Leysen Floris

Long-term sci-fi vegetation  
progression simulation

Graduation work 2019-20

Digital Arts and Entertainment

Howest.be

Contents

[Abstract 2](#_Toc23151505)

[Introduction 3](#_Toc23151506)

[Research 4](#_Toc23151507)

[1. Model to represent the system 4](#_Toc23151508)

[2. Spreading and growth patterns 4](#_Toc23151509)

[2.1 Spreading patterns 4](#_Toc23151510)

[2.2 Grass-like growth 4](#_Toc23151511)

[2.2.1 Sod-forming versus bunching 4](#_Toc23151512)

[2.2.2 Warm versus cool season 5](#_Toc23151513)

[2.2.3 Growth stages 5](#_Toc23151514)

[2.3 Life cycles 5](#_Toc23151515)

[3. Surface detection 6](#_Toc23151516)

[4. Influence of light on growth 6](#_Toc23151517)

[5. Optimization 8](#_Toc23151518)

[Case study 10](#_Toc23151519)

[1. Engine choice 10](#_Toc23151520)

[2. Surface detection 10](#_Toc23151521)

[3. The factor of light 11](#_Toc23151522)

[4. Optimization 12](#_Toc23151523)

[Conclusion 13](#_Toc23151524)

[References 14](#_Toc23151525)

[Appendices 15](#_Toc23151526)

# Abstract

**An abstract explains the aim of the paper in very brief, (the methods, results, etc.). Maximum length 150 words**

# Introduction

Creating a large vegetation environment to use in a game is a task that many have strived to simplify – Unreal has a built-in procedural foliage tool and Ubisoft’s Far Cry 5 allows vegetation painting in its map editor – as doing it manually is usually not worth the effort. All of those efforts are targeted towards merely filling up an environment, yet none allow users to have the vegetative contents change over time.

This research aims to simulate the progression of those contents over a period of several years – or as long as you wish. The influential elements that are specifically considered include how plants spread, general responses to seasonal changes, growth cycles, and influence of light, while larger scale outside factors such as weather, animals or the planet’s rotational changes are left out. Plants can have their rules defined, meaning they ultimately determine which elements are important and how the world it takes place on works.

Since the intent is to use this generator inside a game, creating a system that is equal to reality is not possible. This means the simulation’s nature will tend to lead to a result less realistic than carefully hand-crafted vegetation. Because of that limitation, environments feel less familiar, but still grounded in reality. A science fiction setting takes place in the same reality as humans live in, though in a world beyond current understanding. This similar feeling makes it ideal for the expected result. *No Man’s Sky* is an example of a game with a sci-fi environment that is not manually produced.

Figure 1: Example of a No Man’s Sky environment (Hybridpaolo, 21 August 2016, [reddit.com/4yuhxz](http://reddit.com/4yuhxz))

Because the system is meant to be utilized for a game, performance is of importance. It is, however, not intended to follow the real runtime of the application, only going in steps of one day each. These steps are meant to be taken not while a player is inside of the environment, but during some other time – probably during loading. As a result, performance is not as critical as it would otherwise be. Nevertheless, optimization techniques for a model of this type are looked into.

The result of all of this is a project that accurately simulates the progression of vegetation in a predetermined environment, based on seeding, in a game engine.

# Research

## Model to represent the system

Before considering which model is best, it is important to define the characteristics of the intended complete system. Expandability and maintainability are crucial, so adding a new plant, removing one or updating an existing one should not necessitate any changes to the system. It is also crucial to keep complexity of individual vegetation elements low, while keeping an intricate larger scale evolving environment.

One model that has been extensively used in biological simulations is an agent-based model [[1]](#AgentBasedBiology). Agents define it and the system itself is kept simple, only being a framework for the agents to get information from. It’s a model based on emergence, meaning the complete environment possesses features that cannot be observed in its agents. Those two elements make it a perfect fit for our goal, though other possibilities were not ruled out.

