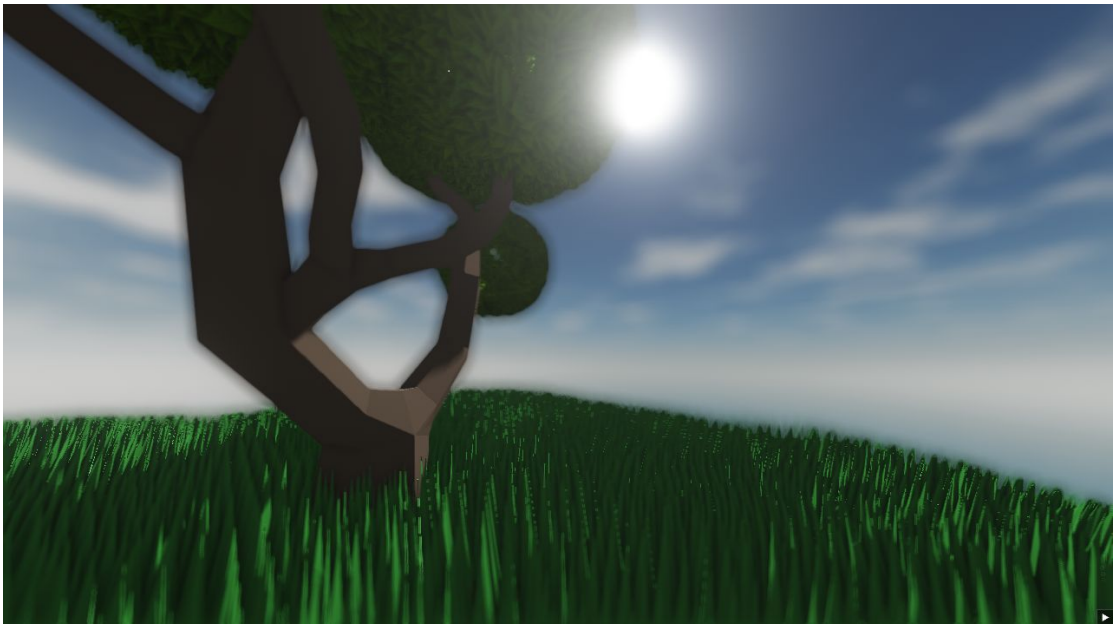
3D Animation and Visualisation 2020

Required:
Project Code + Report
(Online submission)

Peaceful Breeze

A small scene containing a tree surrounded by procedurally generated grass which is enhanced by post-processing features and shadows to create beautiful scenes that can be captured



Jorge Nunes	**João Patrício**	**Marcos Pêgo**	**Daniel Rodrigues**
87677	97046	86472	65881
MEIC	MEIC	MEIC	MEIC
jorge.costa.nunes@tecnico.ulisboa.pt	joaopat98@gmail.com	marcosppego@gmail.com	daniel.rodrigues@tecnico.ulisboa.pt

Abstract

The scene is a mix between blender models and procedurally generated grass and leaves using tessellation techniques. To further develop the scene, it's also possible to add wind that shifts the grass and leaves along it. The scene is then enhanced with shadow mapping to add realism between the sun and the scene and post processing effects such as ambient occlusion, bloom and hdr that highlights the natural beauty of the scene. To allow for different lighting angles, alongside the skybox was added a movable sun that shifts the light around the scene to see different angles of shadow and light. There is a lot of flexibility with the scene using the UI available that allows the user to change the color of the sun, the hdr and bloom settings, the quality of shadows, how much tesselation the leaves or grass have and other minor features. The end goal is to capture good angles of the scene so all the aspects are highlighted so a screenshot feature was added to save different looks of the scene.

1. Concept (max 2 pages)

In this project the goal was to create a high quality scene using post-processing techniques and lighting techniques and minimal graphical sources. So we decided to create a simple scene inspired in a gif we saw when searching for ideas:

Fig. 1 Inspiration for the project

The gif itself was a scene with a rotating sun that rotated around the scene showing different perspectives of light and shadow that achieved stunning results with a simple set of grass and one tree. We decided to try to recreate it but in our own style and using some fancier effects to achieve different better visual results. Thus the main features that we knew had to be included were shadows, toon like shading and a sun that allowed us to see the scene with different lighting directions.

Then to add more complexity to the scene and make it better looking we also decided to add post processing effects such as hdr, bloom and depth-of-field and add ambient occlusion for a more realistic feel to the scene. To conclude we also found it would be interesting to add procedurally generated grass and possibly even the leaves of the tree using tessellation techniques.

2. Technical Challenges (max 1 page / student)

2.1. Cel-based Toon Shading (Marcos Pêgo)

The cel-based toon shading is a two toned shader with a linear specular light to give a toon look to the scene.

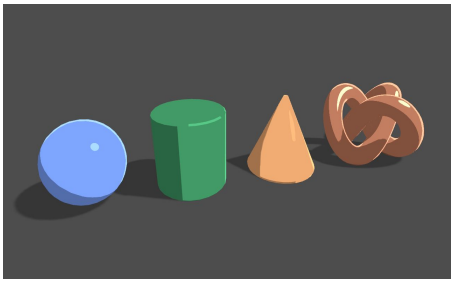


Fig. 2 Toon Shading

2.2. Shadow-Mapping (Marcos Pêgo)

Shadow-mapping is a technique used to create shadows from one object to another.

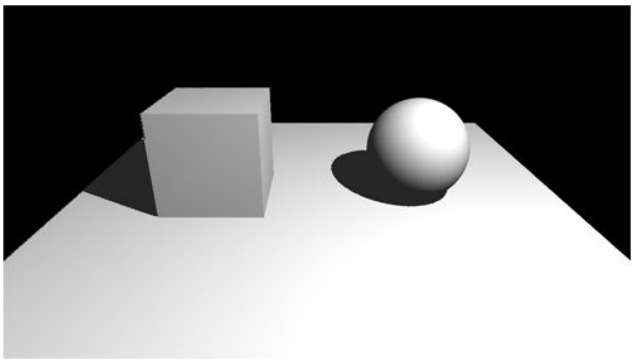


Fig. 3 Shadow Mapping

2.3. Screen Space Ambient Occlusion (João Patrício)

SSAO is a screen space effect which simulates areas in the scene which are more occluded from light, improving depth perception in scenes with lots of objects. Given this, it would certainly improve the quality of our scene.

Fig. 4 - Comparison showing the effect of SSAO

2.4. Post-Processing Stack (João Patrício)

Post-Processing can substantially improve the look of a game. Effects like bloom, tonemapping and depth of field can give more realism to the scene and give life to the objects within it. For this, a post-processing stack which allows the consecutive application of effects will easily allow us to improve the look of our scene.

Fig. 5 - Comparison showing the effect of Bloom

2.5. Skybox + Sun (Jorge Nunes)

Since we are trying to create an aesthetically pleasing scene, it's a must we have a skybox in which we could put our scene in. Besides the clear sky (with a few clouds), we will also need a sun, which should be separate from the skybox itself, hence a second skybox, that should be transparent with only a sun.

Fig. 6, 7 - Left: Skybox, Right: Sun

2.6. Sun Movement (Jorge Nunes)

One of the main features we want to be available in our scene is to move the sun. This movement should change the way the shadows appear. It should be done using a user's input.

2.7. Screenshots (Jorge Nunes)

Aiming for a high-quality scene demands for a way to save what we are seeing in the scene. We should be able to take as many screenshots of the scene as we desire without any delay and affecting the scene itself. This should also be done using a user's input.

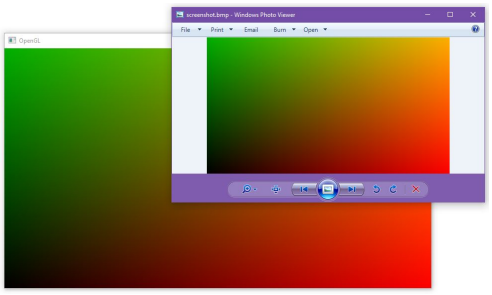


Fig. 8 - Example of a screenshot being taken

2.8. Terrain composed of grass (Daniel Rodrigues)

One of the technical challenges consisted in creating a terrain composed of grass. The main requirement was the use of other shader techniques than vertex and fragment shaders. It was our intent that the desired result should look like this:

Fig. 9, 10 - Left: Grass terrain, Right: Grass blades

What we proposed was a combination of Tessellation and Geometry Shader Techniques. We decided to use the Tessellation technique because that way we could use low polygon models for the terrain instead of a complex model, offloading the vertex calculations to the GPU.

