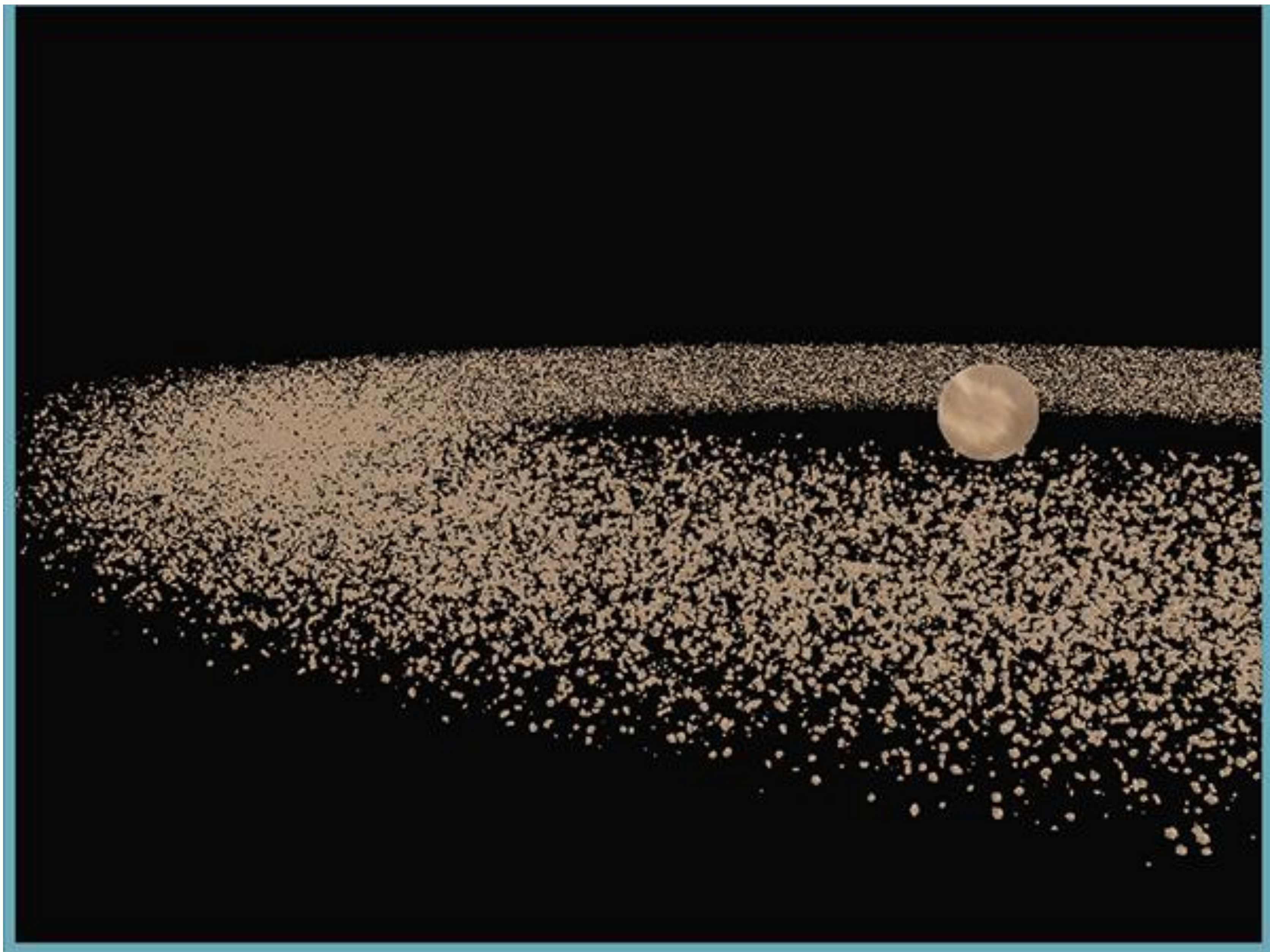
Инстанцирование, или инстансинг , или инстанцированный рендеринг

Компьютерная графика



Инстанцирование, или инстанцированный рендеринг

Инстанцирование, или инстанцированный рендеринг, — это способ выполнения одних и тех же команд рисования много раз подряд, причём каждая из них дает несколько иной результат.

