Computer Graphics II
– Instancing

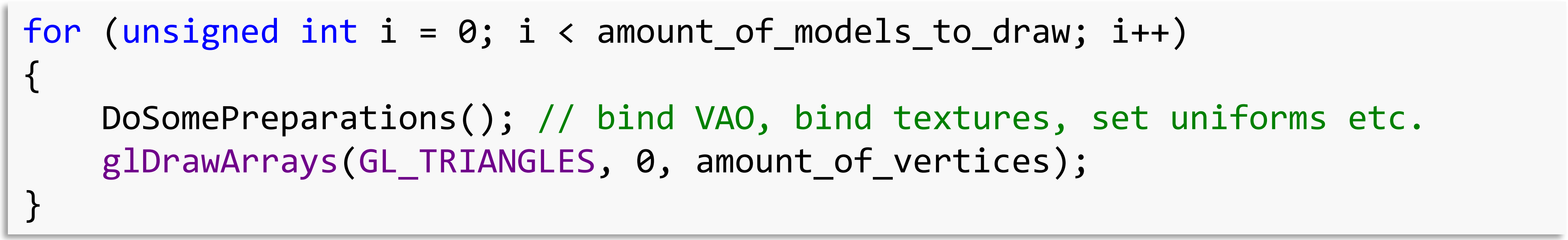
Kai Lawonn

Introduction

- What if we have a scene with a lot of models containing the same set of vertex data, but with different world transformations
- E.g., a scene with grass leaves (small model consisting of only a few triangles)
- Scene might end up with thousands of grass leaves that you need to render each frame
- Each leaf consists of few triangles the leaf is rendered almost instantly, but thousands of render calls drastically reduce performance

Introduction

- Render such a large amount of objects will look like this:



```
for (unsigned int i = 0; i < amount_of_models_to_draw; i++)
{
    DoSomePreparations(); // bind VAO, bind textures, set uniforms etc.
    glDrawArrays(GL_TRIANGLES, 0, amount_of_vertices);
}
```

Introduction

- When drawing many instances of your model like this you'll quickly reach a performance bottleneck because of the many drawing calls
- Telling the GPU to render vertex data with glDrawArrays or glDrawElements reduce performance (necessary preparations, e.g., telling the GPU which buffer to read data from, where to find vertex attributes and all this over the relatively slow CPU to GPU bus)
- Even rendering vertices is super fast, giving your GPU the commands to render them isn't

Introduction

- Better: send data over to the GPU once and then tell OpenGL to draw multiple objects with a single drawing call using this data → instancing
- Instancing can draw many objects at once with a single render call → saving the CPU -> GPU communications
- Render using instancing change:
  - glDrawArrays → glDrawArraysInstanced
  - glDrawElements → glDrawElementsInstanced
- These functions take an extra parameter: number of instances to render
- Sent all data to the GPU only once, then tell the GPU how it should draw all these instances with a single call
- The GPU then renders all these instances without having to continually communicate with the CPU

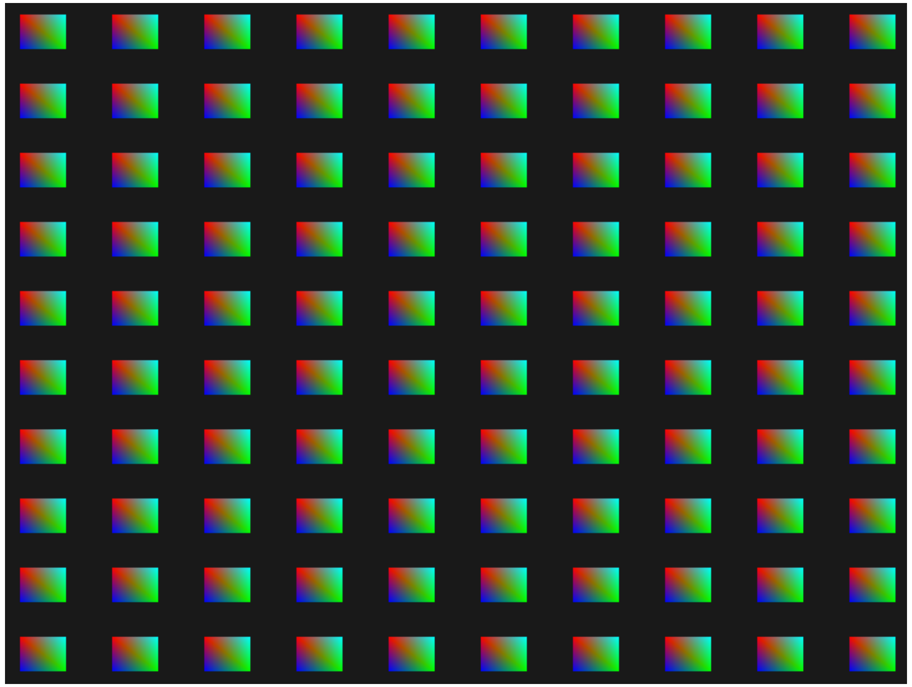
Introduction

- By itself this function is a bit useless
- Rendering the same object a thousand times is of no use (objects are rendered at the same location → we would only see one object)
- For this reason GLSL embedded another built-in variable in the vertex shader called gl_InstanceID
- Drawing with instanced rendering calls, gl_InstanceID is incremented for each instance being rendered starting from 0
- Unique value per instance means we could index into a large array of position values → position each instance at a different location

Example: Quads

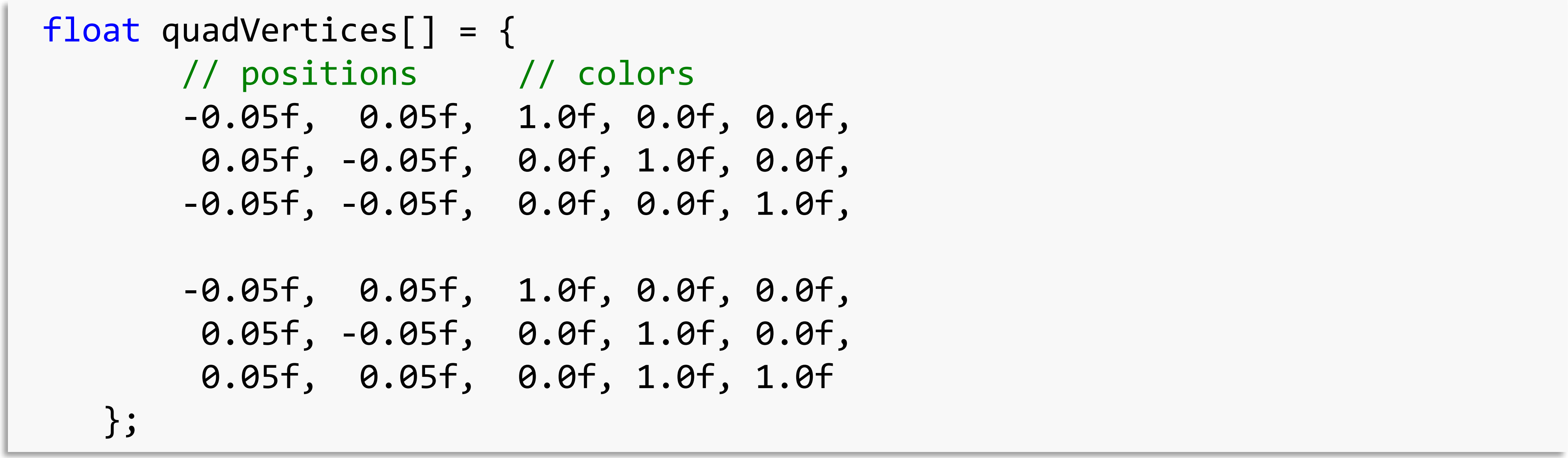
Example: Quads

- Example: render a hundred 2D quads in NDCs with just one render call
- Add a small offset to each instanced quad by indexing a uniform array of 100 offset vectors
- The result is a neatly organized grid of quads that fill the entire window:



Example: Quads

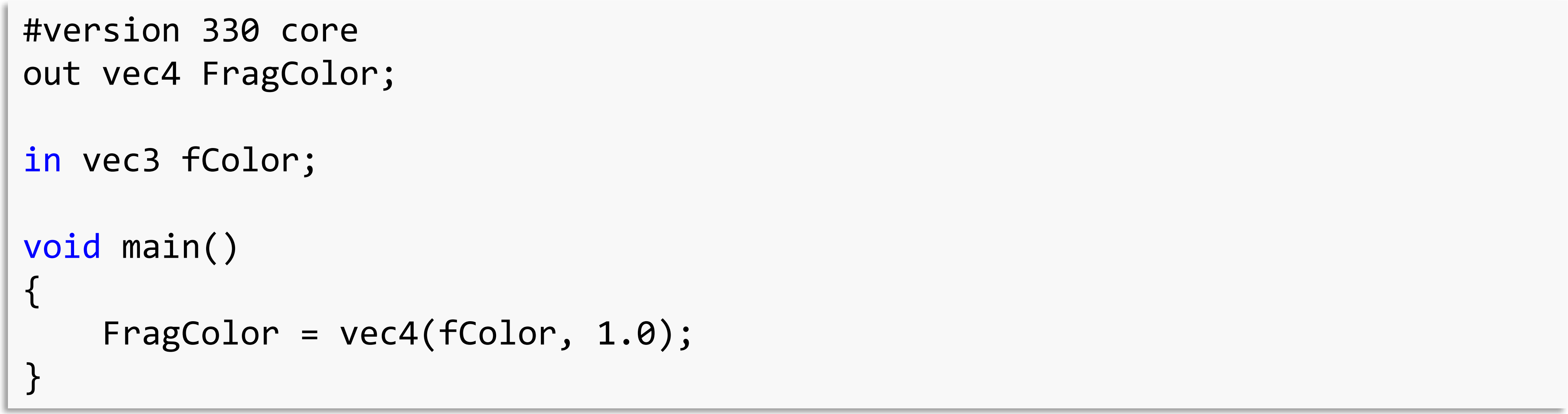
- Each quad consists of 2 triangles with a total of 6 vertices
- Each vertex contains a 2D NDC position vector and a color vector
- Triangles are quite small to properly fit the screen in large quantities:



```
float quadVertices[] = {
       // positions     // colors
       -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
   };
```

Example: Quads

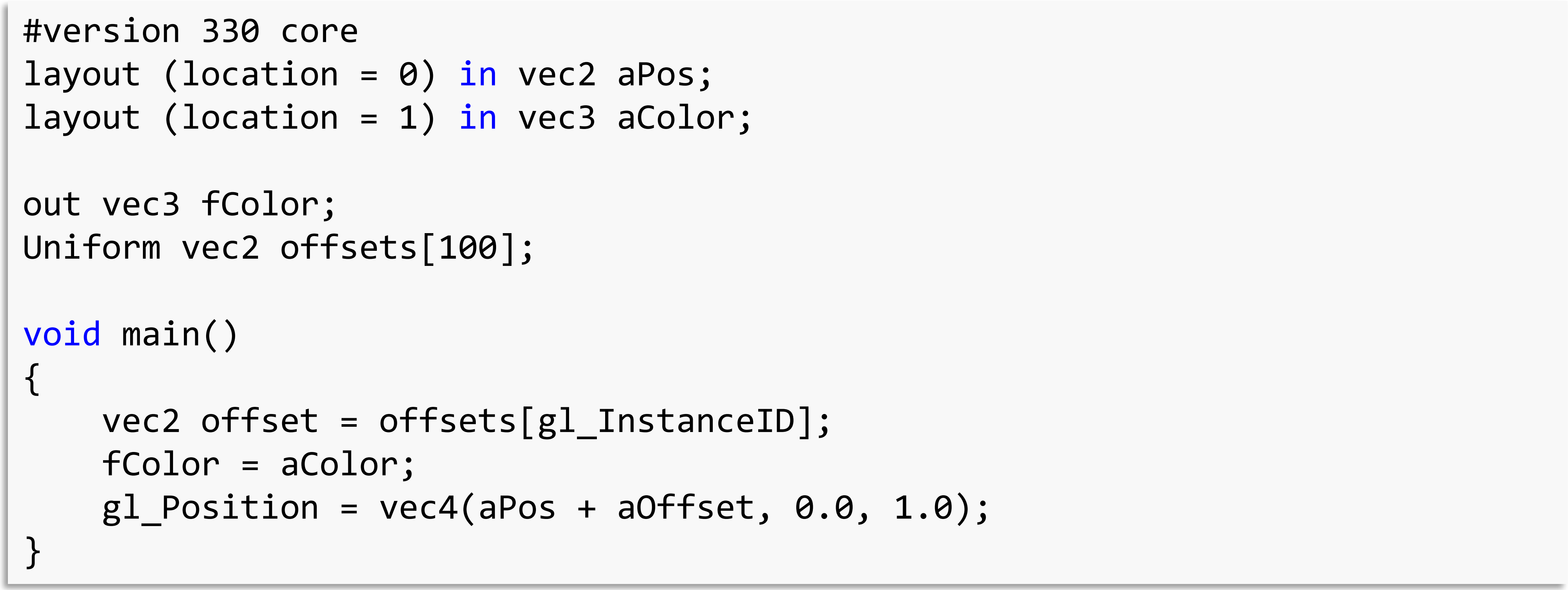
- Colors of the quads are accomplished with the fragment shader (receives a color from vertex shader and sets it as its color output):



```
#version 330 core
out vec4 FragColor;

in vec3 fColor;

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

Example: Quads

- At the vertex shader it’s starting to get interesting:



```
#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];
    fColor = aColor;
    gl_Position = vec4(aPos + aOffset, 0.0, 1.0);
}
```

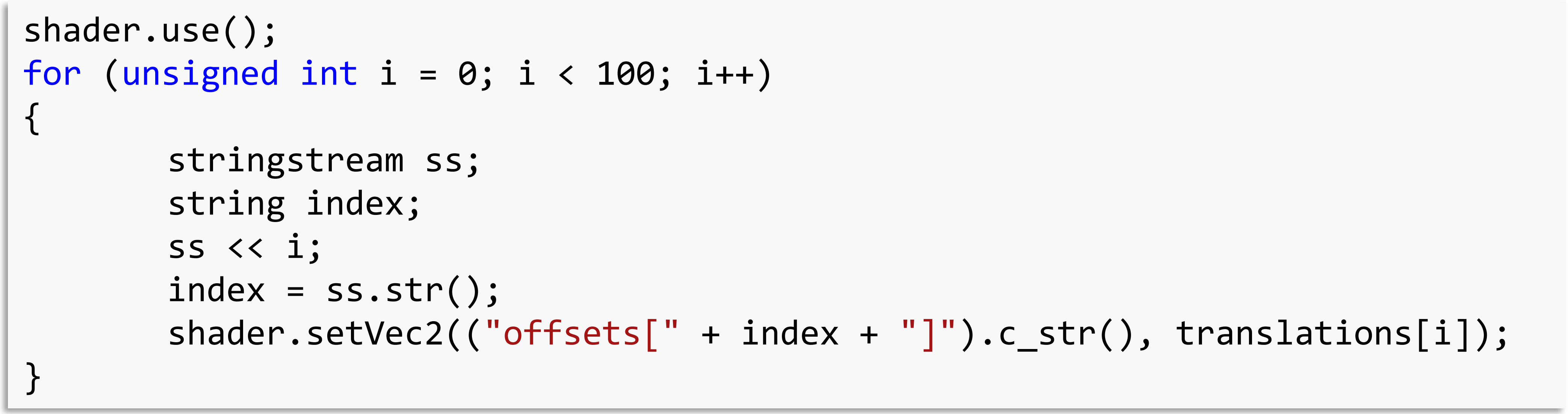
Example: Quads

- Set the offset positions that we calculate in a nested for-loop before the game loop:

```
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;
    }
}
```

Example: Quads

- Aside from generating the translations array, need to transfer the data to the vertex shader's uniform array:



```
shader.use();
for (unsigned int i = 0; i < 100; i++)
{
        stringstream ss;
        string index;
        ss << i;
        index = ss.str();
        shader.setVec2(("offsets[" + index + "]").c_str(), translations[i]);
}
```

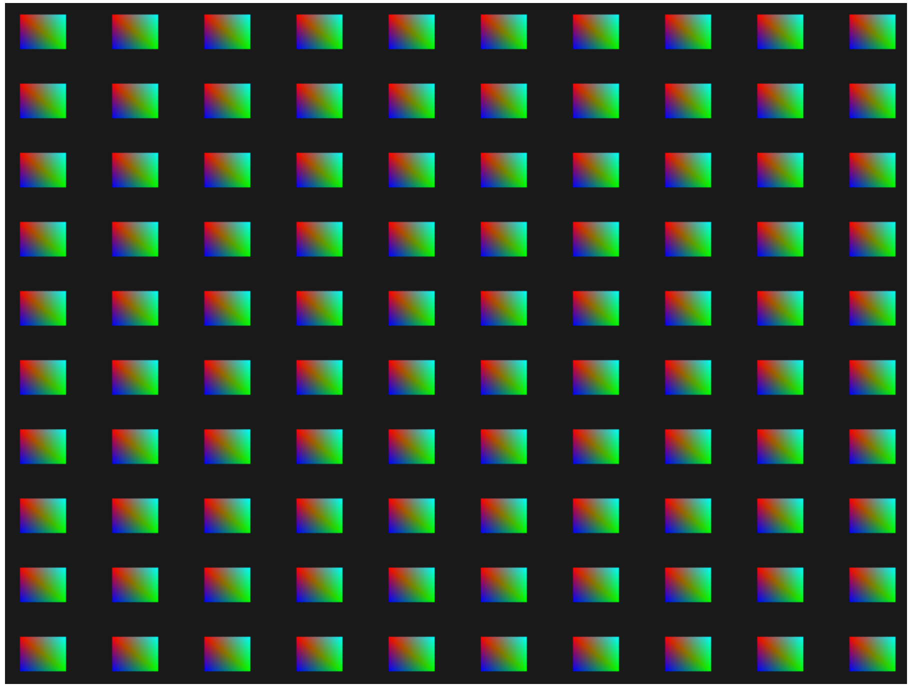
Example: Quads

- Preparations are finished → start rendering the quads
- To draw call glDrawArraysInstanced or glDrawElementsInstanced
- Since we’re not using an element index buffer we’re going to call the glDrawArrays version:

```
glBindVertexArray(quadVAO);
glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100); // 100 triangles of 6
                                                // vertices each
```

F5…

- … colorful quads



Instanced Arrays

Introduction

- Previous implementation works fine for this specific use case, but rendering a lot more than 100 instances will eventually hit a limit on the amount of uniform data we can send to the shaders
- Another alternative is called instanced arrays that is defined as a vertex attribute (allowing us to store much more data) that is only updated whenever the vertex shader renders a new instance

Introduction

- With vertex attributes, the vertex shader will cause GLSL to retrieve the next set of vertex attributes that belong to the current vertex
- When defining a vertex attribute as an instanced array, vertex shader only updates the content of the vertex attribute per instance instead of per vertex
- Allows to use the standard vertex attributes for data per vertex and use the instanced array for storing data that is unique per instance

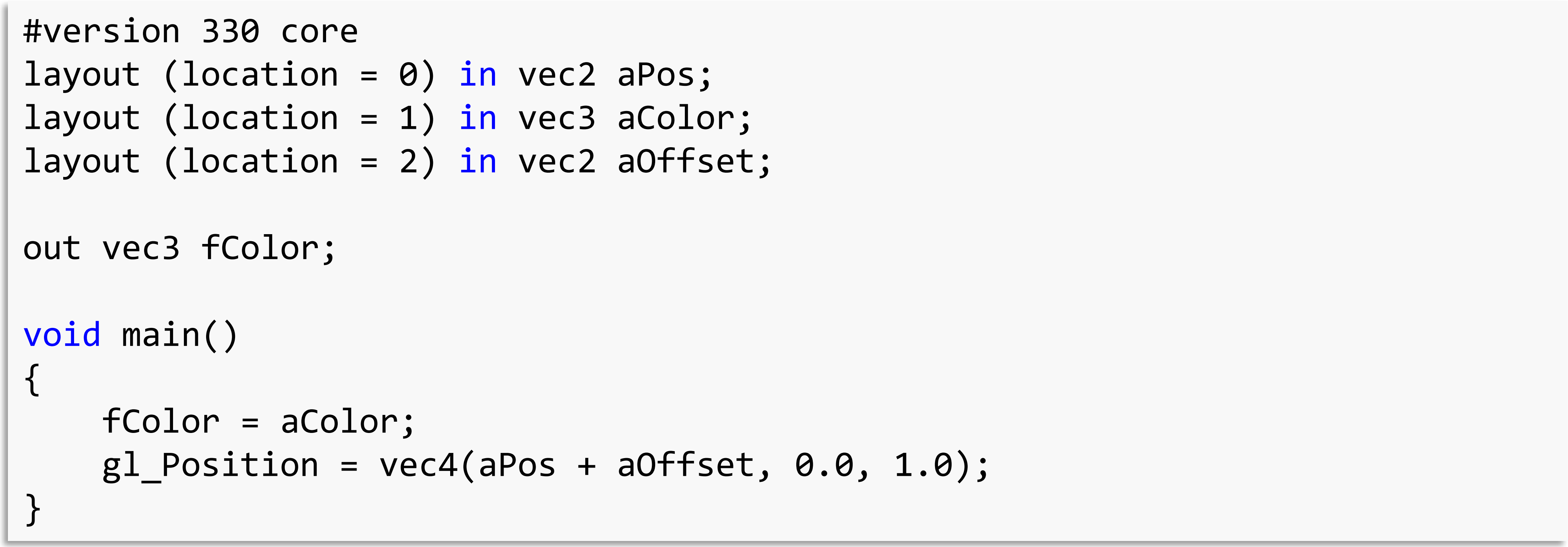
Instanced Arrays

- Example instanced array: represent the offset uniform array as an instanced array
- Update the vertex shader by adding another vertex attribute:

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

Instanced Arrays

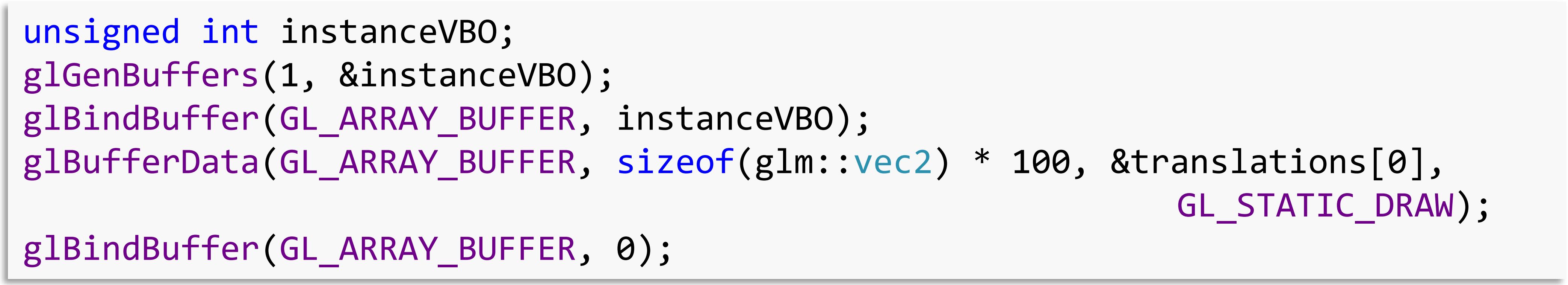
- No longer use gl_InstanceID and can directly use the offset attribute without first indexing into a large uniform array:



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

Instanced Arrays

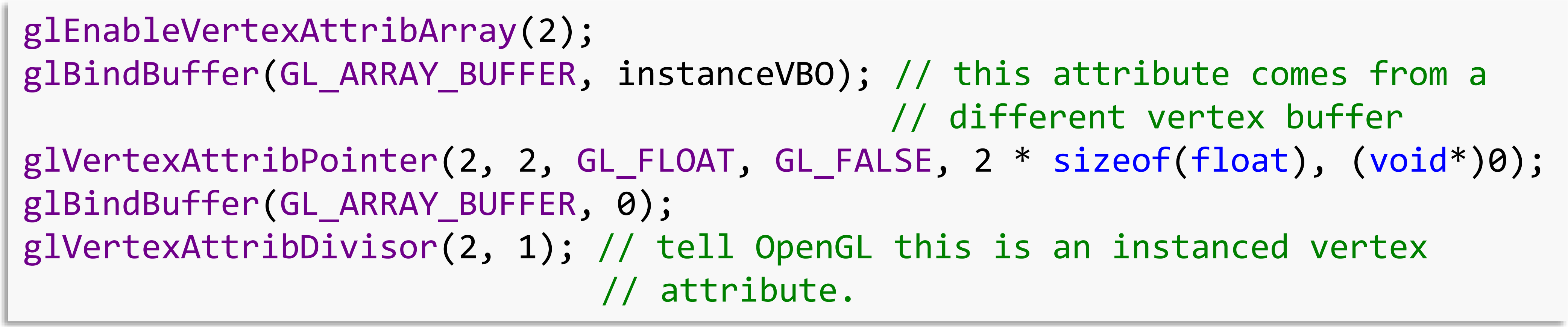
- Instanced array is a vertex attribute, need to store its content in a VBO and configure its attribute pointer
- First, store the translations array in a new buffer object:



```
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);
```

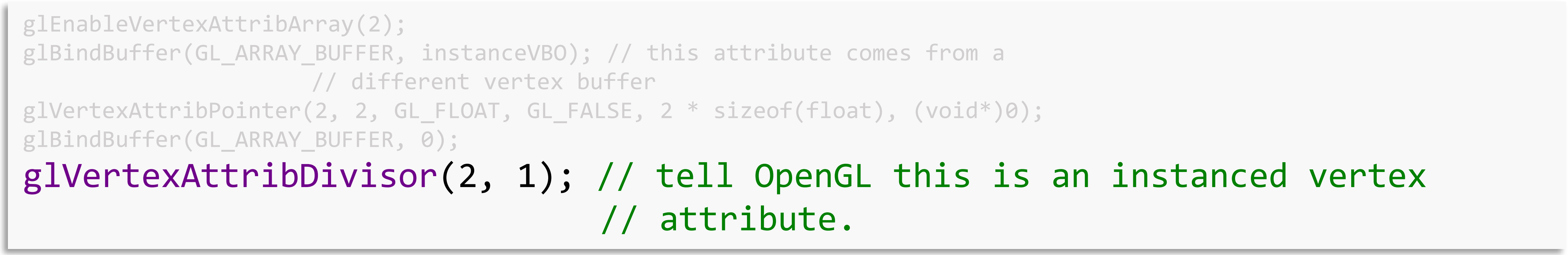
Instanced Arrays

- Then, set its vertex attribute pointer and enable the vertex attribute:



```
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO); // this attribute comes from a
                                            // different vertex buffer
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glVertexAttribDivisor(2, 1); // tell OpenGL this is an instanced vertex
                             // attribute.
```

Instanced Arrays

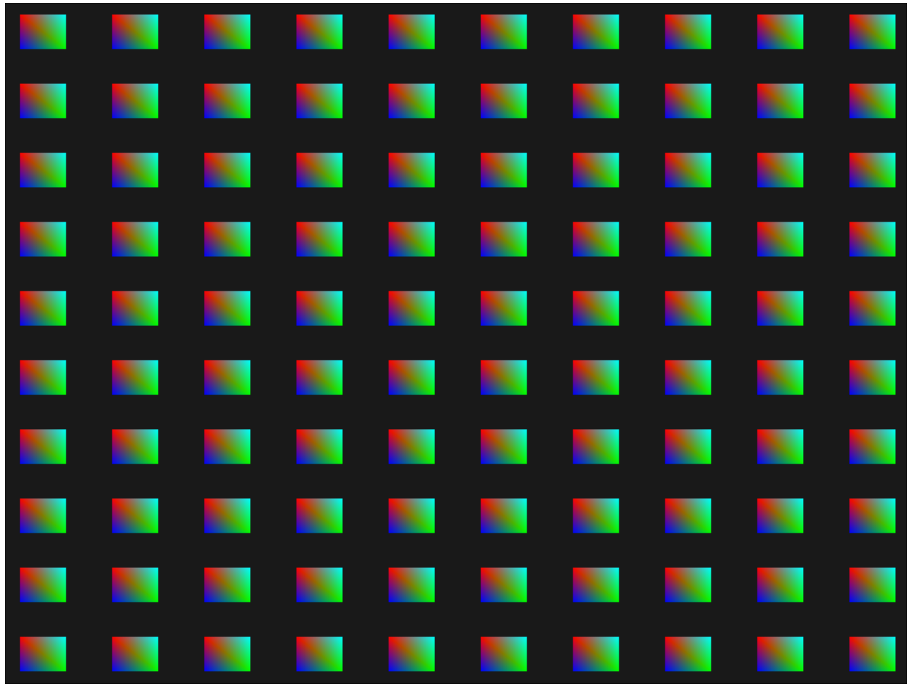