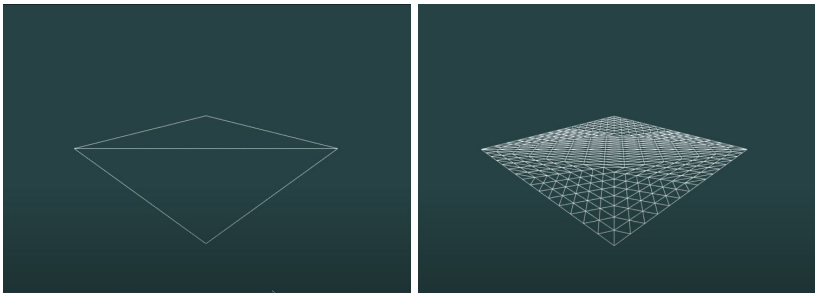


Fig. 11, 12 - Left: No Tessellation, Right: Tessellated square

In addition we wanted to create a flowing grass animation (as is displayed here) that resembled the presence of the wind in the scene. It was defined that applying this effect to the grass terrain was the priority and optional for the tree leaves.

3. Proposed Solutions (max 5 pages / student)

3.1. Cel-based Toon Shading

To implement a Cel-Based Toon Shading we followed an article by roystan.net for unity shader and adapted it to opengl. The effect is created using a blinn-phong shader composed of ambient light (natural light of objects), diffuse reflection and specular reflection.

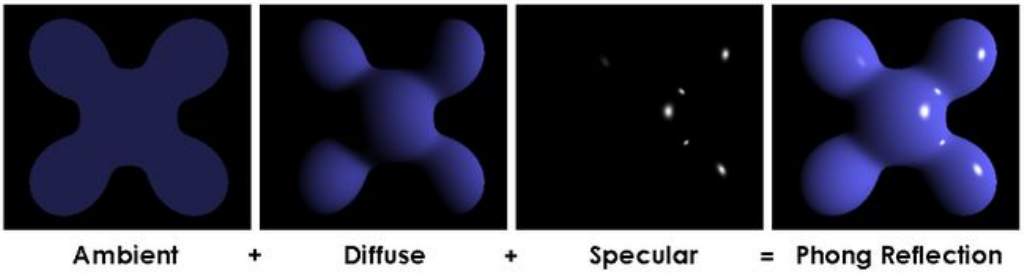


Fig. 13 - Phong shading

Rather than the continuous gradient of intensity (more bright to dark shade of the same color) the cel shading method makes flat changes between colors depending on how many levels are defined.
For example a cel shading method with 4 levels will change its color 4 times from its brightest color to its darkest depending on its angle to the light source.

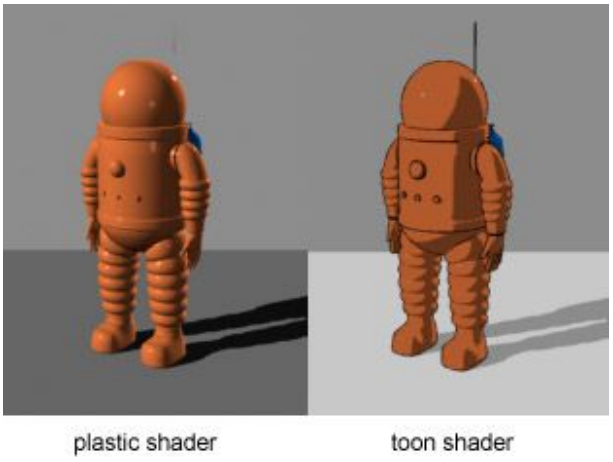


Fig. 14 - Toon Shading example

In our case we implemented a variant of the cel shading which is more toon like. It's a two tone shading, meaning there is only the bright color for the illuminated side and the dark color for the overshadowed side.
To make the transition smoother a mix between the cel shading and blinn-phong was added so there is a slight gradient between the bright color and the dark color. This gives a more realistic effect without losing the toon like effect.

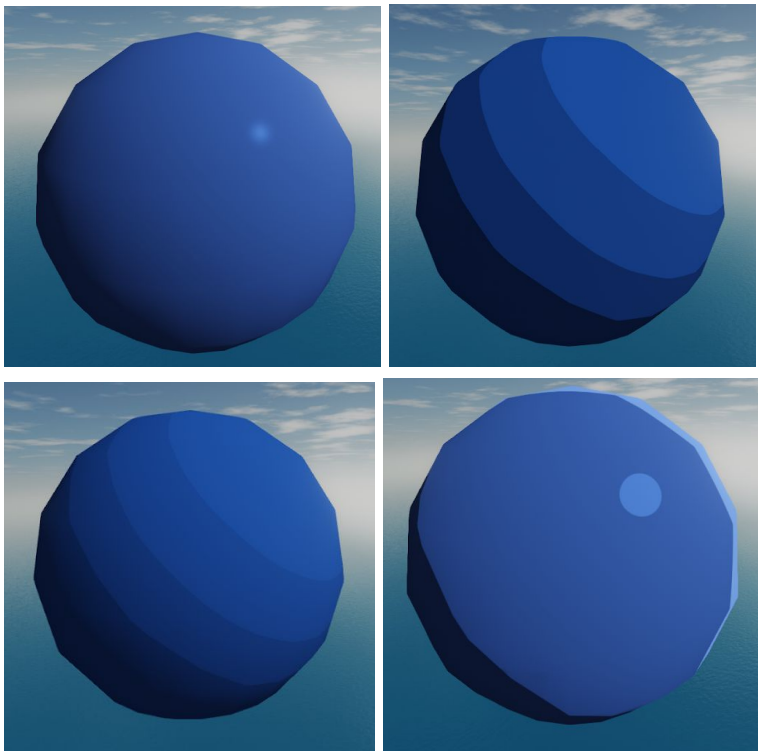


Fig. 15, 16, 17, 18 - Blinn-Phong, Cel Shading,
Mix Cel Shading and Blinn-phong,Two toned toon shader

There is also a rim effect on some surfaces, however not all of them, around some objects that complement the effect of the toon shading.
The rim is created by surfaces that are facing perpendicular to the camera (therefore the surfaces that are at the edge of an object) and facing a light source, meaning unlit surfaces do not have this effect. The effect is then restrained to a flat value using smoothstep to give a flat color. The specular light is also calculated using smoothstep to flatten the color.

3.2. Shadow-Mapping

To achieve the shadow-mapping we followed an article by learnopengl.com which gives insight on how to create it in opengl.

The idea is simple, we take a “screenshot” of the scene from the light’s perspective and everything we can see from the light’s perspective is lit and everything we can’t is under a shadow of another object.