Это может быть очень эффективным методом рендеринга большого объема геометрии с очень небольшим количеством вызовов API

Что хотелось бы

Сцена, содержащую огромное количество моделей объектов, причем преимущественно эти модели содержат одинаковые вершинные данные, разнятся только матрицы трансформации, примененные к ним.

Как-то так

```
for (unsigned int ix = 0; ix < model_count; ++ix) {
    // привязка VAO, текстур, установка юниформов, проч...
    DoSomePreparations();
    glDrawArrays(GL_TRIANGLES, 0, vertex_count);
}
```

Проблемы

При рендере множества экземпляров одной и той же модели мы быстро достигнем **бутылочного горлышка** в плане производительности — им станет **множество вызовов функций отрисовки примитивов**.

По сравнению с временными затратами на непосредственный рендер, **передача данных в GPU** о том, что вы хотите что-то отрендерить, с помощью функций типа **glDrawArrays или glDrawElemenets** занимает весьма ощутимое время.

Это **время** уходит на **подготовку, необходимую OpenGL** перед непосредственным **выводом** данных вершин: передача в GPU данных о текущем буфере чтения данных, расположении и формате данных вершинных атрибутов и прочая, прочая.

И весь этот обмен осуществляется по **относительно небыстрой шине**, связующей CPU и GPU.

Складывается **парадокс**альная ситуация: **рендер** вершинных данных **молниеносен**, но вот **передача команд на** осуществление **рендер**а довольно **медленная**.

Цель

Было бы здорово иметь возможность отправить необходимые данные в видеокарту однократно, а затем всего одним вызовом попросить OpenGL осуществить рендер множества объектов, используя эти данные.

Инстансинг

Инстансинг — технология, позволяющая выводить множество объектов, используя **один вызов функции отрисовки**, что избавляет нас от лишнего обмена CPU -> GPU при рендере.

Все что нужно сделать для начала использования инстансинга:

сменить вызовы glDrawArrays и glDrawElemenets на

glDrawArraysInstanced и glDrawElementsInstanced соответственно.

Версии, поддерживающие инстансинг, принимают один дополнительный параметр, помимо уже знакомых по обычным версиям функций.

Этот параметр — число экземпляров инстансинга, т.е. число отрисовываемых экземпляров модели.

Таким образом мы единожды передаём GPU все необходимые для рендера данные, а затем сообщаем ему как осуществить рендер желаемого числа экземпляров объекта всего за один вызов специальной функции.

И видеокарта отрисует все множество объектов без постоянного обращения к CPU.

Этого ли мы хотели?

Выведя тысячи объектов одним и тем же образом, в одном и том же положении, мы **в итоге** все равно получим **изображение единственного объекта** – все экземпляры окажутся наложены друг на друга

Решение этой проблемы

Для решения этой проблемы в вершинных шейдерах доступна встроенная переменная GLSL **gl_InstanceID**.

