поверхностей лишним не будет



2. Смотрим в nvidia nsight

Видим, что технологически отстали от жизни:



Но впрочем, что-то программа показывает. Показывает, что два инстанса, занимающие бОльшую часть экрана в заданном кадре суммарно стоят нам ~50 мс, топ 4 инстанса - 48.8 мс. Вклад остальных в 10-тки раз меньше. Из этого делаю предварительный вывод, что отсутствие инстансинга - не главная наша проблема

3. Смотрим в код

Бейзлайн: когда на экране сразу все драоны, фпс почти всегда колеблется в районе 3-6

Для начала исследую вертексный шейдер, ведь на него оказывается сильнейшее давление. Оказывается, это стандартный шейдер из предыдущих домашек. Самое простое, и потенциально полезное, что я придумал - не повторять вычисление обратной модельной матрицы 2 раза, а вынести его в переменную. Это, ожидаемо, не принесло заметного результата и на этом этапе я решил оставить пока вершинный шейдер. Взял только на заметку функцию `DecodeNormal`. Там есть несколько ветвления, и вызов sqrt, но оптимизировать эти вещи пока явно преждевременнно.

```
mat3 inversed = mat3(transpose(inverse(params.mModel)));
```

Вот во фрагментном шейдере уже больше интересного. Я решил действовать следующим образом(в порядке обратном ожидаемому эффекту):

1. Убрать повторные вычисления (и заодно заменить деления на умножения, где получится)
    - Самый яркий пример - при установке изначального значения `time_dependent_color` Несколько раз вычислялись синусы и косинусы от времени

      ```
      const float time_multiplier = 1.231;
      const float pi = 3.14;
      const float half_pi = pi / 2.0;
      const float sin_time = sin(Params.time);
      const float sin_time2 = sin(Params.time * time_multiplier);
      const float cos_time_plus = cos(Params.time + half_pi);
      ```

    - Самая полезная замена деления на умножение(по факту просто убирание деления), скорее всего, произошла тут(внутри цикла и менее вероятно, что компилятор сам до этого додумается):

      ```
      // time_dependent_color.x += 1.0 / pow(max(time_fract - subtraction,
      addition + i), power)
      time_dependent_color.x += pow(max(time_fract - subtraction, addition +
      i), -power)
      ```

- Не уверен, что эффект действительно есть, но фпс начал колебаться между 5 и 6, а не 4 и 6

2. Минимизировать ветвления
    - Эффект не заметен на фоне погрешностей. Либо это действительно не важно в данном случае, либо я пытался быть умнее компилятора и проиграл
3. Оптимизировать циклы
    - Первый цикл - в функции `fog_density`.

      ```
      int num_cycles = min(int(z * 100.0), 10000);
      return 0.0003 * num_cycles;
      ```

      Ура, замена цикла в 10000 итераций на 1 строку нам всё таки помогла, фпс теперь обитает между 24 и 27!
    - Второй цикл
        - Наблюдение 1: `max()` всегда возвращает второй аргумент, потому что `0.0 <= fract() < 1.0` (печальное наблюдение, потому что теперь добавленное значение всегда зависит от `i`)
        - Наблюдение 2: внутренности цикла зависят от входных данных шейдера только в смысле выбора одного из двух набора констант, то есть цикл можно представить вот так

          ```
          const int num_cycles = N.x < 0 ? 300 : 400;
          const float addition = N.x < 0 ? 3.0 : 64.0;
          const float subtraction = N.x < 0 ? 0.0 : 0.3;
          const float power = N.x < 0 ? 7000 : 4000;

          for (int i = 0; i < num_cycles; ++i) {
              time_dependent_color.x += pow(addition + i, -power);
          }
          ```

          А значит, можно предпосчитать два возможных значения суммарной прибавки и выбирать нужную в зависимости от N.x. Так как в glsl нет поддержки consexpr, то простейший вариант - использовать питон

          ```
          def calc(less_than_zero):
              num_cycles = 300 if less_than_zero < 0 else 400
              addition = 3.0 if less_than_zero < 0 else 64.0
              subtraction = 0.0 if less_than_zero < 0 else 0.3
              power = 7000 if less_than_zero < 0 else 4000

              res = 0.0
              for i in range(num_cycles):
                  res += (addition + i)**(-power)
              return res

          print(f"less than zero: f{calc(True)}")
          print(f"more or equal than zero: f{calc(False)}")
          ```

          Питон выдал 0.0, что справедливо если подумать. Самый большой член такой суммы - 64^(-4000) = (2^(6))^(-4000), так что смело можно писать 0, даже без ифов :)

Это добавило ещё несколько фпс.

Пришло время добавить куллинг:

```
maker.rasterizer.cullMode = VK_CULL_MODE_FRONT_BIT;
```

Результат минимальный, либо пара фпс, либо вообще никакого.

Насчёт lod, я не уверен, что эта оптимизация формально подходит под описание "работает быстрее, а картинка та же самая", потому что формально говоря, изображение может поменяться.

Финальный результат: 28-29 фпс со всеми драконами на экране. то есть ускорение в ~9 раз