DH2323 VT24 Computer Graphics and Interaction: Shell Texturing

Byushra Ashak
byushra@kth.se
https://github.com/busra-ashak/Shell-Texturing
KTH Royal Institute of Technology

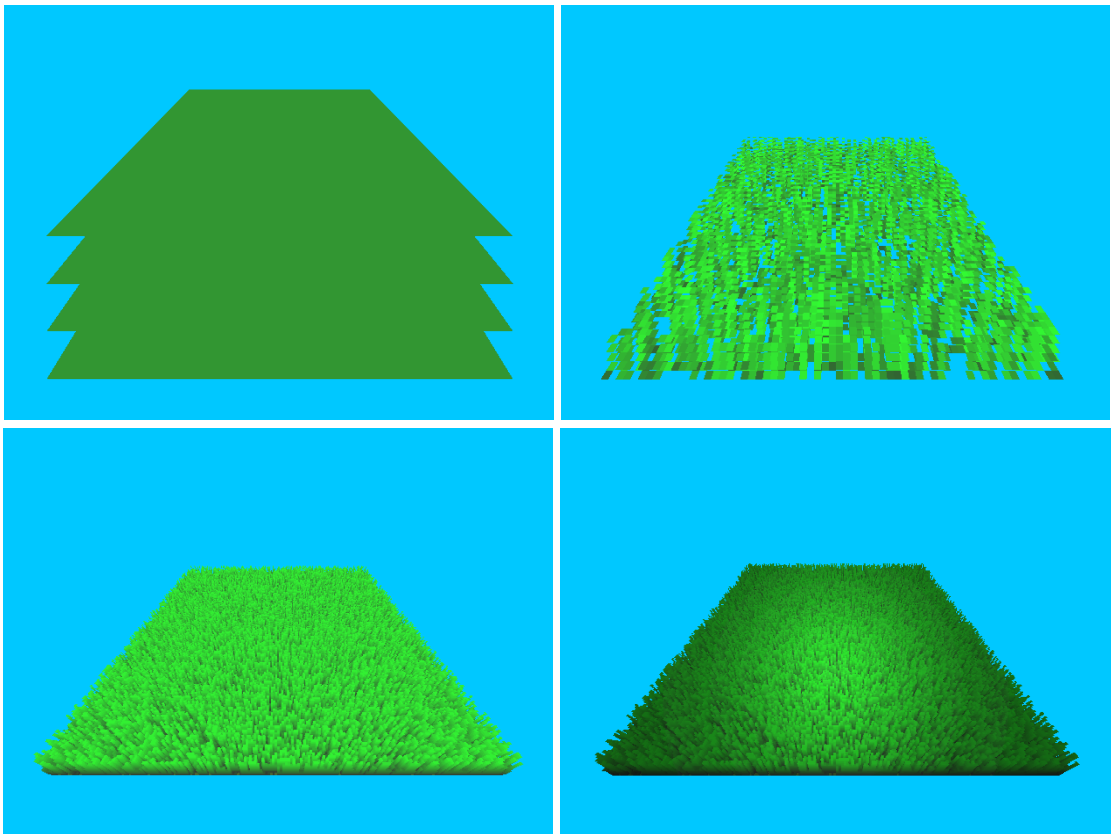

Fig. 1. *The progression of the shell texturing implementation on a plane of grass*

***Abstract*—We have created a minimal implementation of the shell texturing method in C++ via OpenGL 4.5 where we rendered a plane of grass. The application includes a moveable 3D camera allowing interactive navigation through the scene and allows users to increase or decrease both the spacing between the shells and the number of the shells at runtime.**

I. Introduction

One of the bigger challenges in computer graphics is creating realistic natural environments, particularly when it comes to rendering complex surfaces like grass or animal fur. Creating realistic foliage can be difficult due to several factors such as the vast amount of geometry involved in rendering hundreds of thousands of grass blades, high occlusion that drives up the cost of lighting effects, and the expensive nature of physics simulations for individual hairs or grass blades. To simulate them, a number of methods have been developed, all trying to achieve a compromise between computational efficiency and visual fidelity. In this project, we render a realistic grass plane using C++ and OpenGL in an interactive application using the shell texturing technique.

Shell texturing, proposed by Lengyel et al. [1], is particularly effective for rendering complex surfaces that require a high level of detail. Shell texturing is an alternative to traditional texturing techniques, which apply a single texture map to a surface. Instead, it layers several semi-transparent textures, or "shells," to give the impression of depth and volume. Although it has been commonly used for rendering fur and hair, because of grass's similar complex and dense structure to hair, shell texturing can be used to render grass as well. This method works great for grass because it can

mimic the complex overlapping structure of individual blades without requiring an unreasonably large number in polygons.