```
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO); // this attribute comes from a
                      // different vertex buffer
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), (void*)0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glVertexAttribDivisor(2, 1); // tell OpenGL this is an instanced vertex
                             // attribute.
```

- glVertexAttribDivisor tells OpenGL when to update the content of a vertex attribute to the next element: 1st parameter - vertex attribute in question; 2nd the attribute divisor
- Default attribute divisor = 0 (update the content (vertex attribute) each iteration of the v. shader)
- Setting this to 1, want to update the content of the vertex attribute when we start to render a new instance
- Setting it to 2, update the content every 2 instances and so on
- Setting it to 1, telling OpenGL that the vertex attribute at attribute location 2 is an instanced array

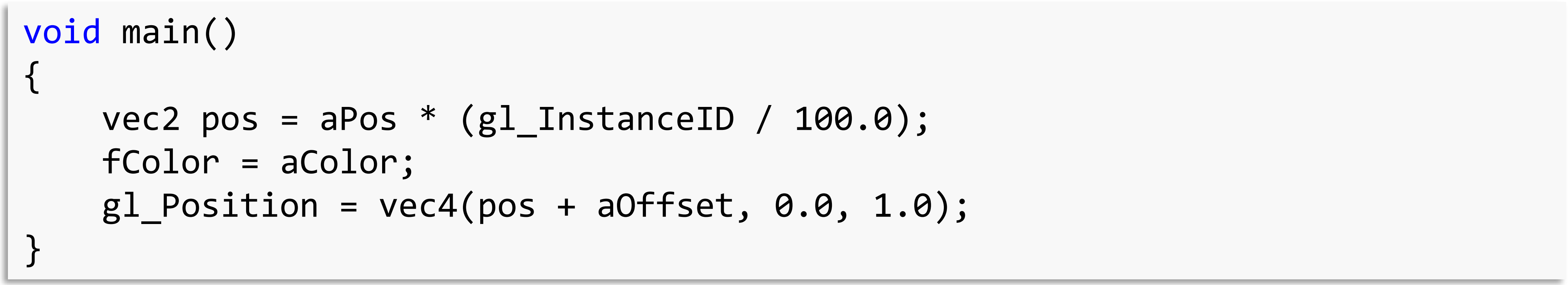
F5…

- … colorful quads (again)



Instanced Arrays

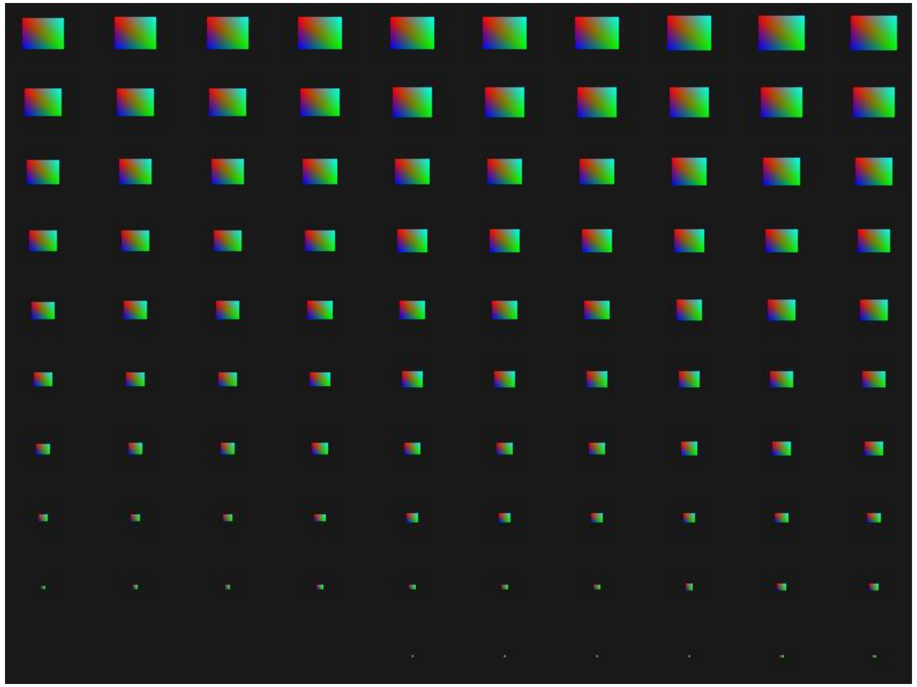
- For fun we could also slowly downscale each quad from top-right to bottom-left using gl_InstanceID again:



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

F5…

- … colorful quads (again), but sized down



Example: Asteroid Field

Introduction

- Imagine a scene where we have one large planet at the center of a large asteroid ring
- Such an asteroid ring could contain thousands of rock formations and quickly becomes un-renderable on any decent graphics card
- This scenario proves itself particularly useful for instanced rendering, since all the asteroids can be represented using a single model
- Each single asteroid then contains minor variations using a transformation matrix unique to each asteroid

Introduction

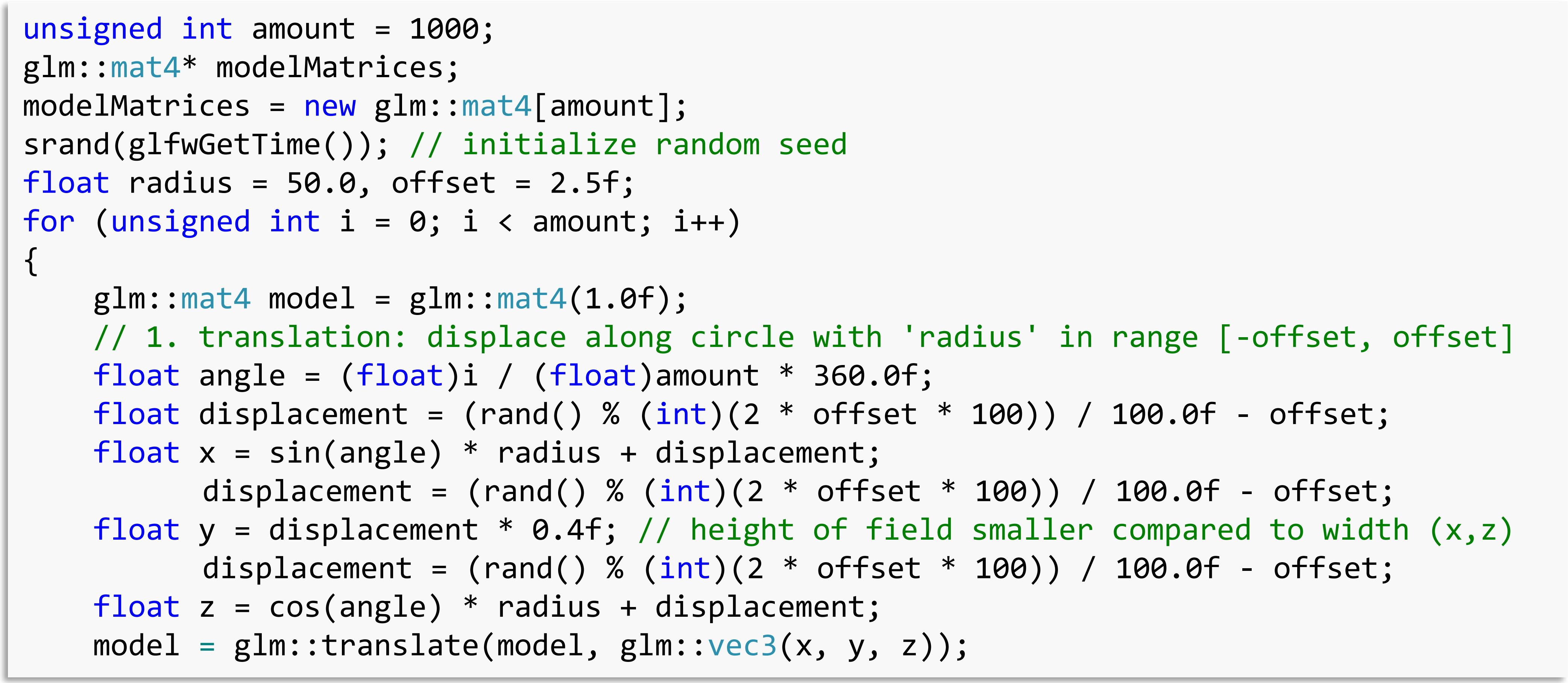
- To demonstrate the impact of instanced rendering, first render a scene of asteroids flying around a planet without instanced rendering
- Within the code samples we load the models using the model loader

Asteroid Field

- Start generating a model matrix for each asteroid
- Translating the rock somewhere in the asteroid ring - add random displacement value to make the ring look more natural
- Then a random scale and a random rotation around a rotation vector
- Results in a transformation matrix; each asteroid placed somewhere around the planet with unique look

Asteroid Field

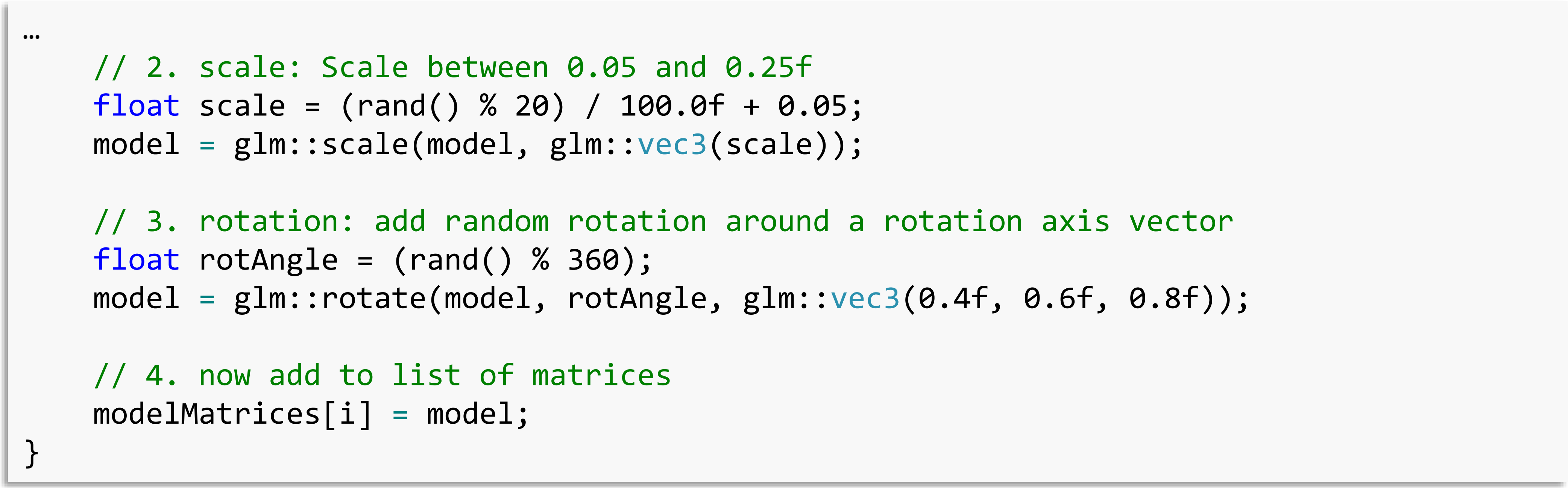
- Ring full of asteroids where each asteroid looks different to the other



```
unsigned int amount = 1000;
glm::mat4* modelMatrices;
modelMatrices = new glm::mat4[amount];
srand(glfwGetTime()); // initialize random seed
float radius = 50.0, offset = 2.5f;
for (unsigned int i = 0; i < amount; i++)
{
    glm::mat4 model = glm::mat4(1.0f);
    // 1. translation: displace along circle with 'radius' in range [-offset, offset]
    float angle = (float)i / (float)amount * 360.0f;
    float displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float x = sin(angle) * radius + displacement;
          displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float y = displacement * 0.4f; // height of field smaller compared to width (x,z)
          displacement = (rand() % (int)(2 * offset * 100)) / 100.0f - offset;
    float z = cos(angle) * radius + displacement;
    model = glm::translate(model, glm::vec3(x, y, z));
```

Asteroid Field

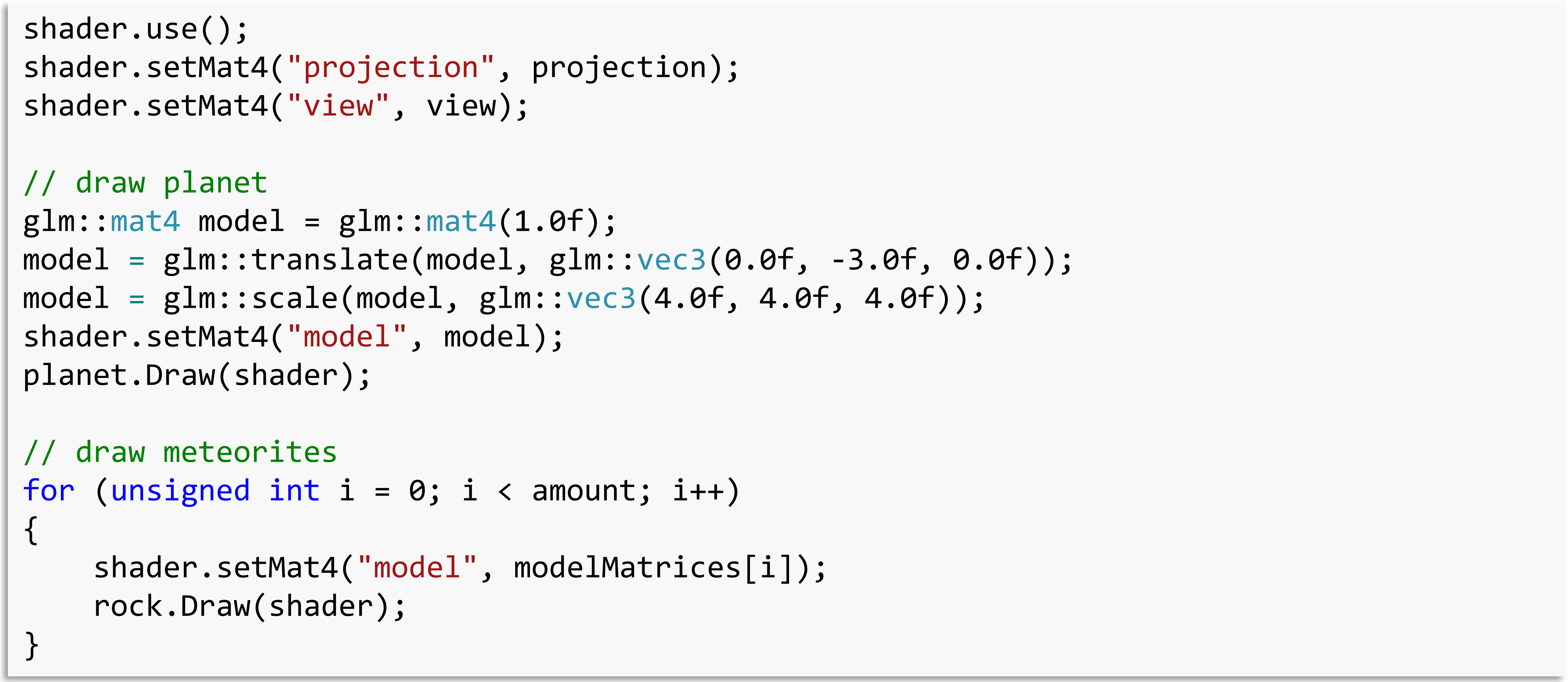
- Ring full of asteroids where each asteroid looks different to the other