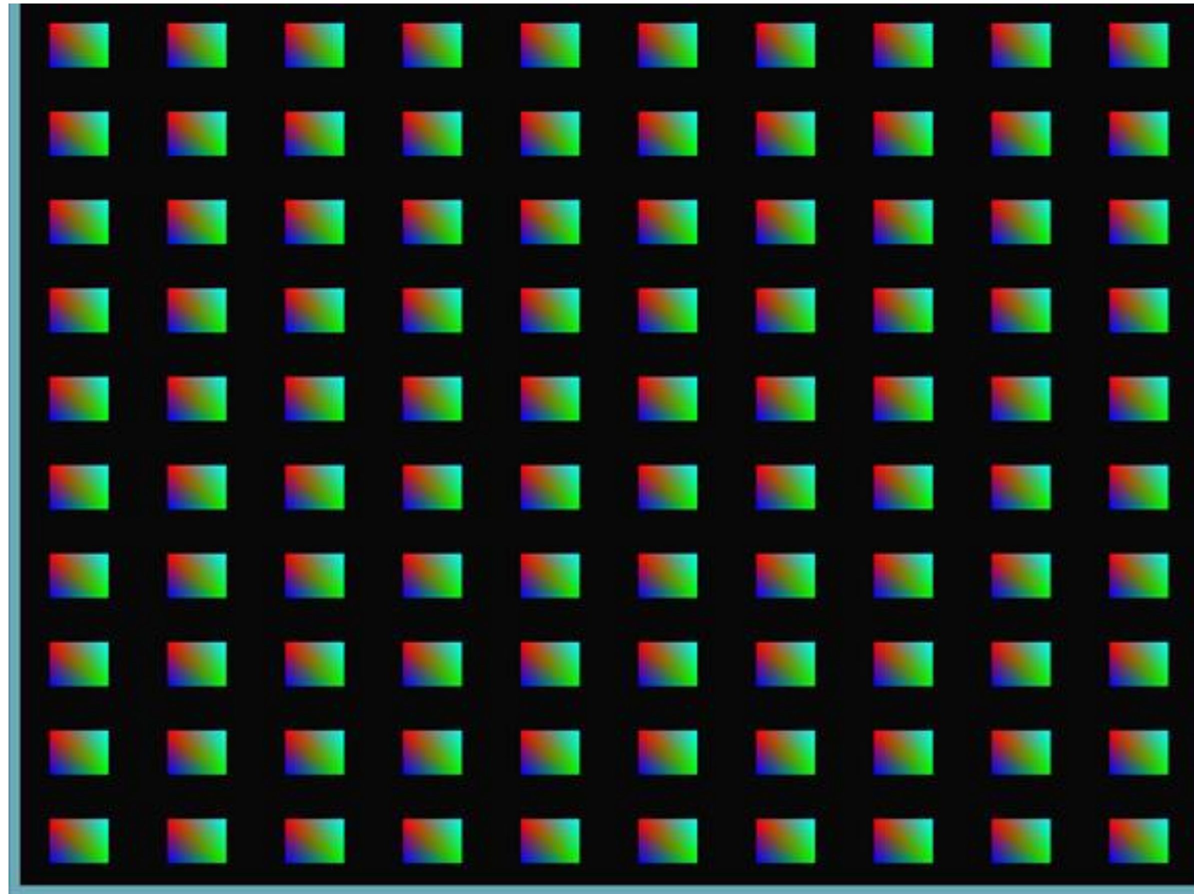
При **использовании** для **рендера функций**, поддерживающих **инстансинг**, **значение** данной переменной будет **увеличиваться на единицу** для каждого выводимого экземпляра, начиная с нуля.

Таким образом, рендеря 43-ий экземпляр объекта, в вершинном шейдере мы получим gl_InstanceID равную 42.

Имея уникальный индекс, соответствующий экземпляру, мы могли бы, к примеру, использовать его для выборки из большого массива векторов положений, дабы осуществить рендер каждого экземпляра в определенном месте сцены.

Если хотим так

Прямоугольники в нормализованных координатах устройства с помощью единственного вызова отрисовки.



Смещение определяется с помощью выборки из юниформа, представляющего собой массив, содержащий сто векторов смещения.

Данные для одного прямоугольника

```
float quadVertices[] = {
    // координаты  // цвета
    -0.05f,  0.05f,  1.0f, 0.0f, 0.0f,
     0.05f, -0.05f,  0.0f, 1.0f, 0.0f,
    -0.05f, -0.05f,  0.0f, 0.0f, 1.0f,

    -0.05f,  0.05f,  1.0f, 0.0f, 0.0f,
     0.05f, -0.05f,  0.0f, 1.0f, 0.0f,
     0.05f,  0.05f,  0.0f, 1.0f, 1.0f


};
```

Каждый прямоугольник составлен из двух треугольников, что даёт нам шесть вершин.

Каждая вершина содержит двухкомпонентный вектор положения и вектор цвета.