This paper details the design and implementation of our grass rendering technique. We begin by reviewing related work in the field of grass rendering and shell texturing method implementations. The subsequent sections cover the technical details of our implementation and visual results. Finally, we discuss the potential areas for future research.

II. Related Work

The realistic appearance of grass, or any other type of natural vegetation, relies on two factors: detailed modelling of the form and realistic lighting of the model. A 3D artist can model grass blades with a modelling program like Maya or Blender. Though it takes a great deal of physical labour and attention to detail, the outcome is the most accurate approximation of grass geometry. A fair trade-off between visual accuracy and rendering expense must be made, though, as precisely lighting and rendering this massive collection of polygons is not feasible for real-time applications. Advances in technology have led to the development of increasingly effective techniques, many of which are suitable for real-time application. This section discusses the most practical ones.

In computer graphics, billboarding is a commonly used method for rendering complex objects, like grass, at a relatively low computational cost. A billboard is a semitransparent polygon with a vertical orientation that is orthogonal to the direction of vision. The texture's transparency allows it to depict items with intricate shapes, including trees and plants. Jakulin [2] precomputes an alternative billboard from more than two perspectives, extending this technique for rendering trees. Using linear interpolation, the nearest pair of billboards is chosen at runtime and combined. Parallax is maintained because more than one billboard is utilised along the viewing beam. Unfortunately, this method does not take into consideration changes in illumination.

As another approach, Lengyel et al. [1] presented a real time method employing texture layers based on the work by Neyret [3], establishing the shells approach for fur rendering. In the implementation of shell texturing for grass rendering, each fragment undergoes a process to determine whether it is part of a grass blade or empty space. This is achieved by sampling a noise function using the current vertex's texture coordinate. The noise function generates a value that is then compared against a threshold. If the sampled value exceeds the threshold, the fragment is deemed to be part of a grass blade; otherwise, it is discarded. This method ensures that only the necessary fragments are rendered, optimising performance by reducing the number of fragments that need to be processed.

The threshold value is dynamically adjusted based on the shell's ID, creating a gradient of grass density. At the lower shells, which are closer to the ground, the threshold is lower, resulting in denser grass. As the shell ID increases, moving towards the top, the threshold value also increases, making the grass sparser. This gradient effect mimics the natural appearance of grass, which is typically denser at the base and more sparse at the tips. This approach not only enhances the visual realism of the grass but also improves the efficiency of the rendering process by minimising unnecessary computations for fragments that do not contribute to the final image. Lengyel's approach is one of the most intuitive and easy ways to render grass and fur. Thus, we were inspired to implement shell texturing.

III. Implementation and Results

We present the steps taken in the implementation of the shell texturing method to render a plane of grass. We begin by discussing the libraries and technologies used to create the application. Then we explain the implementation of persistent mapped buffers, the algorithm and the logic of shell texturing, and the Phong illumination. Finally, we present the user interaction aspect of our application and the controllable 3D camera.

A. Libraries Used

In order to construct a realistic grass rendering system, we made use of a number of important libraries and technologies that are supported on Windows and Linux operating systems. The project's foundation was the OpenGL 4.5 API, which offered the features required for rendering graphics. we utilised GLFW to control the window's construction, the OpenGL context, and keyboard and mouse input. This library made it easier to build up the graphical user interface and manage user interactions. As the OpenGL loader, GLAD was used to access contemporary OpenGL features and extensions, guaranteeing compatibility with the most recent releases. Furthermore, vector and matrix operations—which are essential for transformations and calculations within the graphics pipeline—could only be completed with the aid of GLM (OpenGL Mathematics). When combined, these libraries made it easier to apply the shell texturing method, which enables the layering of textures to replicate the volumetric appearance of grass in real time.

B. Persistent Mapped Buffers

In this project, shader storage buffer objects (SSBOs) are employed to manage vertex data, diverging from the traditional use of vertex buffer objects (VBOs). Storage buffers remove the requirement to inform OpenGL about the vertex data format, enabling a fully shader-driven approach. This method greatly reduces boilerplate code and simplifies setup by doing away with the need to define vertex characteristics within the programme.