```
…
    // 2. scale: Scale between 0.05 and 0.25f
    float scale = (rand() % 20) / 100.0f + 0.05;
    model = glm::scale(model, glm::vec3(scale));

    // 3. rotation: add random rotation around a rotation axis vector
    float rotAngle = (rand() % 360);
    model = glm::rotate(model, rotAngle, glm::vec3(0.4f, 0.6f, 0.8f));

    // 4. now add to list of matrices
    modelMatrices[i] = model;
}
```

Asteroid Field

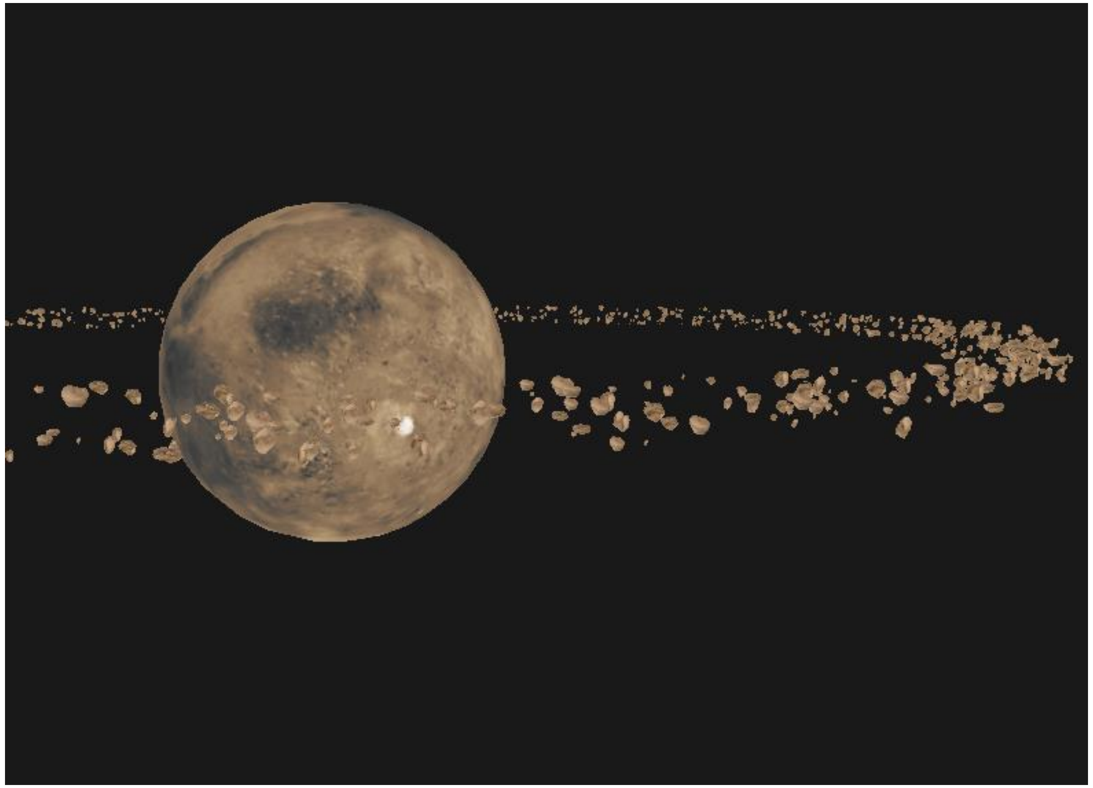
- After loading the planet and rock models and compiling a set of

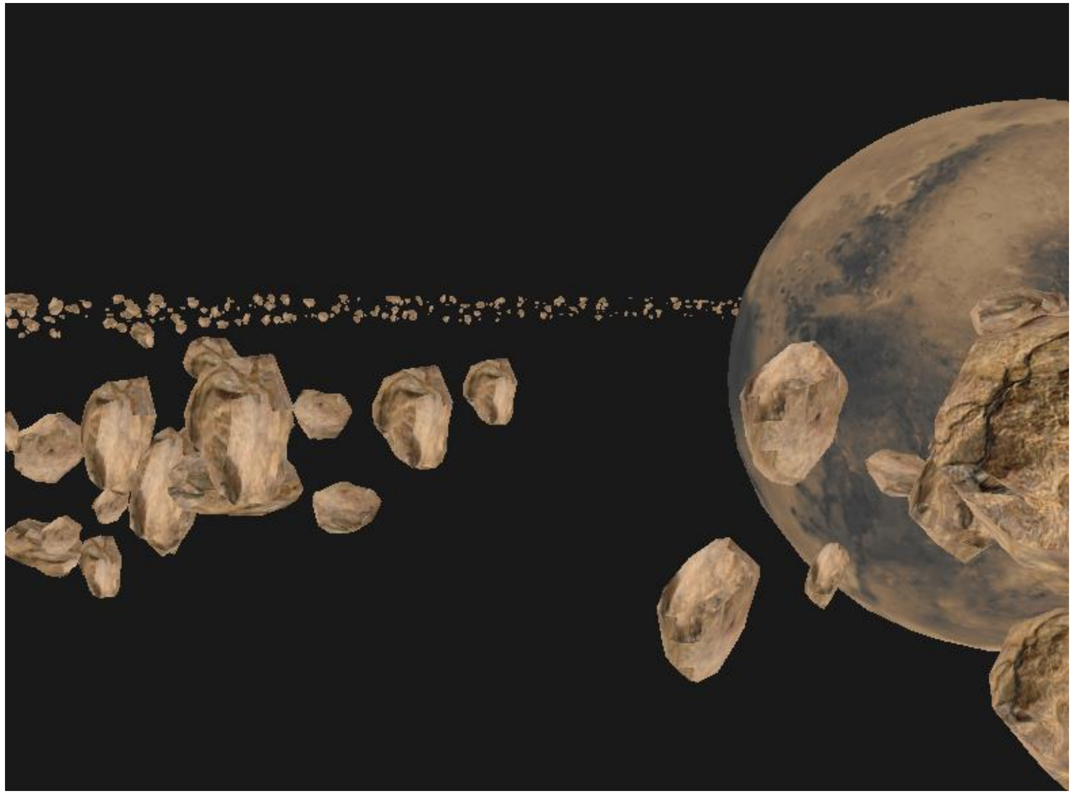


```
shader.use();
shader.setMat4("projection", projection);
shader.setMat4("view", view);

// draw planet
glm::mat4 model = glm::mat4(1.0f);
model = glm::translate(model, glm::vec3(0.0f, -3.0f, 0.0f));
model = glm::scale(model, glm::vec3(4.0f, 4.0f, 4.0f));
shader.setMat4("model", model);
planet.Draw(shader);

