Graphics Programming
Exam Project

Date: 17/05/2023

Alexander Berg alebe@itu.dk

Course: Graphics Programming
Course code: KGGRPRG1KU

1 Introduction

This project was made as part of the course Graphics Programming. The goal of the project was to simulate grass using OpenGL and C++. Grass is used in many games, especially open world games, to improve visual fidelity and immersion.
Good looking grass presents several challenges which must be overcome. It can be very hardware intensive because it often requires rendering a large number of vertices. Lighting can also be a challenge since real life grass has a lot of transparency which can be difficult to implement properly. This project will render grass efficiently using a technique known as instancing and will use techniques such as shadow mapping and PBR to improve visual quality.

2 Controls and Settings

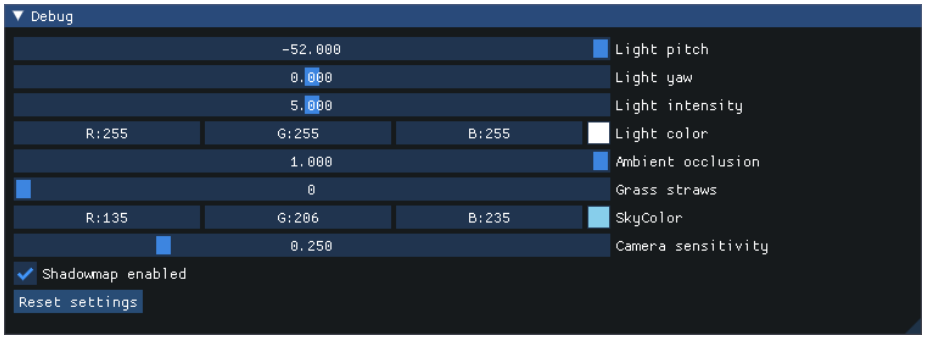

Figure 1: The settings menu

The player can walk around using the WASD keys and look around using the mouse. When the program is started the cursor will be locked. The player can free the cursor by pressing the F key.
The program has a few settings which change the visual appearance of the scene. Some players may have more powerful hardware than others. Therefore, the number of grass straws can be changed, and the shadow map can be toggled as these are the most demanding features. The appearance of the scene can also be changed by tweaking the light direction, color, and intensity.
Note that the default setting for the number of grass straws is 0. Therefore, this setting has to be increased manually for any grass straws to show. The list of settings is as follows:

- The direction of the light (pitch and yaw).
- The intensity of the light.
- The color of the light.

- The intensity of ambient occlusion.
- The number of grass straws.
- The color of the sky.
- The sensitivity of the camera.
- The shadow map can be enabled or disabled.

There is also a button for resetting the settings to default values.
The settings menu can be seen on figure 1.

3 Implementation Details

This section describes the implementation. This includes the additional calls to OpenGL functions as well as how grass is rendered. Due to the time the project was started the implementation does not use the newest version of ItuGL. This means that the parts of ItuGL modified may be different in the current version. In addition, when the project was started ItuGL did not contain functionality for shadow mapping. This has therefore been implemented manually.

3.1 Ground

The scene consists of procedurally generated terrain on which the player can walk. The terrain is generated using Perlin noise. This is very similar to how the terrain was generated in exercise 1 during the course.

3.2 Instancing

Since instancing has not been covered in the course a explanation will first be given. Then the implementation of instancing in ItuGL will be described.

3.2.1 What is Instancing?

Rendering thousands or even millions of meshes can easily become very hardware intensive. When the meshes rendered use the same vertex data the GPU has to repeat the same work many times. In addition, the VBO may be larger than necessary since a lot of data is repeated. This is not optimal. Luckily, OpenGL has a feature for solving this exact problem. This feature is called instancing and can be used to efficiently render the same mesh a large amount of times [1]. Usually, one of the OpenGL functions *glDrawArrays* or *glDrawElements* is called when rendering. However, when instanced rendering is required *glDrawArraysInstanced* or *glDrawElementsInstanced* is used. These functions have the same parameters as *glDrawArrays* and *glDrawElementsInstanced* except with the addition of a parameter called *instanceCount*. This parameter specifies, as the name suggests, how many instances of the mesh (or meshes)