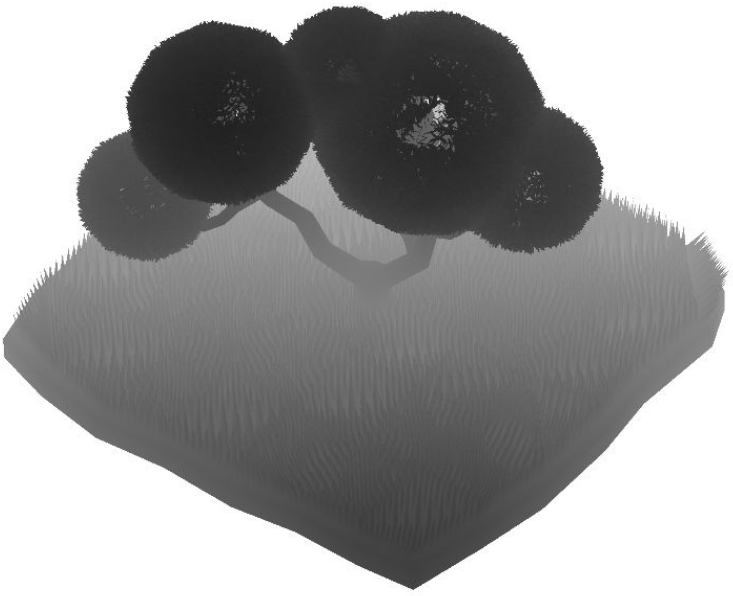


Fig. 19 - Depth map from the light source

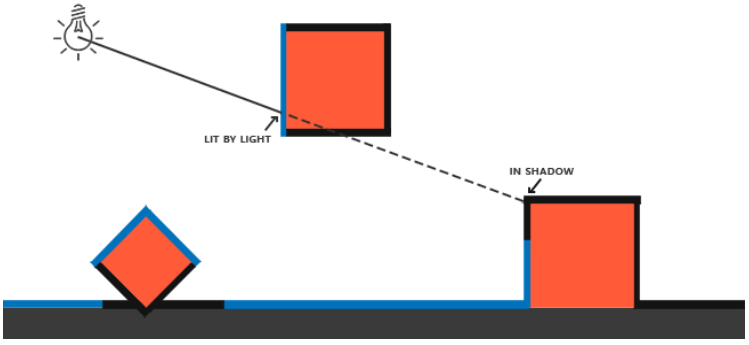


Fig. 20 - Lit surfaces and unlit surfaces

To achieve this we render the scene from a camera on the light's position and oriented with the light's direction to a framebuffer so we can use the texture later on. However what we save from the texture is not the whole scene but rather the depth value of the scene relative to the camera's position. So now besides knowing what is lit in the scene we also know how far it is, which will help us to assess if the fragments are lit or not by checking if their depth value is higher or lower than the value on the depth map rendered to the framebuffer.

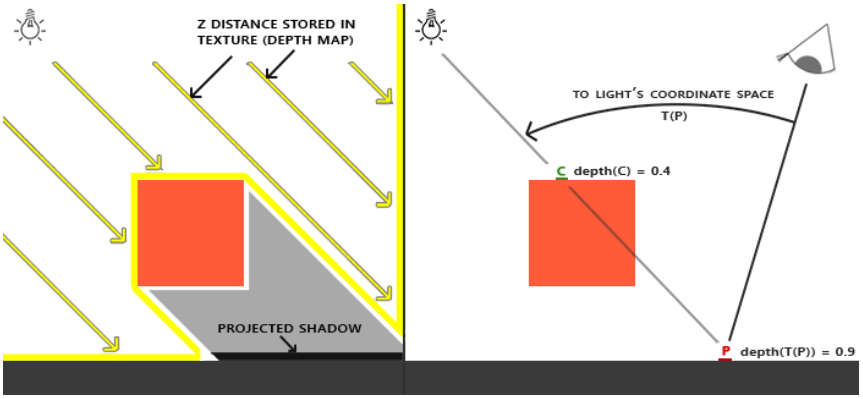


Fig. 21 Depth testing

The texture is then given to the main rendering shaders and for each fragment they can calculate the depth value from the light source's perspective and compare it to the equivalent depth value on the texture. If the value is larger than the one on the texture it means that fragment is in shadow. This on its own is not enough as often we will have several artifacts, one of the worst ones being shadow acne.

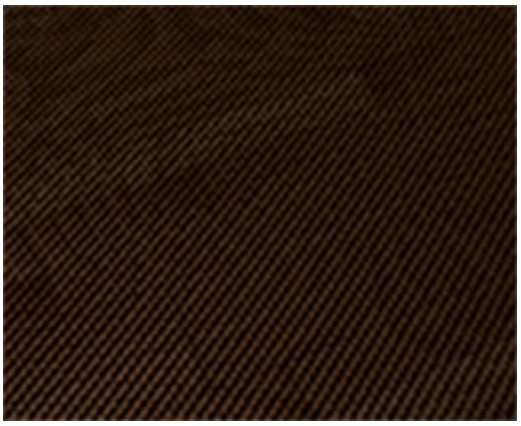


Fig. 22 Shadow Acne

It comes from the limitations of the resolution. For example when light comes at an angle the depth map is also rendered at an angle, which can create situations where the texel is considered over shadow because it considers the fragment below the surface rather than the above.

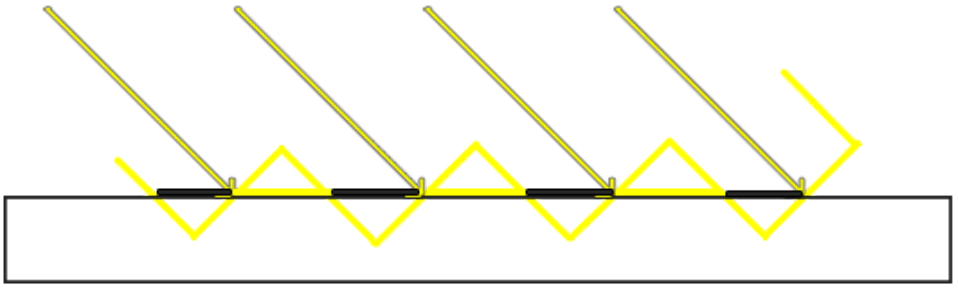


Fig. 23 Surface inconsistency

To fix this we subtract a bias that pushes the depth of the surface and makes it a little bit smaller, this makes the depth always a bit smaller than the original so that it isn't mistakenly assuming the fragment is under the surface.

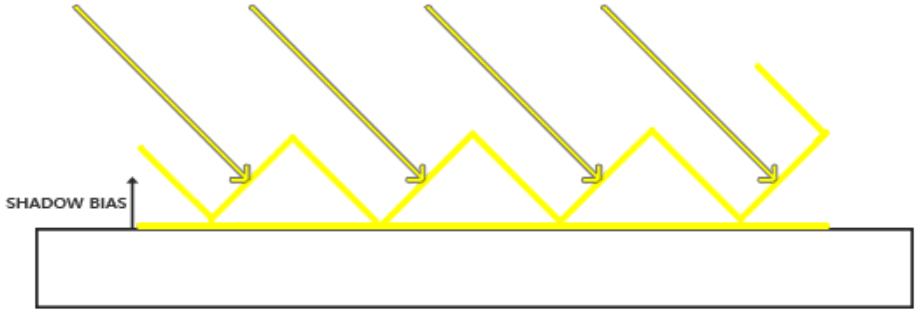


Fig. 24 Shadow Bias

The shadow acne however is dependent on the angle of the light to the surface, so we don't always need to have a big bias, we can define a minimum bias and a maximum bias that takes in consideration the angle of the light. The larger the angle, the bigger the bias.

There is however a small drawback to adding a bias. Shifting the depth value can offset the position of the shadow by a short distance, making the shadows seem displaced from the object. This effect is called peter panning.
However this problem is only prevalent in shadows close to the object casting them and is not a visible problem in our project. There is a small amount of peter panning in the scene but is not really visible at first glance, and it didn't affect the presentation too much with the bias values we used.

Another visual problem are jagged edges around the shadows. This comes from the dependency on the shadow map's resolution, as many fragments may depend on a single pixel of the shadow map and because of this they may all use the same shadow value, creating blocks around the edges of the shadow. The simple and obvious solution is increasing resolution, making less fragments depend on the same value. However, this quickly becomes computationally expensive.

There is also a technique called PCF that allows us to make multiple samples for the same fragment by using slight variations of the coordinates and getting an average of the samples' depths. The more samples we use, the less blocky the shadows usually get. There are still some artifacts when looked at from close up, but in general it looks better than the jagged look.