For the purpose of managing "static" and "dynamic" shader storage buffers, a custom buffer structure was created. For vertex data that is written only once during initialization and stays the same throughout the program's execution, static buffers are utilised. For the vertex data of the grass blades, which don't need to be updated often, this is perfect. Conversely, dynamic buffers make use of a contemporary OpenGL feature called persistent mapped buffers [4]. Because of this, writing data to the buffer is possible whenever needed, without resulting in the overhead that comes with regular data updates. Dynamic buffers are very helpful in this project for managing camera data, which is constantly changing as the viewpoint changes. The camera data may be updated

effectively and smoothly, resulting in responsive and smooth interactions in the projected scene, by mapping the buffer persistently.

By combining SSBOs with persistent mapped buffers, the grass rendering system's efficiency and adaptability are optimised. Real-time updates are made possible, and the overhead usually related to dynamic data manipulation is reduced. This method preserves high efficiency, which is essential for real-time applications, while also improving the rendered grass's visual accuracy.

C. Shell Texturing

After completing the boilerplate code for the set up, the next task was to draw multiple planes using the glDrawArrays function. Later, these planes would represent the shells of our implementation with the help of a noise function. We created a vector consisting of six vec4's that represent the vertices of two triangles and wrote this into our vertex buffer. The vertices of these triangles would form a plane parallel to the xy plane. Since glDrawArrays takes the number of vertices to be drawn as an argument, we first gave it 24 to draw four planes with the vertices in the vertex buffer. In the vertex shader we use a variable called plane ID to differentiate between different planes and it is calculated by dividing the ID representing the current vertex within the shader, gl_VertexID, by six. Moreover, while retrieving the vertices from the vertex buffer to set the value of gl_Position, we get the vertex at the index of gl_VertexID(mod 6). With these combined we are able to draw the same plane four times with different plane IDs.

We wanted the planes to have spacing in between them and lie in an angle so that the user could see the spacing. So, we created a new struct to store the information for spacing between the shells and a transform matrix to transform these planes into any orientation or scaling that could be needed. We have created a buffer for storing the transform matrix and the spacing which will be used for each vertex in the vertex shader. We set the value for the transform matrix to a rotation matrix that will rotate vertices to be on the xz plane. Then, in the vertex shader the position of the vertices are first multiplied by the transform matrix. Next, we multiply the spacing value and the plane ID to create equal gaps in between the planes and add this value to the result. This procedure is summarised in pseudocode in algorithm (1). Finally, as we set this value to gl_Position, we got the result shown in Figure 2 after setting the colours of the planes in the fragment shader.

Algorithm 1 Drawing Planes

Input: Current vertex ID I, array of vertices V, transform matrix T, spacing between planes S
Output: gl_Position P

1. plane_id = I/ 6
2. pos = V[I%6] * T
3. P = pos + S* plane_id

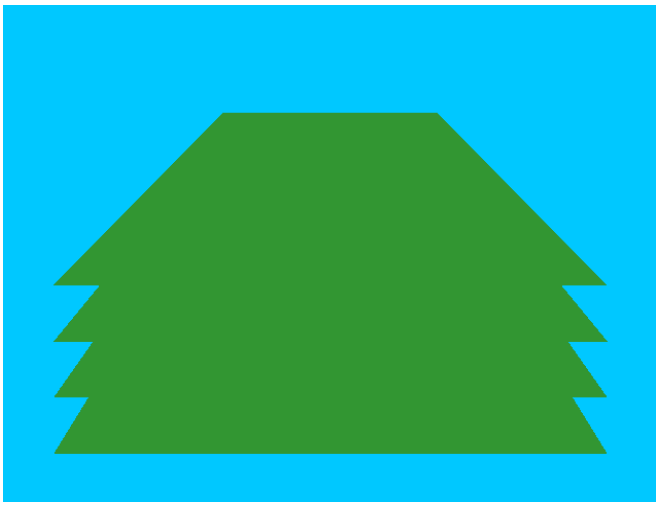

Fig. 2. *Four planes on top of each other with equal spacing in between them*

In the fragment shader we gave access to the plane ID variable and the position of the pixel we are currently drawing. We do this in order to be able to differentiate between the planes and the pixels on them for colouring or lighting purposes. Next, we created a rand function in the fragment shader to retrieve pseudorandom values. The input x will be the position of a pixel on a plane and the output value is the fractional part of the result in expression (1). The function gives seemingly random values based on the input because of the combination of multiplication of large numbers. This way we have a float noise value for each pixel position. We need this value so we can have a non-regular appearance of grass blades. If it was uniformly spread, for example, it would not look like realistic grass.

$$sin(x * [12.9898, 4.1414] * 43758.5453) \quad (1)$$

With this function we can determine whether grass should be drawn in that position or not. This is done by comparing the noise value to the predetermined threshold and if it is above the threshold value, the patch gets drawn. In order to test if the random function we implemented was working correctly, we drew the planes by setting the colour of every pixel to the multiplication of a shade of green and the output of our function with the input being the pixel position. The result of this is shown in Figure 3. Then we discarded the pixels if the noise value was lower than the threshold and got the result shown in Figure 4 and Figure 5 with different threshold values.