A multi-agent system is one that is highly similar to an agent-based model, though both its agents and environments are of greater complexity than the aforementioned model [[2]](#AgentBasedModel). No individual agent is allowed to have a view on the entirety of the system, something that the plants in the simulation do need to know in order to spread. That unfortunate characteristic makes it a less viable model and it can be discarded.

Cellular automaton is a grid-based model, with each cell of that grid being in a certain state. Rules are defined for the updating of states within cells, customarily maintaining those rules over time and remaining the same between cells. This model has seen use in biological simulations, too, but for decidedly simpler results like patterns. Since the aim is to have any number of varying plants with different rules, this model was also deemed inappropriate.

Considering these advantages and disadvantages, the agent-based model was determined to indeed the best fit for the planned system.

## Spreading and growth patterns

### Spreading patterns

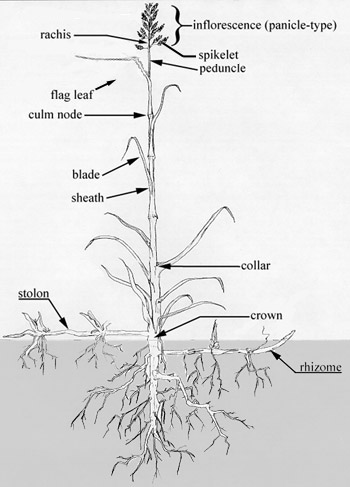
Currently no research is available on accurately predicting patterns of which direction grasses will spread their roots and at what pace, thus eliminating the possibility of considering this element for the simulation. Precisely determining where reproductive seeds will end up is influenced by too many factors – wind, animals, gravity… – to be usable. As a result, both root-type spreading, and seed-based spreading are based on assumptions of maximum and minimum expected distance from their source, unless specified otherwise by the rules of a specific plant.

### Grass-like growth

There are numerous types of grasses, sometimes having vastly different properties from each other. It is therefore more appropriate to analyse those properties individually, allowing combination at will. Ignoring non-grass specific traits – i.e.: height and lifespan – the differentiations can be limited to sod-forming versus bunching and warm versus cool season grasses. Reproductive spreading is omitted, too, due to not being exclusive to grasses.

### Sod-forming versus bunching

In nature, grasses of the sod-forming type form a large interconnected network both above ground – a connection like that is called a stolon – and below ground – called a rhizome. This manner of spreading allows new shoots – plants that are individually sustainable, but part of the network – to grow from a stolon, rhizome or even right at the ground level of another plant [[3]](#GrassManageMent).

 Figure 2: Stolon and rhizome [[4]](#BasicsOfGrassGrowth)

Bunching types, unlike sod-forming grasses, can only grow new plants at the soil surface of an already present plant. Grasses with this property cannot grow as closely packed as its counterpart, though the bunches themselves can be. This does increase the odds of an invasive plant appearing in the empty space between two clumps.

### Warm versus cool season

While both types employ the same method of energizing themselves through photosynthesis, the construction of their leaves is different. The deep science behind these behaviours is unimportant, but the result of it is.

Grasses meant for a cooler season are more active – growing and reproducing – during the more temperate seasons of spring and autumn, while growth is slowed down in the warmer season of summer and they might even become dormant. The warm season grasses, on the other hand, are at their peak productivity during the hot summer season. In terms of nutritional value, the cool season grasses are more densely packed.

### Growth stages

When a seed meets the right conditions to germinate, a single leaf will start growing. In this vegetative stage, leaves will continuously keep appearing and dying. Photosynthesis is the plant’s main purpose now. Changing of climate can cause the next stage. Here, the grass prepares itself for reproduction and reduces energy spent on other tasks like growing. With time, the final stage of flowering is entered. At this point, growth is completely halted, while flowers spawn. Reproductive spreading happens and depending on if the type of grass it might go dormant until the next growth season or die away entirely.

Other plants share their growth stages with grasses for the most part, with the main difference being they can still grow during the reproduction-related stages.