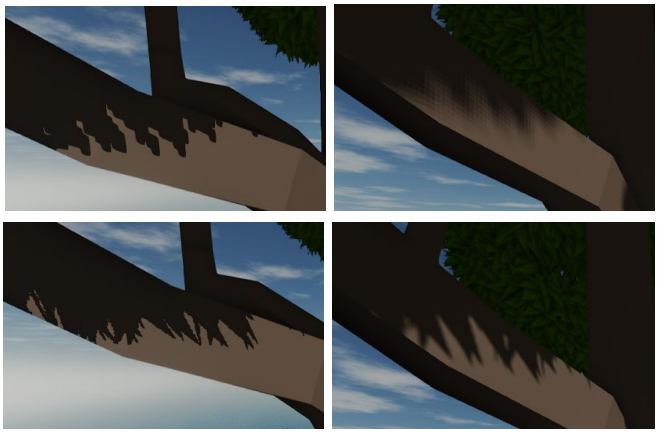


Fig. 25, 26, 27, 28 - TopLeft: No PCF low Res, TopRight: PCF low Res, BottomLeft: No PCF high Res, BottomRight: PCF high Res

3.3. Screen Space Ambient Occlusion

To implement this effect we followed the article by learnopengl.com. The effect works by sampling a number of points around each fragment to see if each point is occluded by another fragment.
To achieve this effect, we must first render each fragment's position and normal in view space to auxiliary textures. This will allow us to later calculate the amount of occlusion for each fragment taking into account the position of the fragments around it, sampling in an hemisphere around each fragment oriented by that fragment's normal.

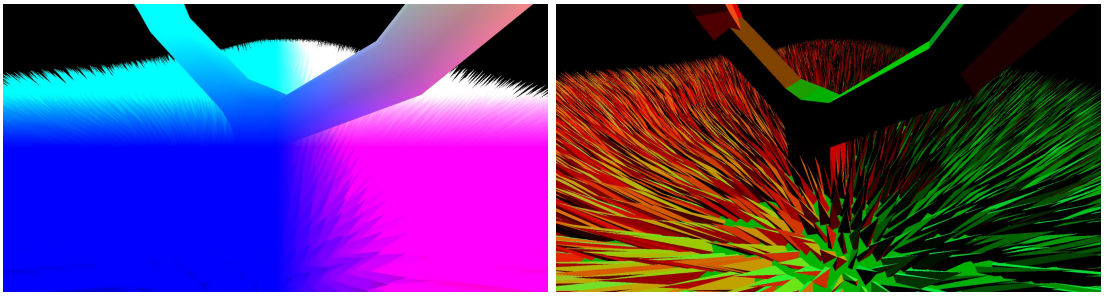


Fig. 29, 30 - Left: Position Texture, Right: Normal Texture

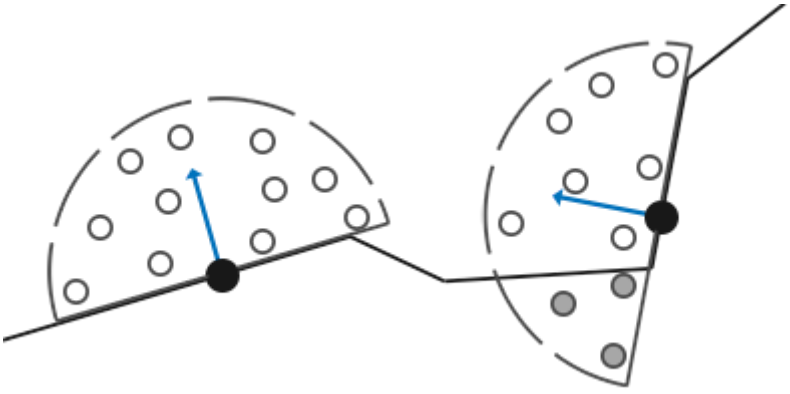


Fig. 31 - Sampling hemisphere, grey points are occluded.

As described in the article, simply sampling around each fragment's position requires a prohibitively high amount of samples to yield good results, with a lower number of samples resulting in banding artifacts. We can get around this by using a lower number of samples and introducing some randomness into the sampling by rotating the sampling hemisphere around a random vector. This vector can be precomputed, by storing a few of these vectors in a small texture with the xyz values stored as the rgb for each pixel, with this texture then being tiled over the whole screen.

Fig. 32, 33 - Left: Banding Artifacts, Right: Using random Rotation.

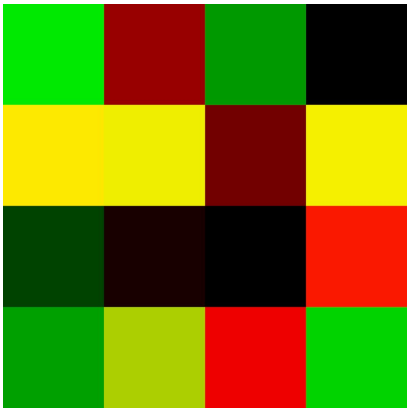


Fig. 34 - Example of a 4x4 random vector texture.

This random rotation will result in some noise, which we can fix by blurring the final occlusion texture. Given that we know that the noise texture will repeat every *n* pixels, we can use an *n* by *n* blur kernel to effectively hide the artifact.

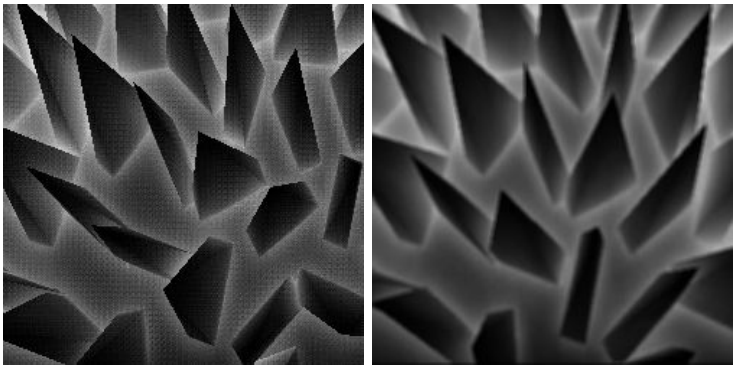


Fig. 35, 36 - Left: Before Blurring, Right: After Blurring.

We can also reduce the number of samples by concentrating them more around the fragment, with less samples as you get away from the center of the hemisphere. We can do this because samples closer to the fragments position would have more impact on the amount of light that would reach it, and thus having more samples closer will result in a more accurate estimate of the occlusion value in the area that has more visual impact.

After the occlusion values have been calculated, they can easily be used in a fragment shader by providing it with the appropriate texture. In our case we use it in our toon shader, multiplying by the ambient component of the light. Overall it greatly enhances the depth perception of objects and makes them easier to distinguish from one another, especially those in shadow, given that they would all have the same ambient component without the occlusion value.

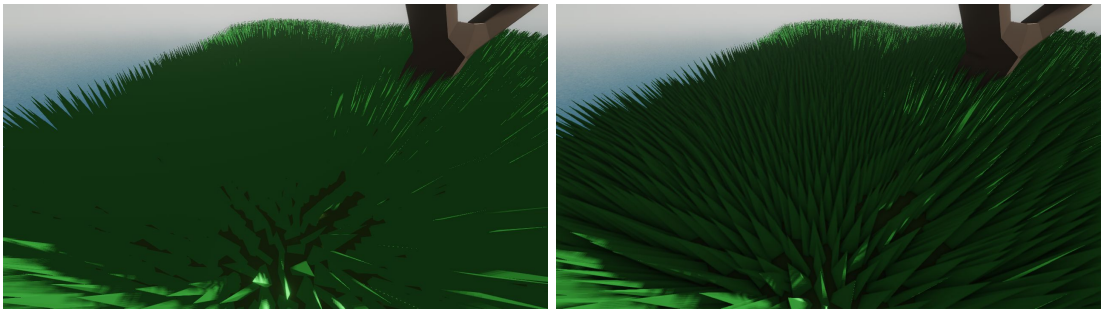


Fig. 37, 38 - Left: Without SSAO, Right: With SSAO.