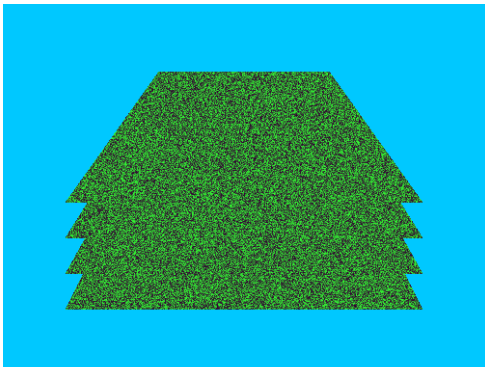

Fig. 3. *Four planes of grass - pixels coloured with random shades of green*

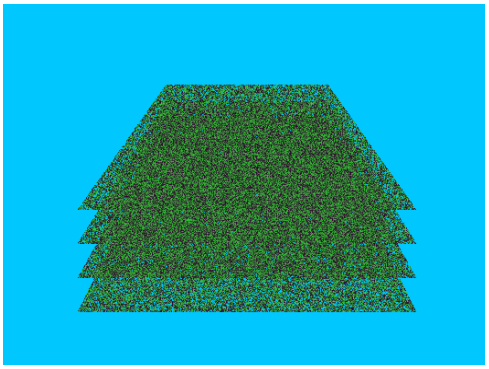

Fig. 4. *Four planes of grass - pixels coloured with random shades of green if the random value is larger than threshold (small threshold value)*

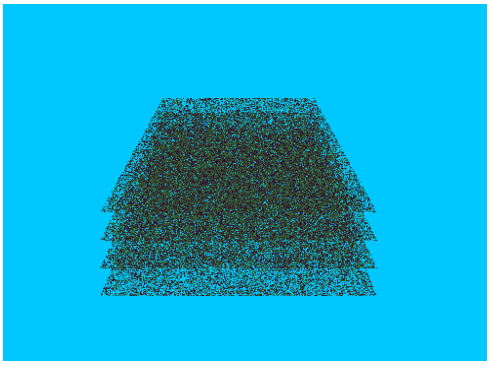

Fig. 5. *Four planes of grass - pixels coloured with random shades of green if the random value is larger than threshold (large threshold value)*

After confirming that the random function works correctly, we decided to create a variable for the number of grass blade patches on one edge of the planes. So, we would create a random value for each patch rather than every pixel position. To do this, we firstly multiply the position value of the pixel with the grass blade count. We then round the result to the nearest integer, and finally divide the result by the grass blade count. This yields a UV that is the same for every single pixel within the patch. With this change, all the pixels in that patch would be given the same colour since they would output the same random value. Now, we were able to set the size of the grass blades as we liked and got the result shown in Figure 6 with the grass blade size variable set to 64.

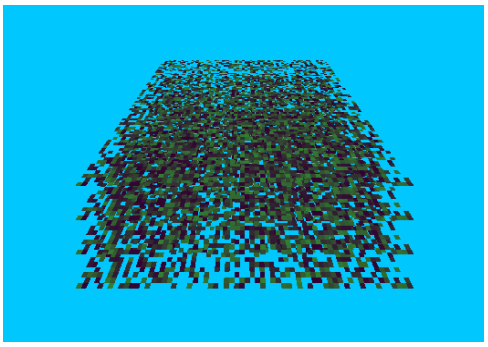

Fig. 6. *Resizable grass blade patches*

Then we reduced the spacing between the planes and increased the threshold value as the plane ID increased in order to give the look of every grass blade having random heights. With relatively large patches and only four planes, the result is shown in Figure 7. The final algorithm for shell texturing in the fragment shader is shown in algorithm (2). It is important to note that we assume that P is both the position in model-space, and also normalised. This is not the case for most meshes, but is in the case of our hard-coded plane. We are essentially treating the local-space position as a texture coordinate.

