OpenGL - Instanced Waving Grass

Thomas Ovesen Markussen - tovm@itu.dk

May 14, 2024

[KGGRPRG1KU] - Spring 2024

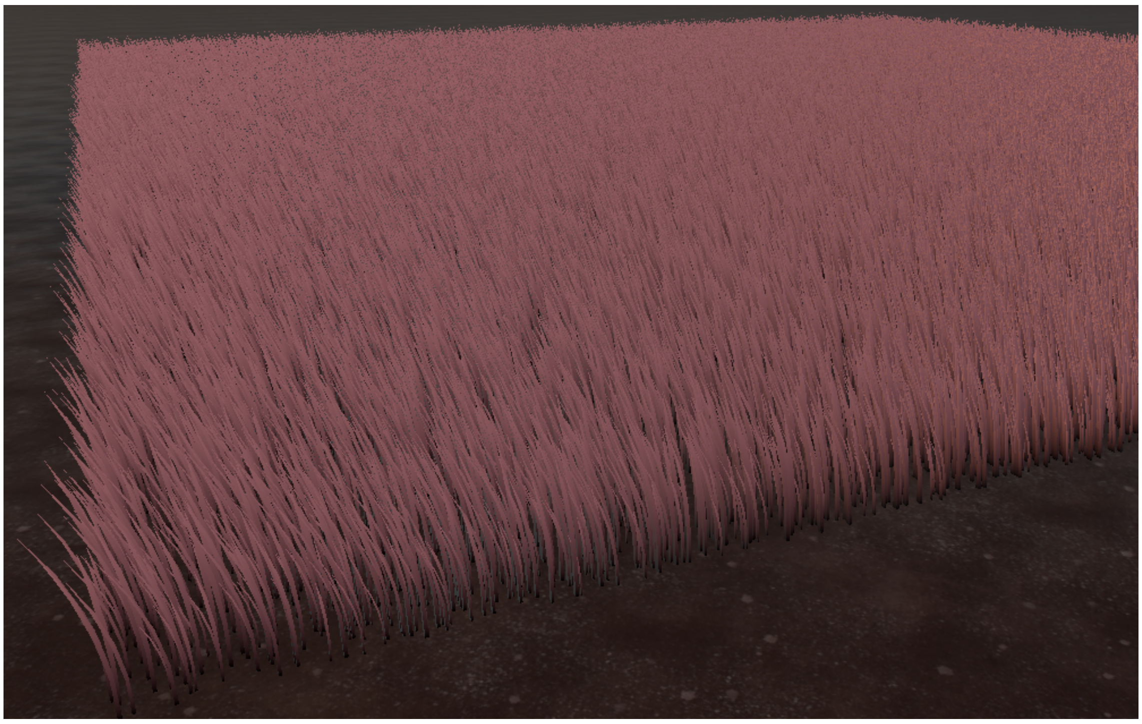

Figure 1: My Grass

1 Motivation

Foliage can make or break the environment in games. Some of my favorite implementations of grass have been done in a stylized way, which my approach has been deeply inspired by. One blatant inspiration has been *Zelda: Breath of the Wild*, which, in my opinion, owes much of its visual appeal to the grass shader that is used. For many years, simple terrains with 2D grass textures were the norm in game worlds, but this approach looks very bland in present times, and as such the importance of a proper grass shader has become that bit more prevalent. I have used the game engine Unity for more than three years now, and many of my games involve forests and open fields. My first naive implementation of grass involved placing grass meshes in mass across these areas, and this does work wonders for the feel and look of the game, but the performance is horrendous. Therefore, I wanted to research how grass could be implemented without slowing down my game, and I wanted to understand the relationship between how the CPU and GPU communicates, such that foliage didn't become a bottleneck for my worlds.

2 Problem statement

How can a customizable field of grass be rendered using OpenGL?

As mentioned in the motivation, performance was something I considered was of utmost importance for my project. My approach, of course, is still very poorly optimized in comparison to what a proper game studio or researchers can produce, but I feel it is more than enough for the scope of this project. While researching the topic of rendering foliage, it was obvious that there are countless ways of optimizing grass, but *instancing* was something that was feasible for me, and that could still improve the performance of my application tenfold. This report will describe the approach of using instancing for grass rendering; how it was implemented; what worked and didn't work, and how I made my grass customizable at the same time.