Размер треугольников подобран достаточно маленьким, чтобы корректно заполнять экран в больших количествах:

Фрагментный шейдер

```
#version 330 core
out vec4 FragColor;

in vec3 fColor;

void main()  {
    FragColor = vec4(fColor, 1.0);
}
```

Цвет прямоугольника задает фрагментный шейдер, который просто перенаправляет полученный из вершинного шейдера интерполированный цвет вершины прямо на выходную переменную

Вершинный шейдер

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;

out vec3 fColor;

uniform vec2 offsets[100];

void main()
{
   vec2 offset = offsets[gl_InstanceID];
   gl_Position = vec4(aPos + offset, 0.0, 1.0);
   fColor = aColor;
}
```

Здесь мы объявили юниформ-массив offsets, содержащий сто векторов смещения.

В коде шейдера мы получаем значение смещения путем выборки из массива по значению переменной gl_InstanceID.

Заполнение массива смещений

```
glm::vec2 translations[100];

int index = 0;
float offset = 0.1f;
for(int y = -10; y < 10; y += 2)   {
    for(int x = -10; x < 10; x += 2)  {
        glm::vec2 translation;
        translation.x = (float)x / 10.0f + offset;
        translation.y = (float)y / 10.0f + offset;
        translations[index++] = translation;
    }
}
```

Заполняется в приложении, до входа в основной цикл отрисовки

Передать данные в юниформ-массив шейдера

```
GLint vertexColorLocation = glGetUniformLocation(shaderProgram, "offsets");
glUniform2fv (vertexColorLocation, 100, translations);
```

```
void glUniform2fv ( GLint location, GLsizei count, const GLfloat *value );
```

Для массивов юниформ-переменных считается, что каждый элемент массива относится к типу, указанному в имени команды (например, glUniform2f или glUniform2fv можно использовать для загрузки массива юниформ-переменных типа vec2).

Количество элементов массива юниформ-переменных, подлежащих модификации, задаётся параметром count

Инстанцированный рендер

glDrawArraysInstanced или glDrawElementsInstanced для вызова инстанцированного рендера

```
glBindVertexArray(quadVAO);
glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100);
```

Проблема

Ограничение разрешенного объёма отправляемых шейдеру юниформ-данных

Инстанцированные массивы (instanced arrays)

Для **обычных вершинных** атрибутов GLSL осуществляет **выборку новых** значений вершинных данных с **каждым** очередным **выполнением** кода вершинного **шейдера**.

Однако, задавая **вершинный атрибут как инстанцированный массив**, мы заставляем GLSL осуществлять **выборку нового** значения атрибута **для каждого** очередного **экземпляра объекта**, а не очередной вершины объекта.

В итоге можно использовать **обычные вершинные** атрибуты для **данных**, представленных **повершинно**, а **инстанцированные массивы** для **данных, уникальных для экземпляра** объекта.

Обновлённый код шейдера

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aOffset;

out vec3 fColor;

void main() {
   gl_Position = vec4(aPos + aOffset, 0.0, 1.0);
   fColor = aColor;
}
```

Здесь мы более не используем переменную gl_InstanceID и можем напрямую обращаться к атрибуту **aOffset**, без необходимость выборки из массива.

Необходимо сохранить данные в объекте вершинного буфера и настроить указатель вершинного атрибута.

```
unsigned int instanceVBO;
glGenBuffers(1, &instanceVBO);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);

glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec2) * 100, &translations[0], GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, 0);
```

Настроить указатель вершинного атрибута и активировать атрибут

```
glEnableVertexAttribArray(2);

glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);

glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);

glBindBuffer(GL_ARRAY_BUFFER, 0);

glVertexAttribDivisor(2, 1);
```

Функция glVertexAttribDivisor

Функция указывает OpenGL, когда следует осуществлять выборку нового элемента из вершинного атрибута.