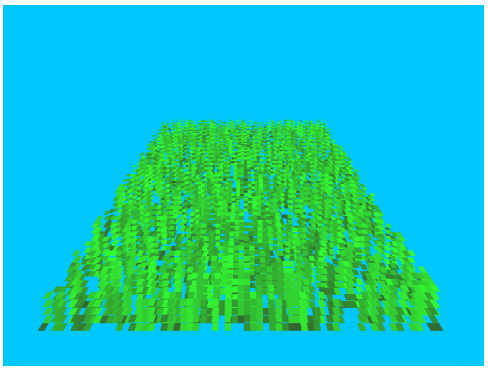

Fig. 7. *Resizable grass blade patches*

Algorithm 2 Shell Texturing

Input: Position P, plane ID I
Output: gl_FragColor F

```
1. GRASS_BLADE_COUNT = 64
2. QUANTIZED_UV = FLOOR(P * GRASS_BLADE_COUNT) /
   GRASS_BLADE_COUNT
3. NOISE_PER_FRAGMENT = RAND(QUANTIZED_UV)
4. IF NOISE_PER_FRAGMENT < 0.2 + I*0.1 THEN
5.   DISCARD
6. ELSE
7.   GL_FRAGCOLOR = (0.2, 0.5*NOISE_PER_FRAGMENT, 0.2)
8. END IF
```

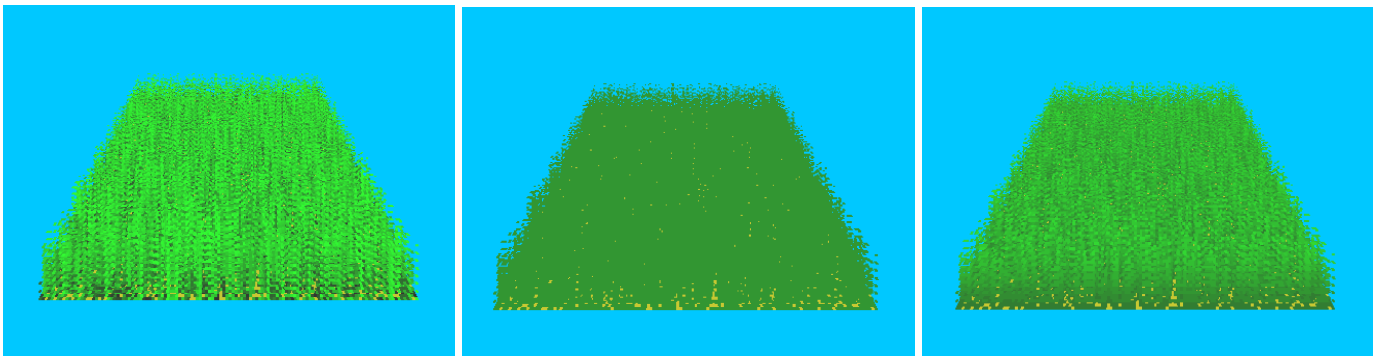

Fig. *8. Number of shells increased to 10 and different colouring techniques. Left: colour of blades is based on the noise value. Middle: colour of blades is uniform. Right: colour of blades brightens as plane ID increases*

In order to get realistic looking grass results we set pixels on the first plane to be a uniform colour and tried different colouring ways for the grass blades as shown in Figure 8. We also increased the number of shells and decreased the spacing in order to make the grass blades look complete. We first had the random shades of green colouring. Then we tried setting all the blades to the same shade of green. Although the grass blades became indistinguishable, this colouring would be needed to test the illumination method later on. Lastly, we tried brightening the shade of green as the plane ID increased. This implementation appeared like the illumination was implemented since the bottom parts of the grass blades looked darker as it would be in real life because of the lack of light reaching there. We decided to keep this implementation until we added the illumination model since it looked more realistic. Then we set the colour of the most bottom plane to a brown shade, increased the number of shells to 20, and decreased the spacing again. The result is shown in Figure 9.

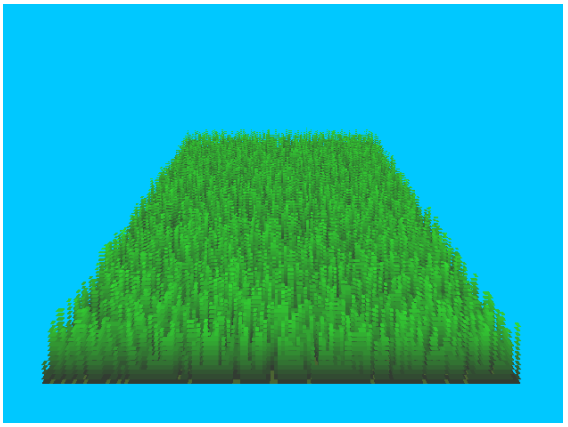

Fig. 9. *Realistic rendering of grass with shell size 20 and spacing 0.01*

It was at this point we noticed a mistake in the implementation. The spacing we have been adding between the planes has been along the y-axis no matter the orientation of the planes. This would have been fine if the planes were hard-coded to be parallel to the xz plane. However, we had transformed these vertices with our transform matrix earlier. We want the grass blades to be growing in the same direction as the bottom plane's normal. So, we updated the spacing to take the transform matrix into account and got the result shown in Figure 10.