3 Settings

My project contains some interesting variables that can be tweaked to alter the appearance of the grass. The built-in User Interface of the application contains the uniforms seen below, such that they can be configured during runtime. Table 2 contains variables in the code that is used for instantiating and for this scope was not included as GUI.

Uniforms	Descriptions
Wind Strength	How much the grass bends
Sway Frequency	How fast the grass sways
Bottom Color	The bottom color of the grass
Top Color	The top color of the grass
Gradient Bias	Determines the gradient transition

Table 1: GUI

Variables	Descriptions
grassCount	How many grass models are instantiated
grassPosBounds	A vec3 of the bounds in which grass will spawn
grassRotBounds	Not implemented
grassScaleBounds	A vec2 defining the min and max scale value of grass

Table 2: Code

4 Implementation details

4.1 Application

The project is built upon existing architecture from exercise 9 of this course. This exercise focused upon post processing with features such as *bloom*, *contrast*, and *hue saturation*, but it also contained all the code infrastructure that was provided by the previous exercises. This proved useful, as I didn't have to implement a sky box, model loader, or a lighting model, such that I could focus on the task at hand: grass - and lots of it.

I had - and still to this day have - no experience in 3D modelling software, such that I could easily create the grass mesh that I needed. I do however possess a lot of knowledge regarding Unity, and I have used their ProBuilder plugin on commercial projects before, so that was how I created my grass mesh and texture. It took many hours of figuring out how a `.mtl` file works, but the first step of my project was done; I had a single blade of green grass and a plane with a dirt texture on it.

Instancing came next, and I spend a lot of time reading the existing code to find the easiest way of implementing it without directly calling OpenGL methods, but with some help from Gabriel, I got the instancing to work. The first step was swapping out the OpenGL method that the Drawcall would use for rendering.

```
    glDrawElementsInstanced(primitive, m_count, static_cast<GLenum>(m_eboType),
    basePointer + m_first, m_instanceCount);
```

Listing 1: Drawcall::Draw

By default, the Drawcall object used `glDrawElements`, but I replaced it with the instanced variant. Then I added the variable `m_instanceCount` to the class with an instantiated value of `1`, such that all my meshes could utilize instancing if it was applicable. The variable `m_instanceCount` then gets injected from the `ModelLoader.cpp` to the `DrawCall` object at initialization using a Set method. If the scope of this course favored software architecture, I would probably have created an abstract factory for providing a model loader and mesh, and then used a Builder to configure them, but this approach worked great for this project.

By leveraging instancing, we effectively reduce the amount of draw calls from the CPU to the GPU. This is significant in that it is an extensive operation to send information back and forth between the two. It is more convenient to pack a container ship with 100.000 packages than it is to pack 100.000 sailor boats with 1 package, and the same can be said for CPU to GPU communication. Instancing also utilizes data recycling, and inadvertently utilizes the flyweight pattern known from Computer Science, in the same way that an Element Buffer Object reuses the same indices to save memory. Concurrently, it serves as the simplest way to utilize parallelism in the GPU, such that it can effectively use more of its cores on the task at hand.

```
vao.Bind();
vbo.Bind();

int location = 5;
int offset = 0;
int stride = 4 * attr.GetSize();

for (int i = 0; i < 4; i++)
{
    vao.SetAttribute(location, attr, offset, stride);
    location++;
    offset += attr.GetSize();
    glVertexAttribDivisor(5 + (1 * i), 1);
}

VertexBufferObject::Unbind();
VertexArrayObject::Unbind();
```

Listing 2: PostFXSceneViewerApplication::InitializeInstancing

After having spawned my grass models in the scene, I setup the framework from which I would populate the attributes of my instanced models. I created a new Vertex Buffer Object for the model's submesh, and then inserted a vector of matrices into its vertex data. These matrices held the offset positions for the instanced grass, such that they could populate a field using some simple math and transformations. Then I found the Vertex Array Object of my model and told it how to use that data, as seen in the snippet above. As a matrix of 4 rows and columns contains 16 floats, it would not fit into the attribute due to it having a size constraint of 4 components. Therefore, I had to allocate 4 attributes of 4 float components using offsets, and tell the instanced grass to use different parts of the large vector of matrices using `glVertexAttribDivisor`.