3.4. Post-Processing Stack

To improve the final look of our scene we developed a post processing stack. To achieve this we use two textures, one to serve as a source, the look of our scene before "applying" a particular post processing effect, and another to serve as a destination, to store the result of the effect. We divide post-processing into a sequence of post-processing passes, with each pass being a particular post-processing effect. After each pass runs, we swap the pointers of each texture, so the output of the previous pass becomes the input for the next one.
After this architecture was in place, creating each effect can be done by simply creating a new pass and inserting into the stack. We implemented a few effects that we believed would enhance the visuals of our particular scene, some of which we will now describe. Auxiliary effects were also included in our projects, such as Dithering and FXAA, but given that we simply included them as quality of life features and did not provide our own implementation, we will not be describing them in detail.

3.4.1 Bloom

Bloom is an effect which simulates the light bleeding from bright areas in an image to the areas around them, which gives the observer a better idea of how bright objects in the scene really are. To develop our version of bloom we based ourselves on articles by learnopengl.com, Charilaos Kalogirou and catlikecoding.com.
The first step in our bloom effect is to extract only the bright areas from the source image. To do this we calculate the brightness of each pixel by using the relative luminance of each pixel using the formula below and only considering those above a certain threshold.

$$Y = 0.2126R + 0.7152G + 0.0722B.$$

Fig. 39 - Formula for relative luminance.

Achieving good quality bloom with shaders presents one main challenge: *blurring*. In order to achieve a strong light bleeding effect, we must be able to perform a large blur on the images brightest pixels. This is easier said than done, since large blur kernels are very expensive to apply, requiring access to thousands of pixels to calculate each fragment's final value. This becomes aggravated by the resolution at which we are rendering our scene, since the same blur kernel becomes smaller as the resolution increases. We can however use this same principle to our advantage if we reverse it: the smaller the resolution of an image the larger that blur kernel will be on it.
With this we can achieve a far reaching bloom effect by progressively downsampling our image and applying the same blur kernel to each iteration. This will result in a progressively blurrier image, given that the more we downsample the image the larger its pixels become and thus the blur kernel affects a larger area. In our implementation we found that a 5x5 gaussian kernel was more than sufficient for blurring the image. We can then composite each of the iterations to get a relatively high-quality and believable bloom. However, this will still present some noticeable artifacts resulting from the process of upsampling the lower resolution textures, as the default bilinear filtering provided by opengl isn't sufficient for this purpose. This becomes more evident the more we downsample the original image.
To solve this issue we can either perform multiple blur passes on each iteration, which becomes expensive rather quickly, or we can progressively upsample each texture in the same fashion in which we progressively downsample the source. We decided to go with a mixture of both, as performing even as low as 2 blur passes per iteration significantly improved the quality of our bloom. To do this we go from smaller to larger texture, drawing the contents of the previous texture into the current texture. By progressively upsampling the image we apply multiple bilinear filtering passes to it, resulting in a smoother appearance and less artifacting This results in a particularly clean looking bloom with a manageable performance cost.

Fig. 40, 41 - Left: Normal Upsampling, Right: Progressive Upsampling.

When compositing each iteration we can also give a smaller weight to the smaller textures to further reduce artifacts and concentrate the light bleeding closer to the origin of the light.

3.4.2 HDR Tone Mapping

Most displays cannot display colors with enough dynamic range to properly convey effects like bloom. Furthermore, most lighting effects also suffer from this limitation, with brighter color values simply being truncated by default, resulting in a loss of detail in brighter areas.

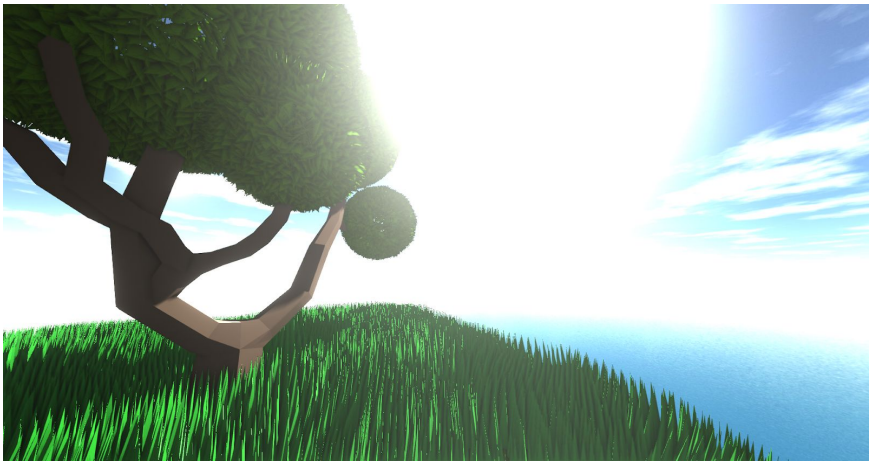


Fig. 42 - With HDR Tone Mapping disabled, most of the skybox's detail is lost

To fix this we can apply a process called tone mapping to the final image. Tone mapping essentially maps floating point color values to the low dynamic range supported by most displays.

The first step to enable the use of tone mapping is to switch to floating point color buffers, which store color values as floating point numbers instead of the usual 0-255 integer value for each color component. This is relatively straightforward in openGL, as textures natively support floating point storage formats such as GL_RGBA16F and GL_RGBA32F.

In our implementation we used Reinhard Tone Mapping as described in an article by learnopengl.com. This algorithm evenly maps HDR color values to the LDR spectrum, and given it's simplicity we decided it would be adequate for our implementation. This resulted in a more balanced image, with bright areas still maintaining detail in an LDR display.

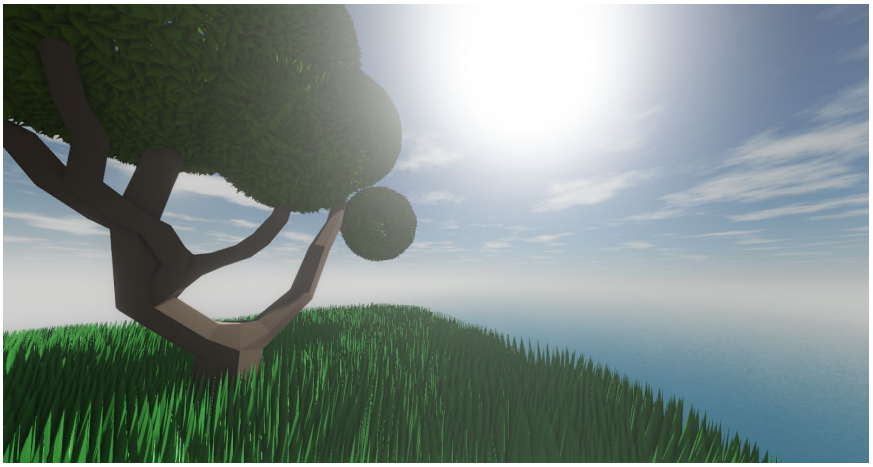


Fig. 43 - Tone mapping brings back the lost detail

3.4.3 Depth of Field

Depth of field is the zone which is in focus from a camera's point of view, with objects becoming noticeably blurrier the farther they are away from it. Simulating this effect in a 3D application can give it a more cinematic look, while also helping with depth perception.

We based ourselves on the technique described by Filip Nilsson and Philip Ljungkvist to achieve a realistic depth of field effect. The core of this implementation relies on an optical phenomenon called Circle of Confusion, in which areas which are out of focus are seen as a mix of the colors in a circle around them, with this circle being larger the less in focus they are.

To do this we must first calculate the values of the circle of confusion for each fragment. These values are calculated depending on each fragment's distance from the camera (it's depth), the sensor's aperture, the plane in focus and the image distance (distance between the sensor and the lens in the case of an actual camera). The meaning of these values is better described in the paper where this technique is detailed. To mitigate artifacts in which unsharp objects in front of sharp objects we then apply a small blur to the CoC values.

We are then ready to apply the effect to the final image. To do this we sample in rings around each fragment within range of that fragment's CoC value, sampling segments of each ring and finally averaging the color of all the samples. This resulted in some artifacts, namely sharp objects bleeding color into unsharp objects behind them, this can be easily mitigated by checking if a sample's depth is too far from the fragment's depth, instead adding the fragment's own color to the samples if it is.