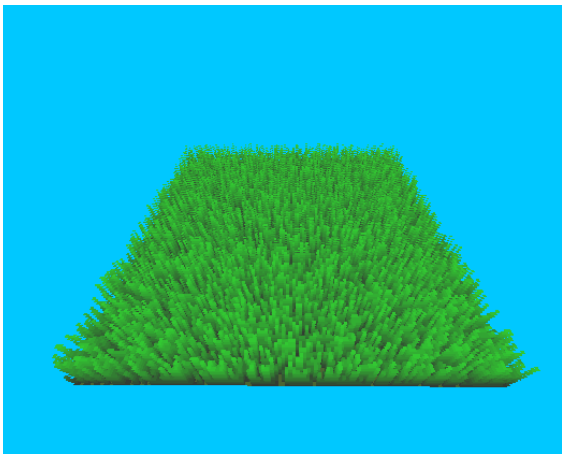

Fig. 10. *The end result when orientation of the grass blades is fixed*

D. Phong Illumination

As we chose to give the grass blades a lighter colour depending on how high up the pixel is, we have virtually lit pixels more depending on how exposed they are to environmental light that would be directed towards the planes. This is a form of ambient occlusion. However, the grass could look more realistic if it had a specular highlight from a directional light and we didn't need the shade of green to get lighter. We believed we could naively implement the Phong illumination model without modifications, however we got the result shown in Figure 11. It was not clear to us whether the shell planes would trivially exhibit the expected results when implementing the illumination model. After seeing the result we realised that all of the normals of the grass blades point perpendicular to the plane, making it appear perfectly smooth. All of the grass blade "geometry" was no longer visible, apart from the edge of the plane. In order to properly fix this, we would need to somehow adjust the current normal of each sampled fragment to be perpendicular to the non-existent grass blade geometry.

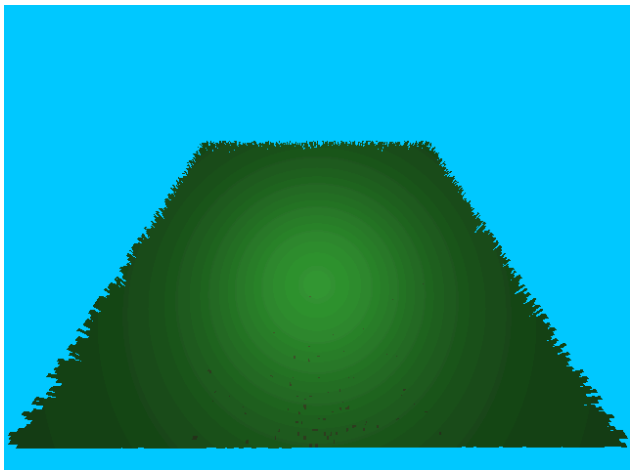

Fig. 11. *Uniform colour grass blades with Phong illumination model*

Then, we tried brightening the shade of the green on the grass blades as the plane ID increases again. Combining this with this specular highlight, the result becomes much more satisfactory although not a complete implementation of the Phong illumination model, shown in Figure 12. This is our final result. Here, we have achieved a shell-textured set of planes, along with adding a Phong specular highlight to give the grass some shine. A real-world use of this technique on grass could be to make the grass look more glossy - as if it has recently rained.

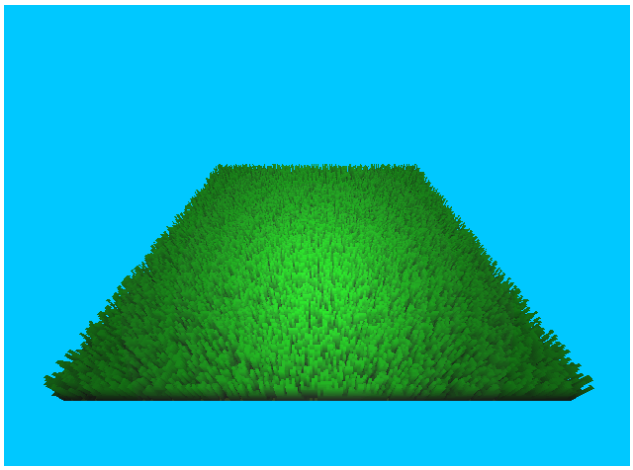

Fig. 12. *Colour of blades brightens as plane ID increases - creating a primitive ambient occlusion effect*

E. 3D Camera & User Interaction

We wanted user controls to be focused on experimenting with and exploring the shell texturing method. So, we wanted users to be able to change the number of shells and the spacing in between shells at runtime. Furthermore, we wanted users to be able to navigate the 3D scene, allowing them to move around and be able to rotate the camera freely around the y-axis and the x-axis.