### Life cycles

There are three main types of plant life cycles: annuals, biennials, and perennials. The annuals go through all their growth stages just once, within a period less than a year, after which they die. Biennials need about two years to fully grow and die, using the first year to stock up on nutrients and reproducing in the second.

The perennial variation is a tad more complex than the others, possibly surviving indefinitely until struck by rot or other outside factors. While they can keep growing forever, their advancement is restricted to their growing season. Reproduction can happen every year, though if it does, the period of growth is reduced. Not all plants of this type can live endlessly, the only true requirement to be considered perennial is to survive for longer than three years [[5]](#PerennialPlants).

## Surface detection

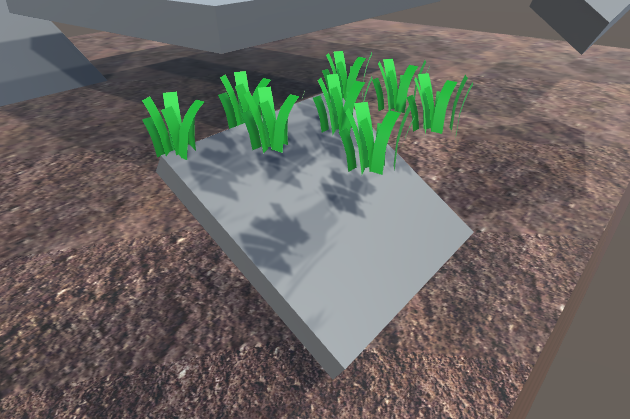
The goal is to detect which surfaces are inside a defined volume and store them for later use in plant placement. Determining which objects are inside a volume is trivial when working in a game engine, so the only unknown is how to extract surface data from it.

One option is to check every individual triangle on the mesh and interpret each of those as a surface, then determining their orientation. This method introduces one major problem; if working on meshes with significant triangle counts, determining which surface is appropriate and available gets highly complex and can take a long time, especially when the variable of other plants taking up space is accounted for.

A second option would be to determine a surface only at the exact time something needs to spawn, e.g.: raycasting from an approximated location and spawning on the hit position. This method is promising, since it allows users to make changes to the environment without having to recalculate where everything is positioned. One great concern is keeping track of which spots other plants are already occupying, since there is no predetermined set of locations.

Since both have a serious flaw, but boast great advantages, too, some middle ground had to be found. Basing placement on a grid solves the problem of maintaining information on already inhabited locations. If the location of plants follows this defined grid, the issue of no longer looking natural appears. This could be solved by making the grid cells extremely small. Doing as such regrettably defeats the purpose of it more simply and efficiently storing locational data. Instead, the grid can be interpreted as cells to remember which cells are occupied and which plant is in that cell.

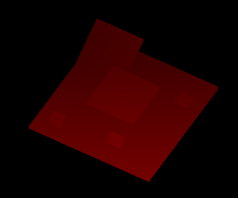
This does not provide a solution to the actual placement, though. Simplifying the mesh-based surfaces, creating simple rectangular boundaries for each object inside yields a similar result. In irregular geometry, this results in floating vegetation, which is then solved by executing a raycast to place the object on the surface.

 Figure 3: Floating grass bunches

## Influence of light on growth

Rather than focusing on mimicking reality, the impact factor of light on the plants’ growth is mostly up to the user to define. Instead, this part of the research aims to calculate whether an area on a surface is in shadow or not and how intense that shadow is.

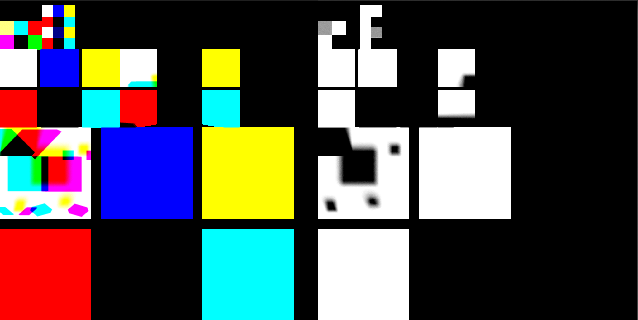
Interpreting the shadowmap at runtime gives easy access to the visible shadows and their intensity. The immediately noticeable and fatal problem with this approach is that it only renders from the perspective of a single light source, while the simulation needs information on the total level of shadow.