The next big problem is performance. This effect is very performance intensive, given that each fragment must sample several segments of several rings around it. To alleviate this performance hit, instead of sampling a certain amount of segments per ring, we can sample less segments for lower CoC values, which will result in faster computation for areas of the image which are more in focus.

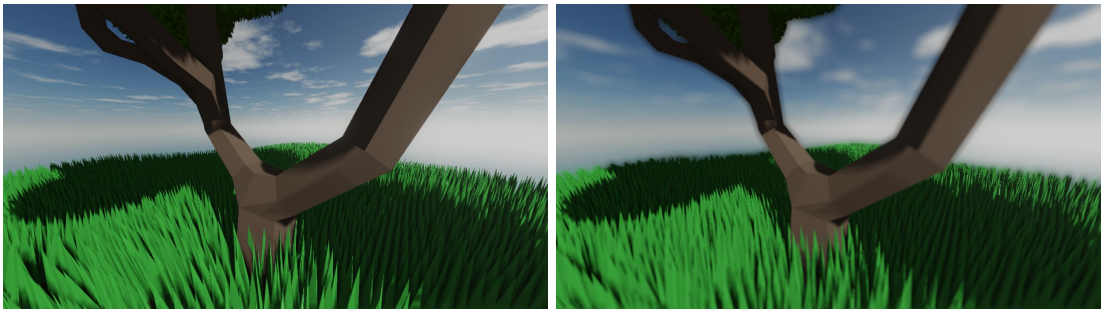


Fig. 44, 45 - Left: Without DoF, Right: With DoF.

3.4.4 ImGui

As more and more effects and components were integrated into our application, tweaking values and testing each component started to become increasingly more frustrating when trying to keep track of different keybindings, and the need for a more traditional GUI became apparent. Due to time constraints, we decided to integrate ImGui into our project. ImGui is a very straight-forward graphical user interface library for C++, which allows us to make simple windows and adjust common values such as floats, colors and booleans with appropriate interfaces such as sliders, color pickers and checkboxes with very little code and low requirements from the base

application. Given the simplicity of this approach we decided to use it for tweaking most of the parameters of our application. This also meant that we could make most of the application's systems user controllable, with most parameters of post-processing, light rendering and tessellation being available for the user to tweak and view their effect in real time.

Fig. 46, 47, 48, 49 - Some of the menus implemented with ImGui.

3.5. Skybox + Sun

Creating the skybox and the "sun" box consisted in multiple challenges: mapping six textures to a cubemap, creating the sun, creating a smaller transparent skybox in which to put the sun. An additional challenge was taken to change the skybox in order of the color selected on the ImGui tool while in runtime. We based ourselves on learnopengl.com articles to tackle these challenges.

The cubemap contains six textures, one for each face, which means glTexImage2D is also called six times, each with a different texture target. To import transparent textures, a fourth channel (alpha) had to be added when needed. Since a cubemap is a texture like any other, its wrapping and filtering methods must also be specified.

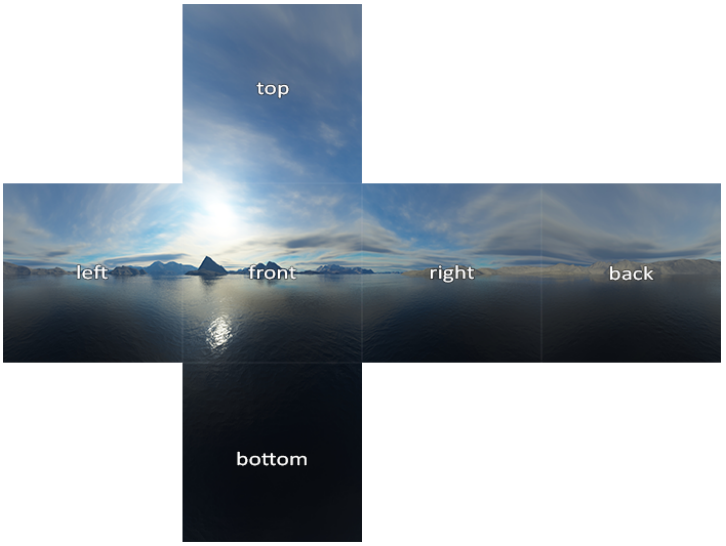


Fig. 50 - Cubemap textures pattern

The skybox is then loaded. Its faces are retrieved from "Models/Skybox/tropical" for the skybox we see and from "Models/Skybox" for the transparent skybox that has the sun. The sun was created in Gimp and is a simple white circle in the centre of a transparent-square png.

The previously mentioned file and 5 same-looking transparent-square, create all six faces for the transparent skybox with the sun. To encompass the entire scene, the skybox is scaled with a factor of 100 and the transparent skybox with a factor of 99.

Finally, to tackle the additional challenge where the skybox changes color according to the light's current color. First we must convert the skybox color and light color to the HSV color space and apply an offset to the skybox's hue equivalent to the light color's hue. Then, we convert the colors back to RGB and calculate a mix of the original skybox color and the hue shifted version, depending on the light color's saturation. This will result in more saturated light colors having a stronger effect on the skybox's color. We then convert this mixed color back to HSV and scale the value component according to the light color's value component. Finally, the fragment's color is converted back to RGB and multiplied by the luminosity component, which indicates how bright the skybox will be (a value of 1.2 for the sky and a value of 10 for the sun). Multiplying by these values will naturally push the slyboxes' colors into the HDR spectrum. This is intentional as it means that they will be handled correctly by the bloom and tone mapping passes to yield colors that appear visibly brighter than the rest of the scene.

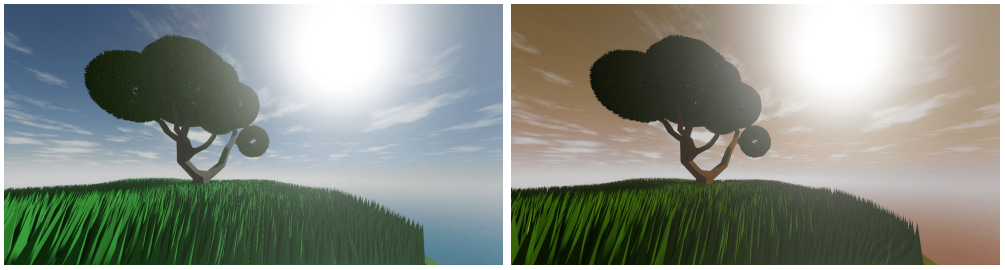


Fig. 51, 52 - Left: Default light color, Right: Orange/Afternoon light color

3.6. Sun Movement

The sun movement was done one way, but was then adapted for the final version. Initially, the sun position (the centre of the front texture of the transparent cubemap) changed by using the arrow keys: by pressing up and down, the transparent skybox's rotation on the X axis decreased and increased (clamped between -89 and 89 degrees); by pressing left and right, the transparent skybox's rotation on the Y axis increased and decreased.
In the final version, the skybox's X and Y rotation depends on the value set on the ImGui interface, which still allows for clamping: -90 to 90 degrees on the X axis and 0 to 360 degrees on the Y axis.
Changing the transparent skybox's rotation also changes the light source position, not affecting its direction.

$$x = cos(sunRotation.y) \times cos(sunRotation.x)$$
$$y = sin(sunRotation.y)$$
$$z = cos(sunRotation.y) \times sin(sunRotation.y)$$
$$Lightpos = sunRadius \times vec3(x, y, z)$$

Fig. 53 - Light position according to transparent skybox's rotation

3.7. Screenshots

We decided to integrate the FreeImage library to make the screenshot feature possible. The screenshot feature also had two versions: the initial one used the F key to take the screenshot, while the final one depends only on one ImGui button.

Pressing the “Take Screenshot” sets a flag, which when checked copies all pixels drawn on the main frame buffer. The ImGui interface does not appear on the taken screenshot since it’s only drawn after the flag check.
All screenshots are sent to a “Screenshots” folder, which is created when the program starts. When they are taken, their name will be “Screenshot” followed by the number of screenshots already taken and finally by its format “.bmp”. We also made sure that if there are already screenshots inside the folder, that these do not get deleted and the new names take this into account.