should be drawn. For example, if *instanceCount* is 1 then both functions will behave the same as their non-instanced counterparts. If *instanceCount* is a non-negative integer n then the functions will behave as if their non-instanced counterparts were called n times, except that the GPU performs less work.
At first this does not seem very useful. If *glDrawArraysInstanced* and *glDrawElementsInstanced* are similar to just calling *glDrawArrays* and *glDrawElements* multiple times then the meshes are just going to be rendered on top of each other. This is usually solved by using two types of vertex attributes: Vertex attributes containing the vertex data to be reused, and vertex attributes containing data specific to each instance. The latter are referred to as *instanced arrays*. For example, we may have a normal vertex attribute containing the vertex positions in local space and an instanced array containing positions of individual instances in world space. We would then add these together to get the final position of each vertex in each instance.
In order to tell OpenGL that an attribute is an instanced array we use the function *glVertexAttribDivisor*. This function takes two *GLuint* as parameters: *index* and *divisor*, where

- *index* is the index of the vertex attribute (the same index as the one used in *glVertexAttribPointer* and *glEnableVertexAttribArray*).
- *divisor* is the number of instances that will pass between updates of the vertex attribute (specified by *index*).

If *divisor* is 0 then the attribute will be updated once per vertex (this is the default), otherwise the attribute is an instanced array and the content will be updated for each *divisor* instance.
New function overloads have been added to ItuGL which utilise the OpenGL functions required for instancing.

3.3 Rendering Grass

The changes described above implement instanced rendering in ItuGL. This part will describe how the new features of ItuGL have been used to render grass.
Firstly, the geometry of grass straws has to be specified. This is done using a struct storing the data for a single vertex. The struct consists of a *glm::vec3* position and *glm::vec2* texture coordinate. The *VertexFormat* is specified as follows:

```
1    VertexFormat vertexFormat;
2    vertexFormat.AddVertexAttribute<float>(3); // position
3    vertexFormat.AddVertexAttribute<float>(2); // texCoord
```

The vertex data itself consists of 5 vertices stored in a *std::vector* which is given as argument to *Mesh::AddSubmesh*. Note that the vertex data contains no normals. This is because all the normals are pointing directly upwards and are created in the vertex shader. This is usually the direction normals are pointing when rendering grass and gives good looking results. An *std::vector* containing indices for the EBO is also created.

Secondly, the characteristics of individual straws grass have to be specified. This is done using a struct storing a *glm::vec3* position, a *float* rotation, and a *float* height multiplier. The rotation is used to rotate a grass straw around the y-axis. This is done because grass straws are 2-dimensional and therefore become invisible when viewed from the side. Randomly rotating the grass straws make this problem much less noticeable.
The *VertexFormat* is specified as follows:

```
1    VertexFormat instanceFormat;
2    instanceFormat.AddVertexAttribute<float>(3); // position
3    instanceFormat.AddVertexAttribute<float>(1); // rotation
4    instanceFormat.AddVertexAttribute<float>(1); //
     heightMultiplier
```

The characteristics of each grass straw is stored in a *std::vector*. The contents of this *std::vector* is randomly generated.
Finally, the three *std::vector*s and the *VertexFormat*s are given as argument to one of the *Mesh::AddSubmesh* overloads.

3.4 Lighting and PBR

It can often be difficult to perceive depth in scenes with no lighting. This is also true for the scene implemented in this project. Especially when many grass straws are being rendered. In this case it is difficult to distinguish individual grass straws as everything has the same texture. Therefore, the choice was made to implement light and shadows (as discussed later). This is done using PBR in order to make the lighting as realistic looking as possible.
The PBR implementation is quite simple but still provides far better than Blinn-Phong lighting. For diffuse lighting the Lambertian model is used while for specular lighting the Cook-Torrance model is used. Indirect lighting using ambient occlusion is also implemented. The PBR material for the ground can be found at [2]. The grass texture was provided by the course manager and the textures for ambient occlusion and roughness were created manually. Normal mapping has also been implemented to properly use the PBR textures. The implementation is inspired by [3].