The camera uses a perspective projection, with some hard-coded field-of-view and aspect-ratio. In order to be able to move the camera depending on keyboard input, we chose to create another storage buffer to store both the view and projection matrices. This exposes the matrices to vertex shader to use in its calculations. In addition, the buffer is a dynamic buffer - allowing the input handler to make edits to the matrices on-the-fly without having to halt the GPU pipeline. Lastly, in order to take the camera position and view into account, in the vertex shader we set the value of gl_Position variable to be equal to the multiplication of view, projection, and the previously transformed position of the vertex. Furthermore, we created callback functions for mouse and keyboard and implemented cases for moving along all three axes and rotating around the y and the x axis. With these implemented the user could navigate in the space and an example is shown in Figure 13. Now, the user could also see where the shell texturing method would break down in specific angles like shown in Figure 14.

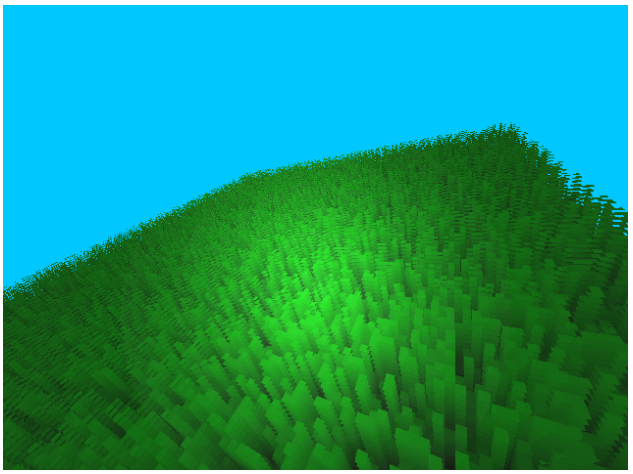

Fig. 13. *Demonstration of user navigation with moving the camera and rotating the planes*

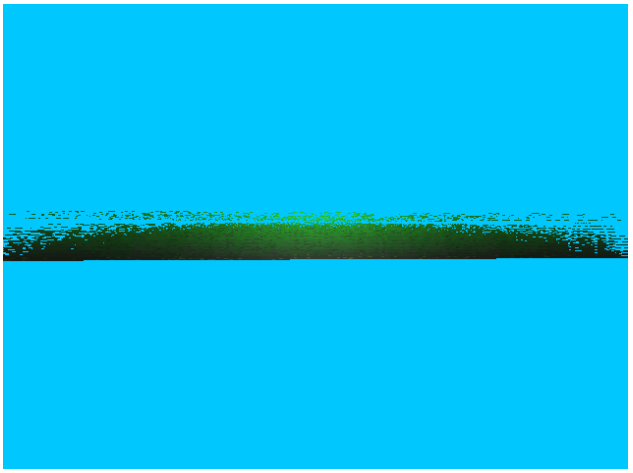

Fig. 14. *Demonstration of shell texturing breaking down when shells are parallel to the forward vector of the camera*

We wanted the user to be able to control the number of shells and the spacing between the shells. In order to achieve this, we added new cases to the keyboard input callback functions and an example of the result is shown in Figure 15.

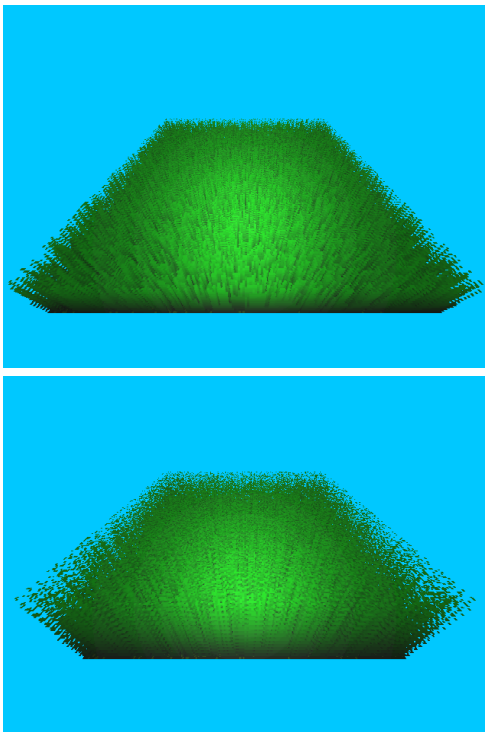

Fig. 15. *Demonstration of different spacing values with the same number of shells.*

IV. Future Work

This project creates a number of opportunities for further development and research. The shell texturing method's current implementation offers a base that can be interestingly expanded upon to increase usefulness and realism.