 Figure 4: Shadowmap for a directional light

Baking the lights in the scene and using the data the baked lightmap provides was the next step. Right away, there was a problem. It does not automatically have import settings which allow reading from and/or writing to the texture. A simple fix: manually baking lighting, generating the necessary files in the process – these can simply be set to allow read/write. If the automatic bake is a requirement, the alternative is to copy the texture to another one – created in code – and use that one for reading and writing. Do note that this alternate technique is considerably less efficient, especially when multiple lightmaps are used.

Now it is merely a matter of reading the lightmap bake at the correct coordinates. Unity’s raycast provides a variable which holds a coordinate on the lightmap UV. Unfortunately, some factors can result in that value being nonsense. To prevent this, the object to check must naturally be marked as ‘lightmap static’ and ‘generate lightmap UVs’ before baking the light, as well as given a non-convex mesh collider.

The generated lightmap has too many variables to reasonably consider – colour and amount of lights present, skybox.... Conveniently, it is not the only texture created by baking the lighting information. The generated shadow mask is a great deal more straightforward. Up to four lights are stored in a single image, with each light occupying the values of precisely one channel. Those values range from 1 – meaning the pixel is completely inside the light, so no shadow is present – to 0, which signifies the hardest shadow. Combining the different lights’ values leads to a reasonably accurate result of how dark a certain position is.

   
Figure 5: Shadow mask for four lights, RGB channels to the left and alpha channel to the right

Truly calculating the average time spent inside shadow over 24 hours is sadly exceptionally complex. Therefore, an estimation is more applicable. Since the sun always raises in the east, rises to the north (or south, depending on the hemisphere) and sets in the west – or takes some other predefined path depending on the user’s idea of the planet this takes place on – it is possible to place up to four lights defining that path to make the shadow mask as representative as possible. The system can then interpret this mask and average out an acceptable approximation.

## Optimization

A tool to be used in games is worth very little if the performance of the result does not run at an acceptable framerate. While the simulation’s calculation times will only matter at times it should not interfere with the actual gameplay, the calculations of the system should still be limited to a point where it does not take more time to calculate a single day passing than it takes to render a frame – otherwise it is required to hide it in loading and any meaningfully long time progressions are out of the question.

The models themselves are created by users of the tool, so rather than looking at optimizing meshes, it is more desirable to provide an option to improve their performance through the information of the system. Since it is undesirable to have every object be a single blade of grass, making bunches more efficient is the target.