3.8. Terrain composed of grass

3.8.1 Tessellation

The first challenge consisted in applying the Tessellation technique to a simple surface. First we started by tessellating a triangle, which consisted in subdividing the triangle into smaller triangles on the fly. To do this we needed to update our graphics pipeline by adding two new shader stages, the Tessellation Control Shader and Tessellation Evaluation Shader. In between those two we have the Tessellation Primitive Generator which is not controlled by a shader program.

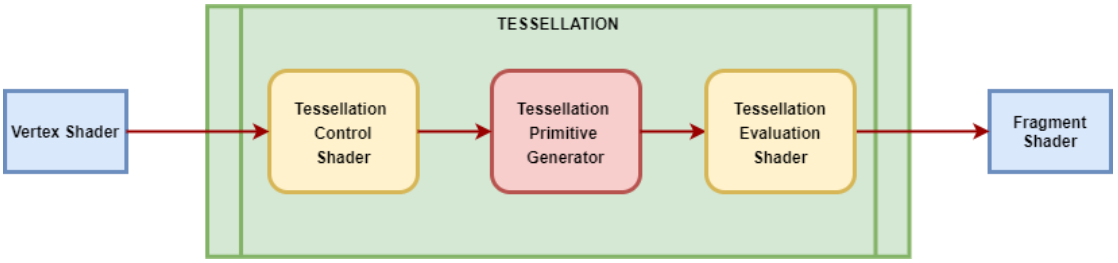


Fig. 54 - Updated Graphic Pipeline

The Tessellation Control Shader controls the size and how much tessellation a patch can get, its main purpose is to feed the tessellation levels to the Tessellation Primitive Stage as well as the patch output values to the Tessellation Evaluation Shader stage. In the Tessellation Control Shader we used the built-in patch output variables *gl_TessLevelOuter* and *gl_TessLevelInner* to define the outer and inner tessellation levels used by the Tessellation Primitive Generator that is responsible to do the actual subdivision. These two variables are also responsible for defining how much tessellation is applied to the patch.

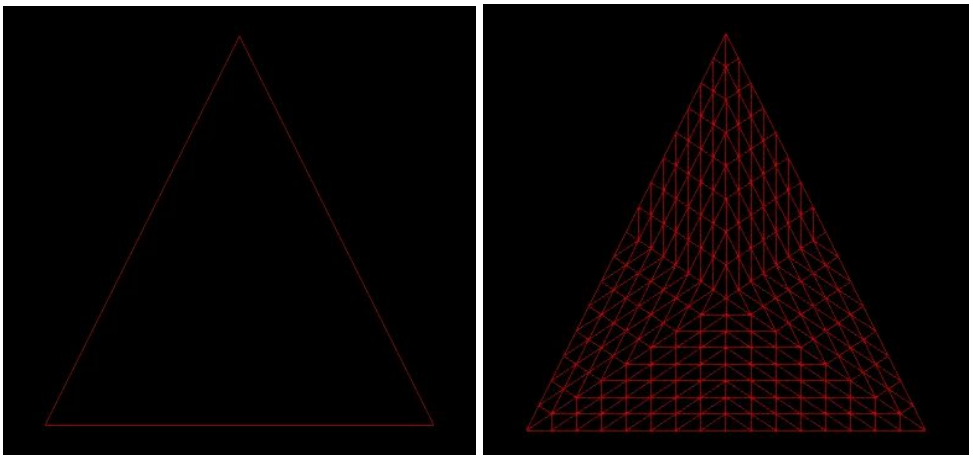


Fig. 55, 56 - Left: Inner level = 0 Outer level = 0, Right: Inner level = 16 Outer level = 16.

Once Tessellation Primitive Generator is done with the subdivision the barycentric coordinates and their connections are sent to Tessellation Evaluation Shader as well as the patch output (from Tessellation Control Shader). Then the Tessellation Primitive Generator executes the Tessellation Evaluation Shader on every barycentric coordinate (by computing the interpolation of the coordinates) generating this way a single vertex for each coordinate, after that these vertices are sent to the next stage of the pipeline that might be the Geometry Shader or other stage. The following result was achieved:

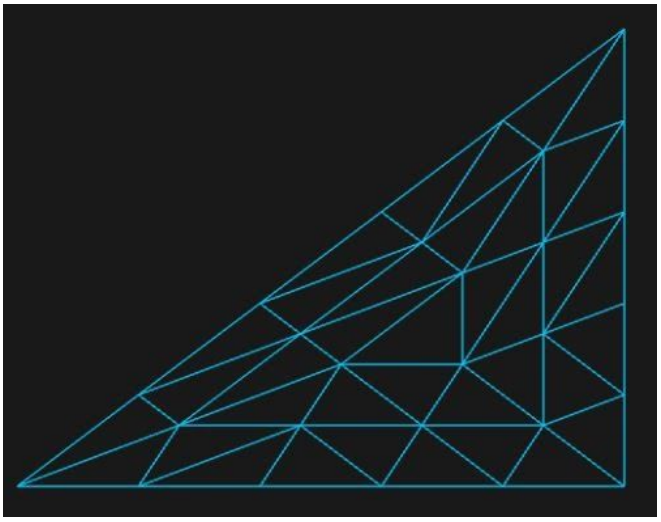


Fig. 57 - Tessellated triangle

The next step consisted in applying the same workflow to a 3D polygon like a cube, but this time we decided to also interpolate the normals in the Tessellation Evaluation Shader and then we used the Geometry Shader to visualize the normal vectors as explained in this tutorial. Basically, we took each triangle primitive and generated from them three lines in the same direction of their normals. Our first approach was trying to draw the grass blades following the same direction of the normals. This was the result achieved:

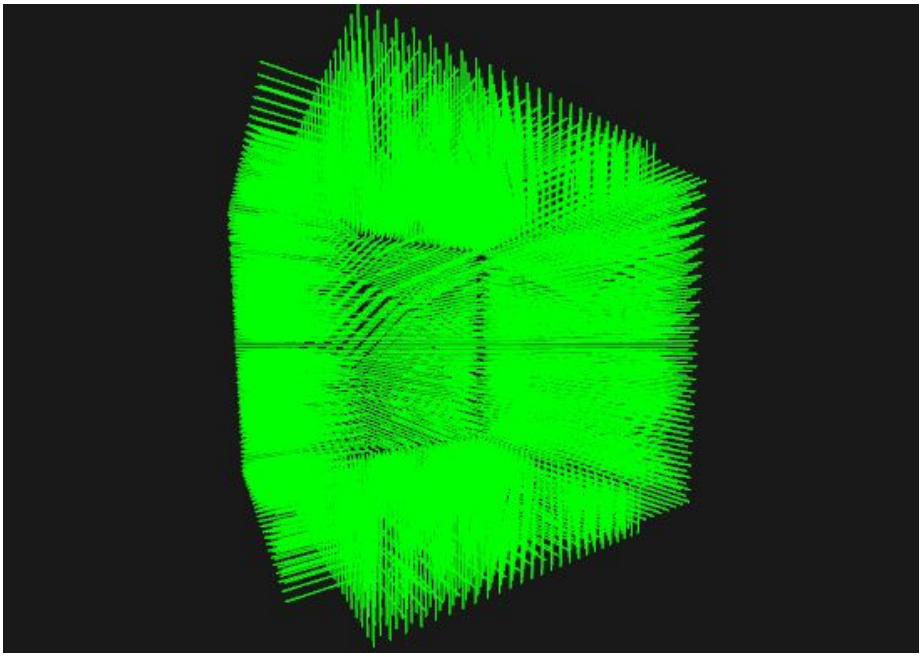


Fig. 58 - Tessellated Cube with lines following the normals direction vector.