3.5 Deferred Rendering

When rendering a large number of grass straws the fragment shader has to be executed a large number of times. The number of fragment shader invocations is multiplied by the number of lights thus exacerbating the performance impact. Therefore, it is important to make as few fragment shader invodations as possible. On older hardware (or when the fragment shader writes to *gl_FragDepth*) this is even more important as the hardware may lack early depth testing, and will therefore waste resources on calculations which are discarded due to depth testing. Therefore, it was decided that the program should use deferred rendering in order to optimize the use of fragment shaders.
The GBuffer used for deferred rendering consists of four parts:

- A depth texture.
- An RGBA texture with 8 bytes per component used for albedo color.
- An RGBA texture with 16 bytes per component used for normals.
- An RGB texture with 8 bytes per component used for specular lighting.

This means that the GBuffer has a total size of 120 bytes plus depth (this is implementation dependent). Notice that the normal texture has 4 components even though only 3 are needed for normals. The extra component is used to indicate whether indirect specular lighting should be disabled. This is necessary because the grass normals are pointing directly upwards. Under normal circumstances this would cause the indirect specular lighting to have an intensity of 0 when viewed from below. The extra component is therefore used to disable indirect specular lighting for grass straws.
In the final version of the program the scene only contains a single directional light. Deferred rendering therefore not actually needed. It is only part of the program because it was initially thought that multiple lights would be used.

3.6 Shadow Mapping

As previously mentioned shadows were implemented to help the player perceive depth and to improve visual quality. This was done by adding a new class, called *LightRenderPass*, inheriting from *RenderPass*. The *LightRenderPass::Render* member function renders the scene at once for each light. If a pointlight is used then the scene is rendered 6 times in order to get shadows in all directions. Only depth is rendered as color is not needed for this pass. Each light has its own framebuffer which is bound before rendering, and a depth texture storing the resulting depth values. Each light also has a *glm::ivec2* specifying the resolution of the depth texture. This is used in a call to *glViewport* before rendering.
The implementation also takes care of shadow acne by using a bias value and prevents uses a simple PCF implementation to reduce jagged edges on shadows. Since only direct lighting casts shadows shadow mapping has been implemented such that indirect lighting is not affected. The implementation of shadow mapping is very similar to the one described by [4].

4 Final Result

The final result can be seen on figure 2. This shows the scene from the starting position and default settings. In this case, no grass is rendered as it defaults to not rendering grass. Using the debug menu the number of grass straws can be increased. Figures 3, 4, and 5 shows the scene from the same starting position

but while rendering 20 000, 350 000, and 1 000 000 grass straws respectively.