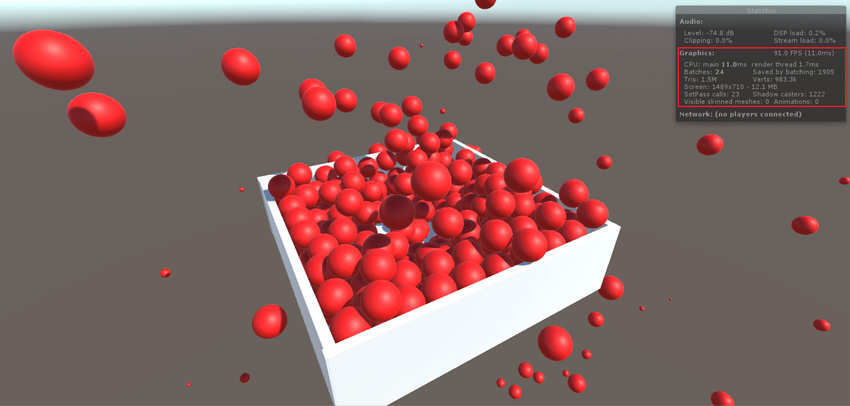
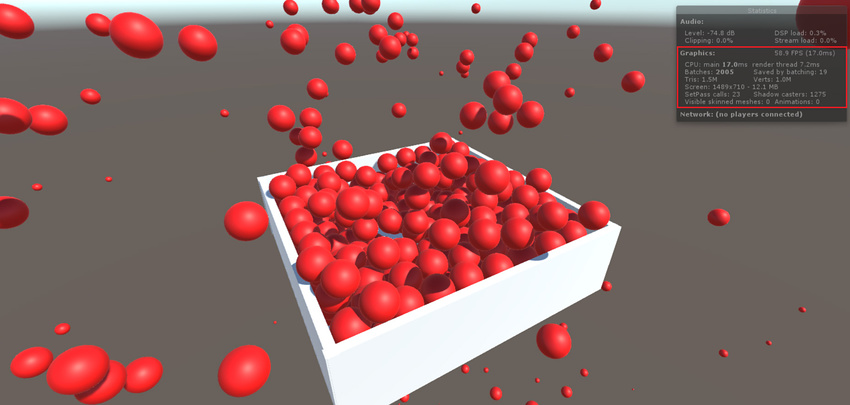
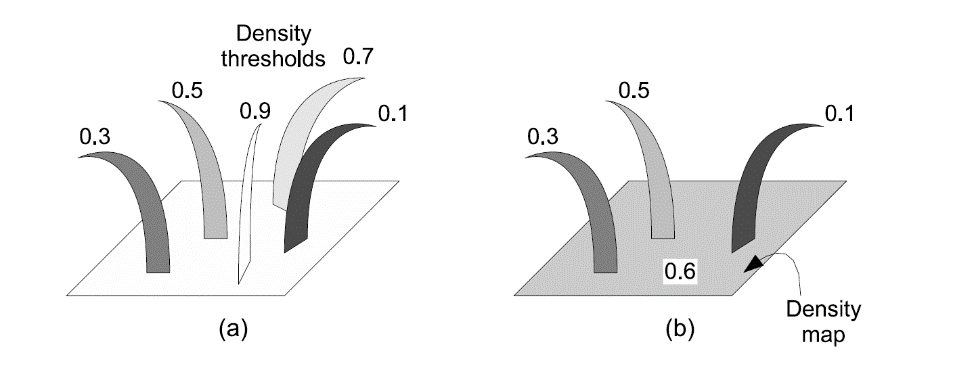
The same plants will appear many times at once in the simulation, so the first step is to enable instancing. Doing as such allows multiple identical meshes to be rendered within the same draw call, but still allows some parameters to vary between instances – most meaningfully the transform. This tackles a large optimization hurdle, but there is still more to improve, knowing the segmented nature of the most likely used models.

Figure 6: Many identical GameObjects, with instancing enabled (left) and without (right). [<https://docs.unity3d.com/Manual/GPUInstancing.html>]

It is not always necessary or even desirable – e.g.: when growing or in areas that should feel less dense – to always render all of the leaves part of a single representative mesh, meaning this is another opportunity to investigate possible gains in performance. Since it is part of the rendering of the object, this is ideally done in a shader, as part of the fragment shader[[6]](#OptimizationThesis). When created, each leaf – for shader purposes, each vertex in that leaf – is assigned a density value. A density map either created in advance or based on some at-runtime variables is sampled whenever the model should be rendered and any leaves that don’t have a sufficient density value are discarded.

  
Figure 7: Each leaf is assigned a density value and only those with low enough density value compared to the map sample are rendered [[6]](#OptimizationThesis)