After that learning process we were ready to apply it to any low polygon model, therefore we could start replacing the lines by the grass blades, but first it was necessary to choose between a low polygon model or a textured quad to represent the grass blade and the leaves. For aesthetic reasons we thought that a low polygon model like a small pyramid would fit perfectly for the grass blade while the textured quad was chosen for the leaves. Also to get a more organic appearance we decided to apply small random rotations around the surfaces normal to each grass blade and each leaf. Thus the final result achieved:

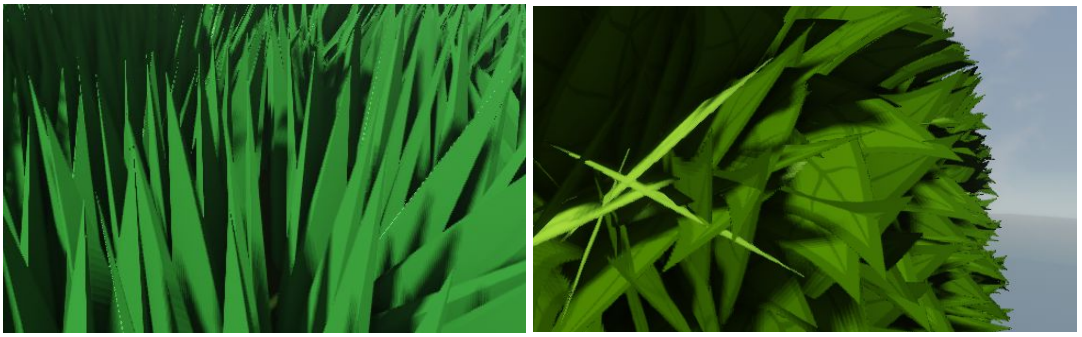


Fig. 59, 60 - Left: Grass blades, Right:Tree leaves

3.8.2 Flowing Grass

Our first approach for this challenge was using a sinusoidal function that would be affected by external factors like offsetValue, the original vertex position (Rootpos), elapsed time period (T), random generated number and finally wind speed. With that in mind we developed the following formula to calculate the wind power:

$$Windpower = Offsetvalue + sin(Rootpos \times Rndnumber + T \times Windspeed)$$

Fig. 61 - Formula for flowing grass movement.

Then we applied the wind power together with the wind direction to the top vertex of the low polygon grass blade in local space coordinates. Although the first result was satisfying it had some strange behaviours like stretching the grass blades (perfectly visible when we increased the WindSpeed variable) and if we were to apply the same behaviour to the leaves the end result wouldn't be pleasant as desired and the movement of the grass wouldn't match with leaves movement.

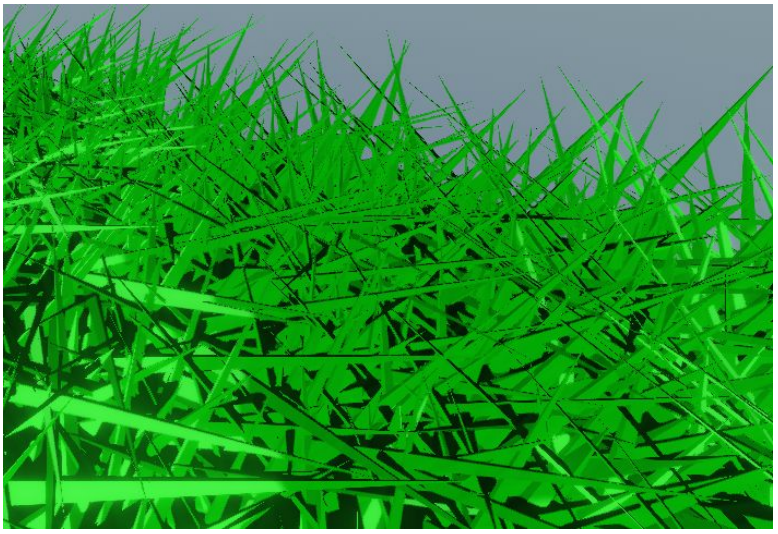


Fig. 62 - Grass blade stretching

It was clear to us that another approach should be taken and with that in mind we came across the idea of applying a 3D noise function. Due to time restrictions and having simplicity in mind, we chose to use a Simplex Noise implementation in glsl developed Patricio Gonzalez Vivo which allows for constructing a 3D noise function similar to Perlin Noise but with fewer directional artifacts and lower computational complexity. For each coordinate we calculate the noise value using the following formula:

$$Noisevalue = noise(Noisepos + Rootpos \times Noisescale)$$

Fig. 63 - Noise Value calculation

The noise scale indicates how dense the noise function should be, with higher values simulating denser changes in air pressure. RootPos refers to the position of the grass' or leaf's root generated by the tesselation stage. The *Noisepos* variable is the current 3D offset within the noise function's domain, which is a result of time and the wind's direction. This position is thus updated every frame with the following formula:

$$Noisepos = Noisepos + Winddirection \times Windspeed \times Seconds \times Noisescale$$

Fig. 64 - Noise Pos calculation

With this we simulate the “scrolling” of this noise function, which can be seen as the wind moving along the scene.
Then, before applying the rotation matrix to each grass/leaf, we apply that noise value affected by the wind’s strength, which defines how much that specific object will be affected by the wind, and by a directional vector that results from the cross product between surface normal and the wind direction vector. This directional vector is responsible for the orientation of the flowing movement. We also added a small random factor to the rotation that made the flowing movement less uniform and consequently more realistic. The result is way more satisfactory than the first approach and can be observed in the demo that was delivered.

4. Post-Mortem (max 2 pages)

4.1. What went well?

The initial development of the techniques was done separately and all group members seemed to achieve the desired results before having to integrate the code. After the hurdles of integration were surpassed the following work went smoothly and the expectations met reality in a beautiful scene where all techniques implemented achieved the visual result we desired.

The feeling of a peaceful breeze was achieved and we can even change the overall ambiance of the scene, from a summer breeze to a sunset.

4.2. What did not go so well?

The main issues we faced had to do with integration of each others’ code. Many issues arose when trying to integrate the tesselation components with the rest of the code base. We believe most of the complications were a result of this technology being the one least known to all of the team’s members, and as a result much of the code presented incompatibilities with the rest of the code due to us essentially “coding in the dark”. Having worked on this together from the start would have massively improved the development process, but this was not possible due to schedule constraints. Having more joint work sessions would also have improved our development speed.

4.3. Lessons learned

All four students had previous knowledge of opengl and had all previously participated in the CGJ course. This means that a lot of obvious mistakes we would usually make were avoided and our experience with opengl made for a faster development than usual meaning less constraints of time. It’s good to see that the CGJ course was efficient in giving us proficiency in opengl.

We learned a lot of different techniques and all went really well. If we would start again, maybe work more together in the beginning and integrate sooner. The integration days were a bit harder than they should have been, had we prepared sooner for it it could’ve gone more smoothly.

For those following a similar path, meaning having already previous knowledge and testing it again in a similar course, we think it's best to choose an ambitious project to really test your skills and the knowledge you've gained. We could've added more complexity to our project as many features were quickly implemented and finished sooner than expected.

References

- Concept Inspiration - https://tinyurl.com/y6sukk4c
- Toon Shader - https://roystan.net/articles/toon-shader.html
- Shadow Mapping - https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
- SSAO - https://learnopengl.com/Advanced-Lighting/SSAO
- Bloom effect - https://learnopengl.com/Advanced-Lighting/Bloom
- How to do Good Bloom for HDR Rendering - https://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/
- Bloom: Blurring Light - https://catlikecoding.com/unity/tutorials/advanced-rendering/bloom/
- HDR - https://learnopengl.com/Advanced-Lighting/HDR
- Implementing realistic depth of field in OpenGL - https://fileadmin.cs.lth.se/cs/Education/EDAN35/lectures/12DOF.pdf
- ImGui - https://github.com/ocornut/imgui
- Cubemaps - https://learnopengl.com/Advanced-OpenGL/Cubemaps
- FreeImage - https://freeimage.sourceforge.io/
- Geometry Shader - https://learnopengl.com/Advanced-OpenGL/Geometry-Shader
- GLSL noise algorithms by Patricio Gonzalez Vivo - https://gist.github.com/patriciogonzalezvivo/670c22f3966e662d2f83
- How to fix color banding with dithering - https://www.anisopteragames.com/how-to-fix-color-banding-with-dithering/
- glsl-fxaa - https://github.com/mattdesl/glsl-fxaa