4.2 Shaders

```
vec3 pos = VertexPosition;

// Calculate wind sway effect
float swayAmount = sin(VertexPosition.z * SwayFrequency + Time + random(vec2(
    gl_InstanceID))) * WindStrength * VertexPosition.y;

// Adjust swayAmount based on height of grass
float heightFactor = (1.0 - (1.0 - VertexPosition.y));
swayAmount *= heightFactor;

pos.z += swayAmount;

gl_Position = WorldViewProjMatrix * InstanceMatrix * vec4(pos, 1.0);
FragPosition = VertexPosition;
```

Listing 3: default_grass.vert

The most important shader for my application has been the vertex shader, in that it is responsible for offsetting the grass models from each other. By using the provided `InstanceMatrix` which was created in the last segment, it can move the whole mesh anywhere and a random scale was also added to it, so the grass doesn't look uniform. The wind effect is created using a simple sine function, and some noise is added to offset the wavelengths of the grass a bit from each other using the `random` function and the instance ID of the mesh, as that is the only property that distinguishes them from eachother. To achive the bending effect of the grass, I created a bias based on the height of the current VertexPosition, so that the upper part of the mesh would bend more than the bottom - example below.

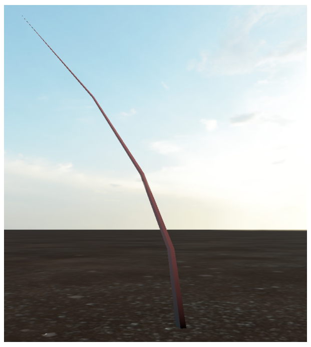

Figure 2: A single blade of grass

The fragment shader is used in combination with the vertex shader to achieve the style that I wanted to create. I pass the position from my vertex shader to my fragment shader, such that I can work with the color at a specified point on the grass blade. A gradient is then calculated using a mix of two colors and a bias, which creates the appealing blending colors using interpolation. The grass material also gets lighting applied by the concurrent deferred lighting pass, which makes it blend into the scene. On the topic of blending, I found that blending the bottom color of the grass to the same color as the terrain created the best looking grass, as it otherwise was blatant where the grass cut into the ground.

5 Future improvements

5.1 Grass Topology

The grass mesh contains 46 triangles which has always been intended. This was done so that the vertex shader could bend the mesh in a smooth motion, but the large amount of triangles are a definite cause of concern for performance. Due to my inexperience with real 3D modelling software, I was unable to optimize the topology of the grass mesh. Given some extensive tweaking, it could be assumed that a mesh with fewer triangles and better topology could optimize the application while keeping the same visual fidelity.

5.2 LODs and Billboarding

Instancing does a lot to help performance by sending information from the CPU to the GPU in large batches instead of in bits (pun intended), but my approach is a naive implementation of the technology in practise. No matter the distance from the camera, the grass will always be rendered at full resolution. To help remedy this I would add levels of detail to the model, which would gradually decrease the mesh's number of polygons at further distances. In addition, it would make sense to use billboarding at large distances which would reduce the grass down to two polygons.

5.3 Bounding Volume Culling

By splitting up the grass into its own volumes, you can check the distance or view frustum of the grass chunks, and thereafter choose whether to cull those volumes on runtime. This would also require some batching optimizations, such at finding the ideal batch size for our instanced grass, which would greatly improve performance as well, but this approach can also be combined with LODs and cross fading techniques to achieve an appropriate compromise between visual fidelity and performance.

5.4 Parallel Programming

Implementing multi-threading and compute shaders would increase the performance of the application by a large margin. By delegating certain tasks such as the wind animation or the position offset calculations to different parts of the GPU or CPU, we can utilize much more of the respective computing power at hand.

6 Bibliography

Gordan, V. (n.d.). Opengl series. https://www.youtube.com/@VictorGordan

Group, K. (n.d.). Opengl wiki. https://www.khronos.org/opengl/wiki/Main_Page

LearnOpenGL. (n.d.). Instancing. https://learnopengl.com/Advanced-OpenGL/Instancing

Renk, T. (2015). From random number to texture - glsl noise functions. http://www.science-and-fiction.org/rendering/noise.html