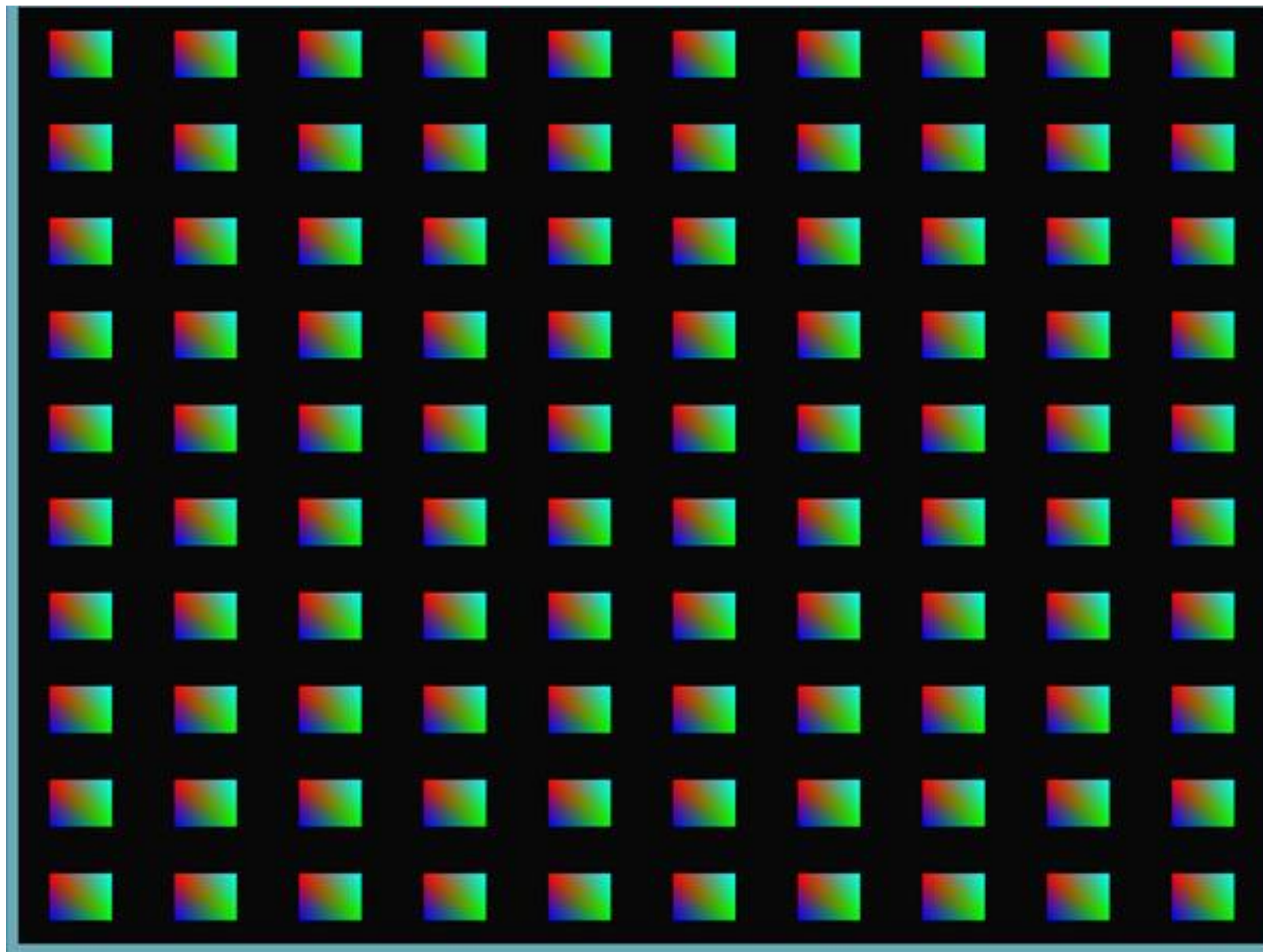
```
glVertexAttribDivisor(2, 1);
```

Первый параметр — индекс интересующего атрибута, а второй — разделитель атрибута (attribute divisor).

По умолчанию он установлен в 0, что соответствует обновлению атрибута для каждой новой обрабатываемой вершинным шейдером вершины.

Устанавливая этот параметр в 1, мы сообщаем OpenGL о том, что следует обновлять атрибут при рендере каждого последующего экземпляра.

Установив разделитель в значение 2, мы обеспечим обновление через каждые два экземпляра, и так далее.



```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
layout (location = 2) in vec2 aOffset;

out vec3 fColor;

void main() {
    vec2 pos = aPos * (gl_InstanceID / 100.0);
    gl_Position = vec4(pos + aOffset, 0.0, 1.0);
    fColor = aColor;
}
```

Попробуем постепенно уменьшать каждый прямоугольник, начиная с правого верхнего угла в направлении левого нижнего угла