# Case study

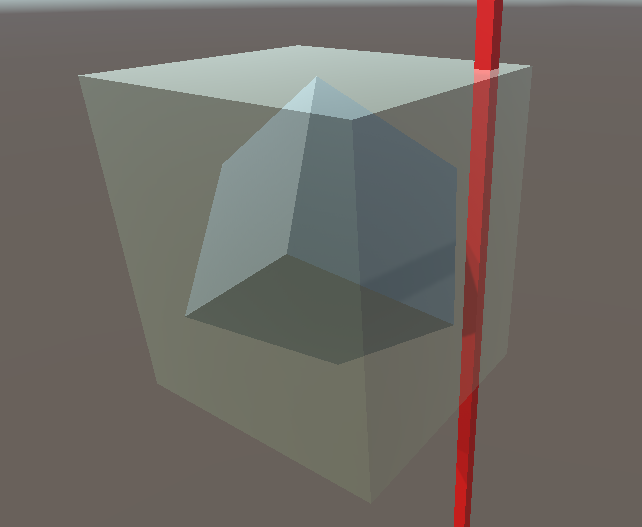
## Engine choice

Since the focus lies on making a system that is useable in a game, it is only natural to pick an engine that is both widely used and accessible. This already narrowed down the choice to Unreal or Unity. Unreal has the advantage of beautifying the scene and vegetation more easily, while the strength of Unity lies in its ease of use and shorter development times – also for developers who want to expand on the system. Because of the intended style of vegetation being sci-fi, the generally less realistic style of Unity is no longer a major negative, thus triumphing over Unreal’s realism.

## Surface detection

Several elements are part of the found surface detection- and tracking solution. The system needs a grid, which is a component already available by default in Unity, so no need to reinvent the wheel. It should also detect surfaces that are inside of the volume from the start but allow changing of the environment. For this purpose, two collision layers are added – one for the system and one for the environment pieces. The system should only ever collide (trigger) with that other defined layer. When such a trigger enter happens, we add the collider to the surfaces to keep track of – and remove it if it leaves.

Whenever the seeds are supposed to spawn on some random location inside the volume, the surface detector will determine which surface is nearest to that location using the bounds of the colliders. Now performing a simple raycast from the position exclusively to the object – after making sure to place the y position high enough to not miss – is enough to determine where on the mesh it would land… Unless the shape does not fill the x-z plane of its bounds, as seen in figure 5. This issue can be solved by simply moving the testing position closer to the centre of the object and attempting again.

   
Figure 8: Mesh that fails to fill the x-z plane of its bounds, transparent cube indicating boundaries and red cuboid showing possible ray miss

Now that a position has been acquired, it is possible to check whether it is already occupied in the grid or not. Again, Unity provides functionality which allows positions to be converted to grid coordinates. Ownership of grid coordinates is maintained in a dictionary, where keys are coordinates and values are plants.

## The factor of light

Using the research for this element, an implementation is quite simple. As it explains, in the properties of all mesh assets part of the bake, the option “Generate Lightmap UVs” should be enabled. Any objects inside the scene that should be taken into account for the shadow mask are marked “Lightmap Static” in their Mesh Renderer. Everything inside of the simulation volume that is on the layer “VegetationEnvironment” is required to use a Mesh-Collider that is not marked convex in order to allow raycast hits to gather the lightmap UV coordinates for their hit points.

Since the generated images are not automatically marked to enable reading and writing and they always have a predictable automatically generated name, a simple editor script completely erases the inconvenience of repeatedly marking them for use.

**Custom editor script to mark all baked shadow masks to allow read/write. Needs to be placed in an “Editor” folder to work.**

using UnityEditor;

using System.Text.RegularExpressions;

class AllowReadWriteBakedShadows : AssetPostprocessor

{

private Regex \_pathMatcher = new Regex("Lightmap-\\d+\_comp\_shadowmask.png");

private void OnPreprocessTexture()

{

if (!\_pathMatcher.IsMatch(assetPath)) return;

TextureImporter textureImporter = assetImporter as TextureImporter;

if (textureImporter)

textureImporter.isReadable = true;

}

}

As for the actual sampling of the masks during the simulation, a dedicated shadow mask sampler is created. At start, it inspects all present shadow masks and determines how many lights are present in each of them. If a channel ever has a value higher than 0, it means a light is affecting that channel. The result is stored in a bitmask for each map. E.g.: if three lights are present, the R, G and B channels will have non-zero values and the bitmask is of value seven. With this information, the calculation of the shadow masks’ average on a certain pixel is reduced to merely averaging the existing lights’ channel values.