One potential extension is to apply the shell texturing method to render fur on more complex models, such as the Stanford Bunny. This would need modifying the existing method to accommodate the curved surfaces and more intricate geometry of the bunny. We can show the adaptability and scalability of the shell texturing technique for many object kinds, increasing its applicability in computer graphics, by effectively rendering fur on the Stanford Bunny.

Achieving more realism requires improving the lighting effects and scene's overall illumination. We had trouble getting the right lighting effects in the current implementation. At first, we tried to use the Phong lighting model directly, but we ran into problems because the grass blade normals were perpendicular to the shell planes, which gave the image of being smooth and unnatural. To address this, we realised the need to adjust the normals of each sampled fragment to align with the imaginary geometry of the grass blades. The implementation of fins to fill in the gaps that was proposed by Lengyel [1] could also help with the calculation of the normal vectors. We believe we can accurately model the light-blade interaction by doing this. Furthermore, decreasing the glass blade size as the plane ID increases could also help with the lighting and make the grass more realistic. Further research could focus on improving this strategy or investigating different lighting models in order to more accurately depict the subtleties of light's interaction with shell-textured surfaces.

Other potential extensions include improving the realism of grass with different behaviour and physics implementations. An example for simulation of specific conditions would be rendering wet fur or dew-covered grass. This would require modifications to the texture and lighting calculations to reflect the increased specularity and altered colour properties of wet surfaces. Adding effects like water droplets and modifying the reflection and refraction properties would make the fur and grass appear more lifelike under different environmental conditions. A study on rendering wet fur with shell texturing has been conducted by Bando and Chen [5] where they realistically render clumping of wet hair. Realistic representation of the grass could be also greatly improved by adding physics. By using a physics engine to mimic wind and other environmental elements, the grass blades may move organically and react to different forces. In order to make the scene more dynamic and immersive, this would include incorporating physics-based algorithms to determine the movement and bending of each blade of grass.

V. Potential Perceptual Study

There are multiple aspects of the shell texturing and our implementation of that technique that could be used in perceptual studies. For example, studies on authenticity, accuracy in environmental conditions, visual quality and usability studies in interactive applications could be conducted in this field.

One possible perceptual study could evaluate the realism and visual quality of the shell texturing method. The perceived realism and aesthetic quality of the images or interactive scenes created using shell texturing and alternative techniques (such as classic texturing or geometry-based grass rendering) would be evaluated by the participants. This would make it easier to assess if shell texturing provides a better visual experience and point out its advantages and disadvantages. The study might also look into how well shell texturing represents various environmental conditions, including fur and grass that is damp or drenched with dew. In order to evaluate the technique's effectiveness and identify areas for improvement, participants would rate the effects' convincingness and accuracy.

An investigation on the usability and performance of shell texturing could be carried out for interactive applications, like virtual reality or gaming. In surroundings with fur or grass with a shell texture, participants may complete activities, and metrics like task completion time, user happiness, and perceived visual quality could be recorded. This research will shed light on the effects of shell texturing on user engagement and the practical ramifications for real-time applications.

References

[1] J. Lengyel, E. Praun, A. Finkelstein, and H. Hoppe, “Real-time fur over arbitrary surfaces,” in Proceedings of the 2001 Simposium of Interactive 3D Graphics, pp. 227–232, 2001.

[2] A. Jakulin, "Interactive vegetation rendering with slicing and blending", In Eurographics 2000 (Short Presentations), 2000.

[3] F. Neyret, “Modeling and animating, and rendering complex scenes using volumetric textures,” IEEE Transactions on Visualization and Computer Graphics, vol. 4, no. 1, pp. 55–70, 1998.

[4] C. Everitt, T. Foley, J. McDonald, and G. Sellers (2014). Game Developers Conference: Approaching Zero Driver Overhead in. [Online]. Available: https://www.gdcvault.com/play/1020791/Approaching-Zero-Driver-Overhead-in

[5] Y. Bando and B. Chen, (2010). “Real-time Rendering of Wet Fur”, 2010.