// draw meteorites
for (unsigned int i = 0; i < amount; i++)
{
    shader.setMat4("model", modelMatrices[i]);
    rock.Draw(shader);
}
```

F5…

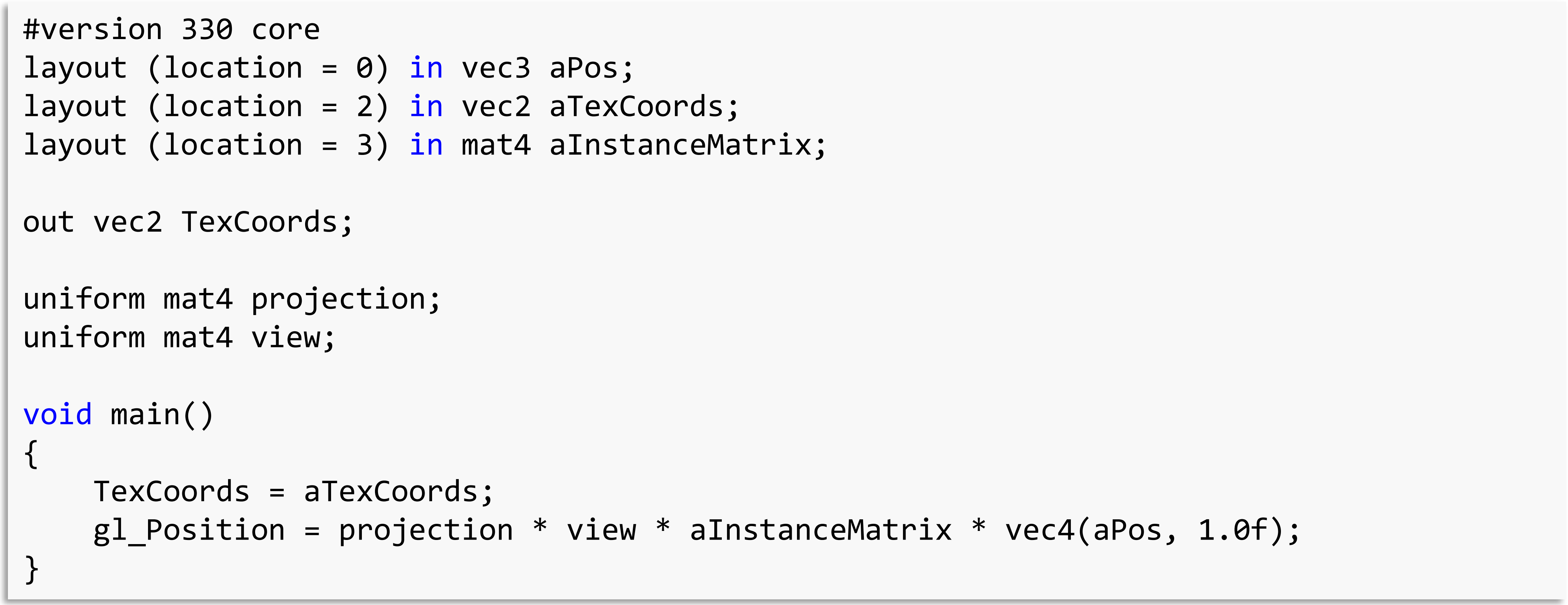
- … space-like scene where we can see a natural-looking asteroid ring around a planet:





Instanced Asteroid Field

- This time: instanced rendering
- Adapt the vertex shader a little:



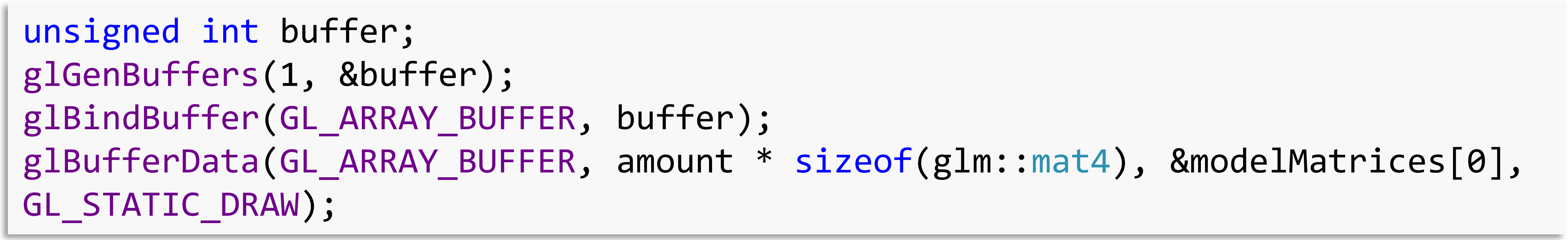
```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 2) in vec2 aTexCoords;
layout (location = 3) in mat4 aInstanceMatrix;

out vec2 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aTexCoords;
    gl_Position = projection * view * aInstanceMatrix * vec4(aPos, 1.0f);
}
```

Instanced Asteroid Field

- When declaring a datatype as a vertex attribute that is greater than a vec4 things work a bit differently
- The maximum amount of data allowed as a vertex attribute is equal to a vec4
- mat4 is basically 4 vec4s → reserve 4 vertex attributes for this specific matrix
- Because we assigned it a location of 3, the columns of the matrix will have vertex attribute locations of 3, 4, 5 and 6

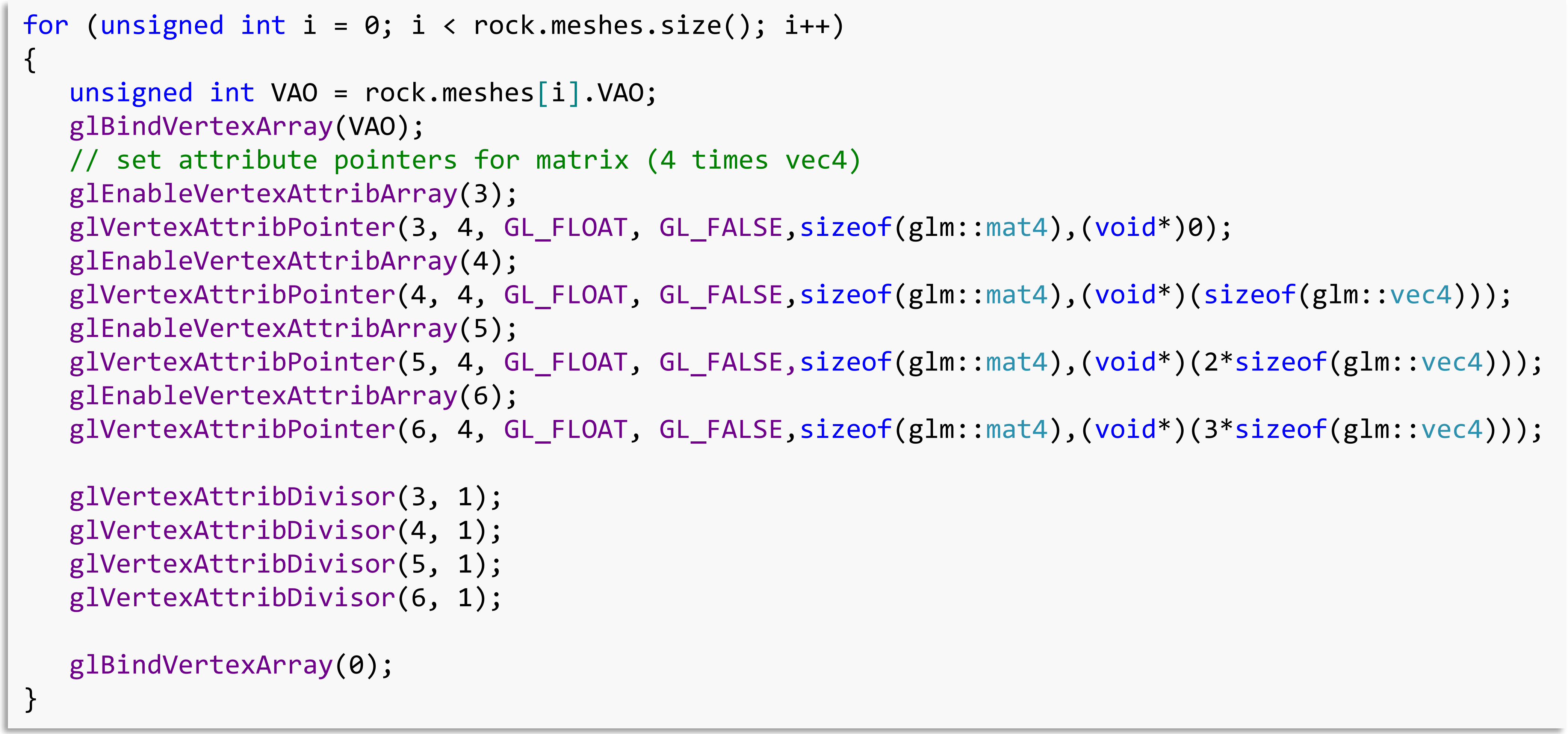
Instanced Asteroid Field

- Set each of the attribute pointers of those 4 vertex attributes and configure them as instanced arrays:



```
unsigned int buffer;
glGenBuffers(1, &buffer);
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, amount * sizeof(glm::mat4), &modelMatrices[0],
GL_STATIC_DRAW);
```

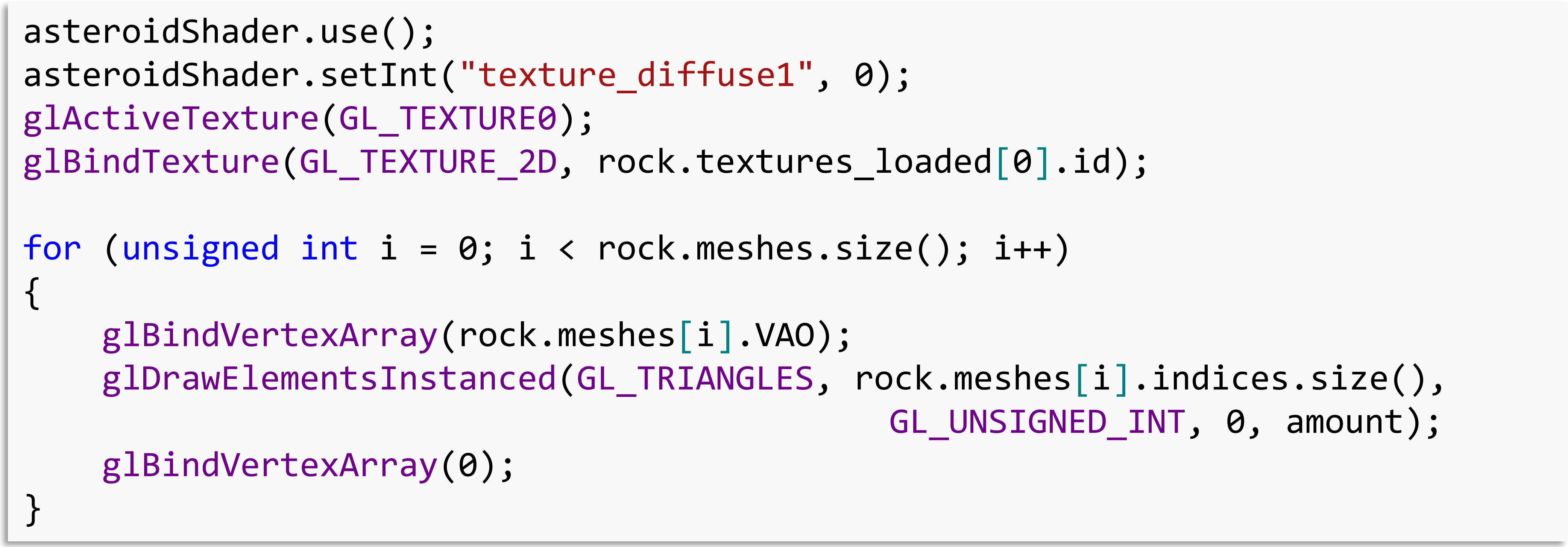
Instanced Asteroid Field



```
for (unsigned int i = 0; i < rock.meshes.size(); i++)
{
    unsigned int VAO = rock.meshes[i].VAO;
    glBindVertexArray(VAO);
    // set attribute pointers for matrix (4 times vec4)
    glEnableVertexAttribArray(3);
    glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE,sizeof(glm::mat4),(void*)0);
    glEnableVertexAttribArray(4);
    glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE,sizeof(glm::mat4),(void*)(sizeof(glm::vec4)));
    glEnableVertexAttribArray(5);
    glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE,sizeof(glm::mat4),(void*)(2*sizeof(glm::vec4)));
    glEnableVertexAttribArray(6);
    glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE,sizeof(glm::mat4),(void*)(3*sizeof(glm::vec4)));

    glVertexAttribDivisor(3, 1);
    glVertexAttribDivisor(4, 1);
    glVertexAttribDivisor(5, 1);
    glVertexAttribDivisor(6, 1);

    glBindVertexArray(0);
}
```

Instanced Asteroid Field

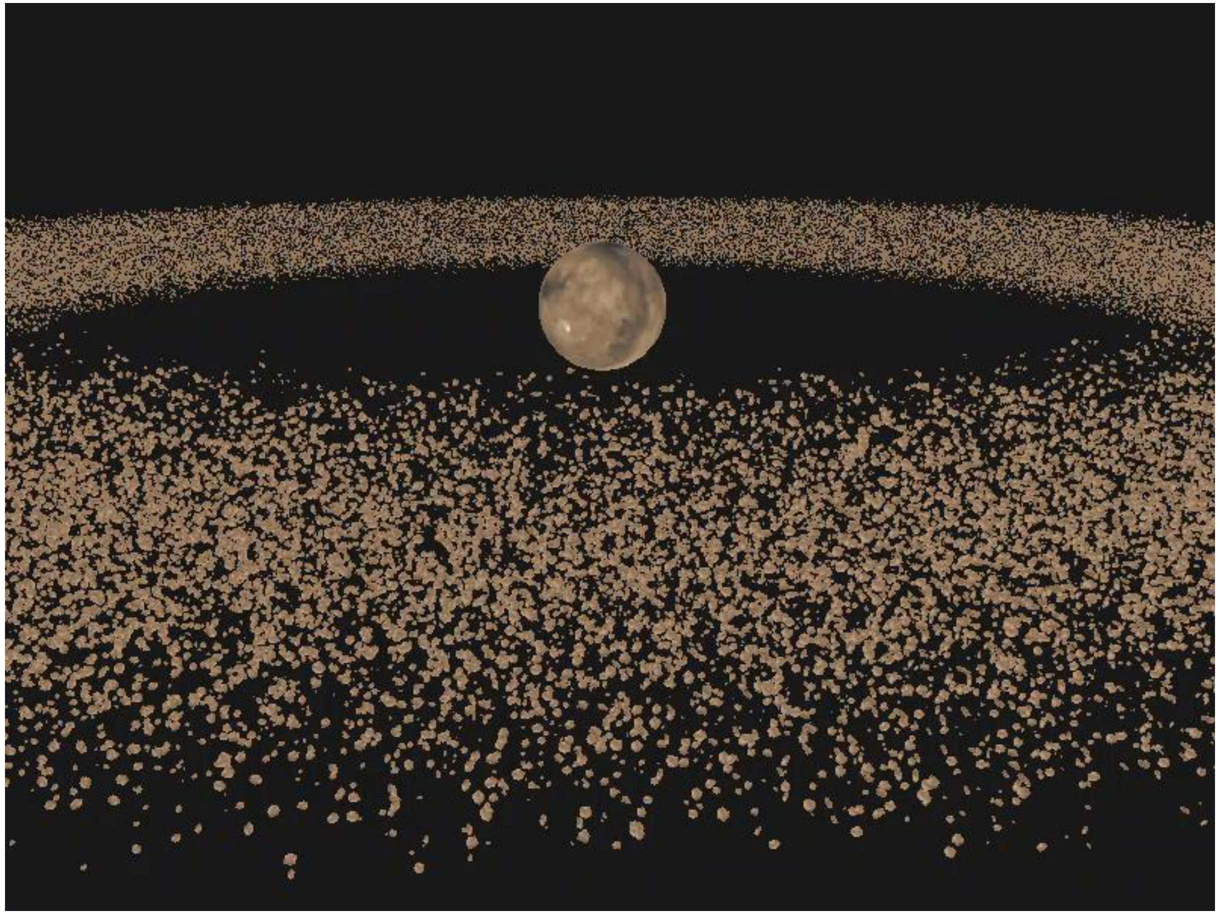
- Take the VAO of the meshes again and this time draw using glDrawElementsInstanced:



```
asteroidShader.use();
asteroidShader.setInt("texture_diffuse1", 0);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, rock.textures_loaded[0].id);

for (unsigned int i = 0; i < rock.meshes.size(); i++)
{
    glBindVertexArray(rock.meshes[i].VAO);
    glDrawElementsInstanced(GL_TRIANGLES, rock.meshes[i].indices.size(),
                                            GL_UNSIGNED_INT, 0, amount);
    glBindVertexArray(0);
}
```

F5…

- … 100.000 smoothly rendered asteroids



Questions???