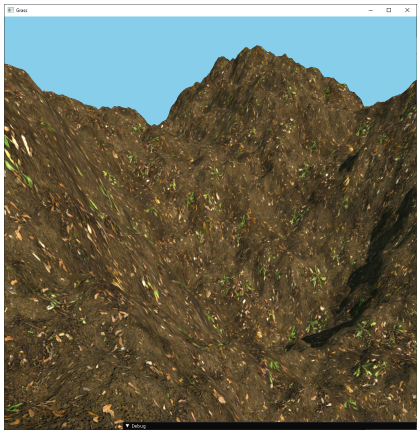

Figure 2: The scene with no grass

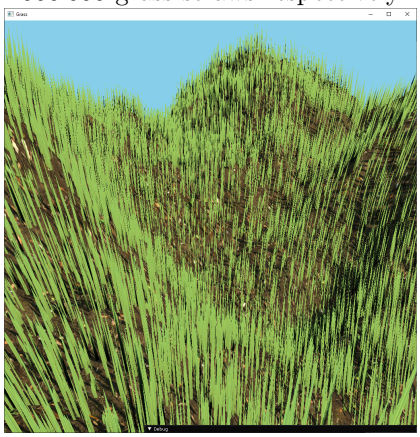

Figure 3: The scene with 20 000 grass straws

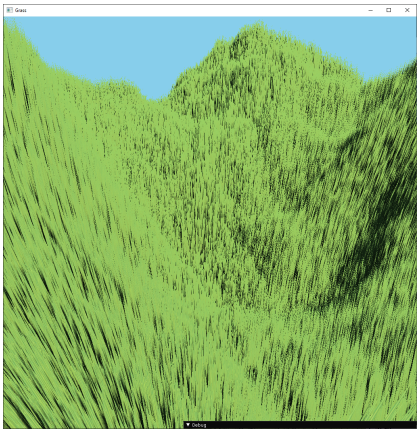

Figure 4: The scene with 350 000 grass straws

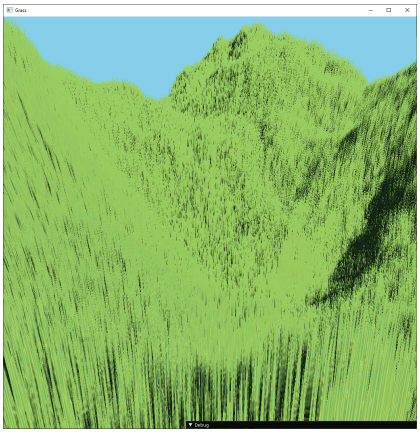

Figure 5: The scene with 1 000 000 grass straws

The direction of the directional light can also be changed. Figure 6 shows the scene from a different position where the light is positioned such that a hill casts a shadow. In figure 7 the light has the same orientation but the shadow map has been disabled. In figure 8 the shadow map is enabled but the orientation of the light is different. Comparing figure 6, 7, and 8 shows the difference in both quality and realism with and without shadow mapping. Using shadow mapping also makes it much easier to perceive depth. Figure 9 shows the scene with the same settings as in figure 6 but with no ambient occlusion. In this figure everything is much darker than when ambient occlusion has a higher intensity as in the other figures. The areas where shadows are cast are also completely dark in this image.

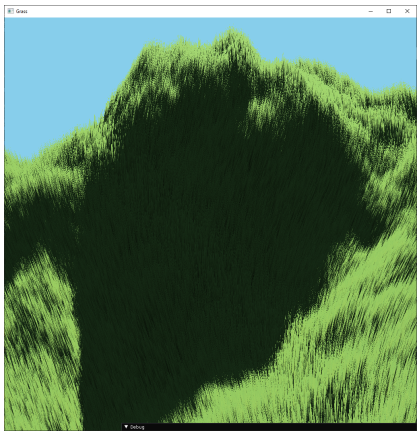

Figure 6: Hill casting shadows

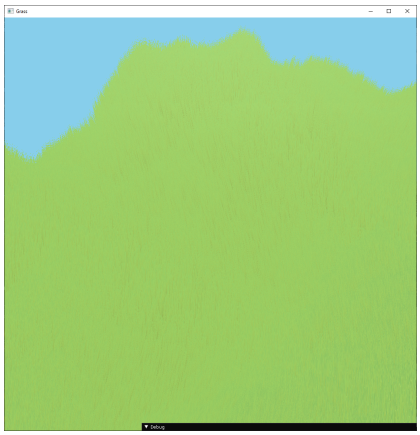

Figure 7: Shadow map disabled

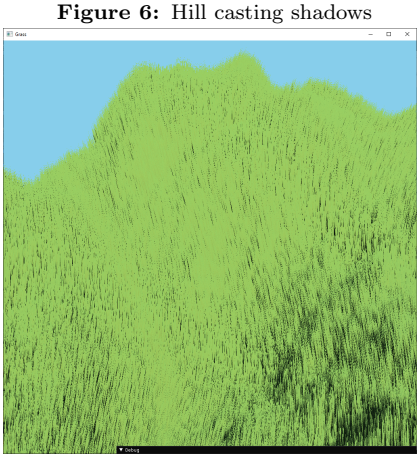

Figure 8: Shadow map with different orientation

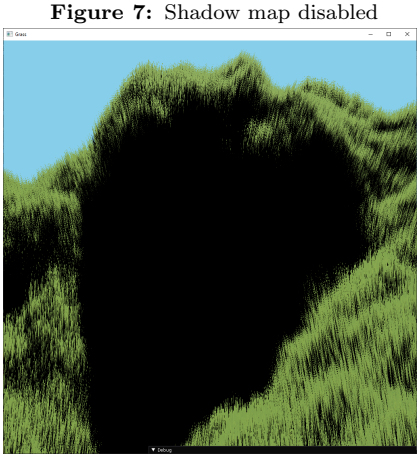

Figure 9: Shadow map without ambient occlusion

5 Performance

The program can be more or less demanding depending on the number of grass straws and whether the shadow map is enabled. When the shadow map is disabled even a laptop, without a dedicated graphics, card can run the scene at a playable framerate with several hundred thousands grass straws. However, when the shadow map is enabled, the scene becomes a lot more demanding. When shadow mapping is enabled a GTX 1070 GPU and a i5-6600k CPU can run the scene at 72 FPS with 350 000 grass straws as in figure 4. With 1 000 000 grass straws as in figure 5 (the largest possible number of grass straws) the scene runs at 37 FPS. Comparing figure 4 and 5 The visual difference between

rendering 350 000 and 1 000 000 grass straws is quite small. If the program was a real game it might therefore be preferable for the player to use a lower number of grass straws in order to achieve better FPS.

6 Additional Features

One feature implemented which was not part of the requirements is wind. Movement makes the scene look more interesting and wind is an easy way to add movement.
Implementing wind is be done by rotating the vertices of the grass straws around the point they intersect the ground (the position of the bottom two vertices). This was done in the vertex shader by rotating around the x and z-axis. How much is rotated around each axis is determined by the direction and speed of the wind.

7 Future Improvements

All the requirements of the project have been met. However, there are still a few features which could improve the final product.
It is clear from looking at the final project that the grass causes a lot of aliasing. This is because there are a lot of grass straws and they very thin. This worsens visual quality as aliasing is not nice to look at. The obvious solution to this problem is implementing anti-aliasing. This would help smooth out jagged edges.
Another improvement is transparency. A lot of what causes grass to look the way it does is transparency. Transparency could be implemented simply by enabling blending. However, properly rendering transparent objects requires sorting them from back to front. This is not practical as the program can render a large number of grass straws which would all have to be sorted. This would most likely be too hardware intensive to do every time the camera moves and has therefore not been implemented. Blending is also not possible to use because of deferred rendering. Transparency could be faked by using a technique known as screen-door transparency. This technique works by discarding every other fragment in a checkerboard pattern.
Another future improvement could be cascaded shadow maps. It is clear from looking at the final result that there is some aliasing even with PCF. The solution to this is the use of cascaded shadow maps where the view frustum is divided into multiple subfrustums. A shadow map is then rendered for each subfrustum. The closer subfrustum get a higher resolution.

8 Conclusion

The goal of this project was to simulate grass using OpenGL and C++. This was done by adding instanced rendering to ItuGL and using it to render a large number of grass straws. Lighting and shadow mapping were also added to increase the quality of the scene and help the player perceive depth.
The player can change several settings which affect how the scene looks. This includes changing the number of grass straws using a slider. This enables the player to render anywhere from 0 to 1 000 000 grass straws depending on their hardware. The shadow map can also be disabled which increases performance but makes it harder to perceive depth.
Wind was also added as an additional feature and makes the scene look a lot more interesting.
Several improvements can be made to the scene to improve the look. This is for example anti-aliasing, transparency, and cascaded shadow maps.

References

[1] Joey de Vries. *Instancing*. URL: https://learnopengl.com/Advanced-OpenGL/Instancing. (accessed: 16.05.2023).

[2] eye-candy.xyz. *Mud Forest*. URL: https://polyhaven.com/a/mud_forest. (accessed: 13.05.2023).

[3] Joey de Vries. *Normal Mapping*. URL: https://learnopengl.com/Advanced-Lighting/Normal-Mapping. (accessed: 16.05.2023).

[4] Joey de Vries. *Shadow Mapping*. URL: https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping. (accessed: 12.05.2023).