**Function that returns the average shadow on a hit. The shadow mask has a value of 1 for no shadow and 0 for max shadow, so it is inverted.**

public float CalculateShadowFromHit(RaycastHit hit)

{

Vector2 uvCoord = hit.lightmapCoord;

int lightIdx = hit.collider.GetComponent<MeshRenderer>().lightmapIndex;

Texture2D tex = LightmapSettings.lightmaps[lightIdx].shadowMask;

uvCoord \*= tex.height;

Color color = tex.GetPixel((int)uvCoord.x, (int)uvCoord.y);

return 1 - CalculateMaskAverage(color, lightIdx);

}

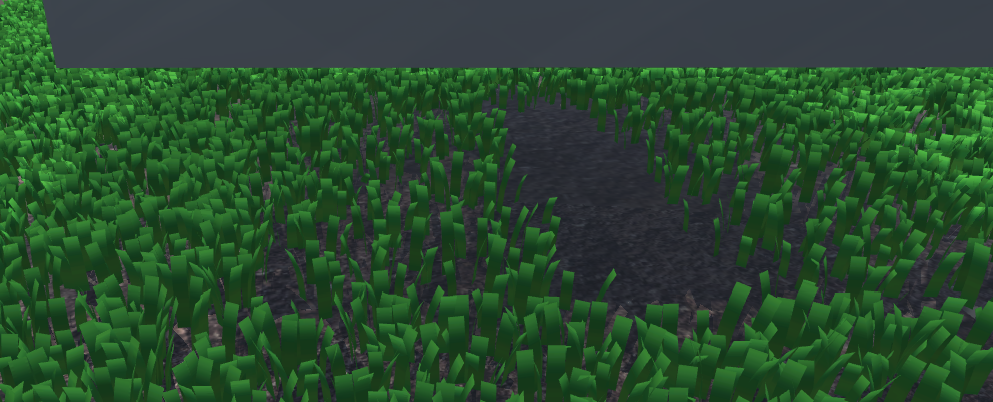
Whenever a plant is created, it is provided with the ‘shadow value’ at its position. If the information is needed later (e.g.: to determine the speed of growth), the plant can simply access its ‘Plant’ properties and retrieve the shadow factor from there. See appendices (extra figure 1) for an example of shadow awareness.

## Optimization

To start implementing the elements found through the research, the first step is to create a shader which handles the fragment discarding and GPU instancing. Fortunately, Unity allows built-in GPU instancing to be enabled on any shader created, meaning that part is already taken care of.

The steps to take are ultimately not very complex, unless the user intends to utilize the meshes’ vertex colours for a purpose other than storing the density. Since overwriting other parts of the mesh data can lead to visible changes, the colour array is the most harmless element of the mesh to save the density threshold to. First, the leaves are assigned their density value – randomly, as to not have every area of similar density look the same. After that, the instanced mesh is provided the data to later interpret for discarding fragments.

Regrettably, Unity’s fragment shader is not based on primitives, but rather on individual pixels. At the time of writing, there is no way to remove vertices, polygons, or anything more than a pixel from the pipeline. This completely invalidates any optimization that applying this technique could provide. It is, however, still possible to clip the fragments – i.e.: pixels – that fall under the density threshold of the area in order to get a result that is visually nearly identical to the primitive discarding solution.

  
Figure 9: Leaf density reduces towards the area under the most intense shadow factor

## Different plant types

# Conclusion

**repeat the main topics, discuss your main findings, discuss the end result.**

# References

[1] An, G. , Mi, Q. , Dutta‐Moscato, J. and Vodovotz, Y. (2009), Agent‐based models in translational systems biology. WIREs Syst Biol Med, 1: 159-171. doi:[10.1002/wsbm.45](https://doi.org/10.1002/wsbm.45)

[2] Agent-Based Model. (n.d.). In Wikipedia. Retrieved September 30, 2019, from: <https://en.wikipedia.org/wiki/Agent-based_model>

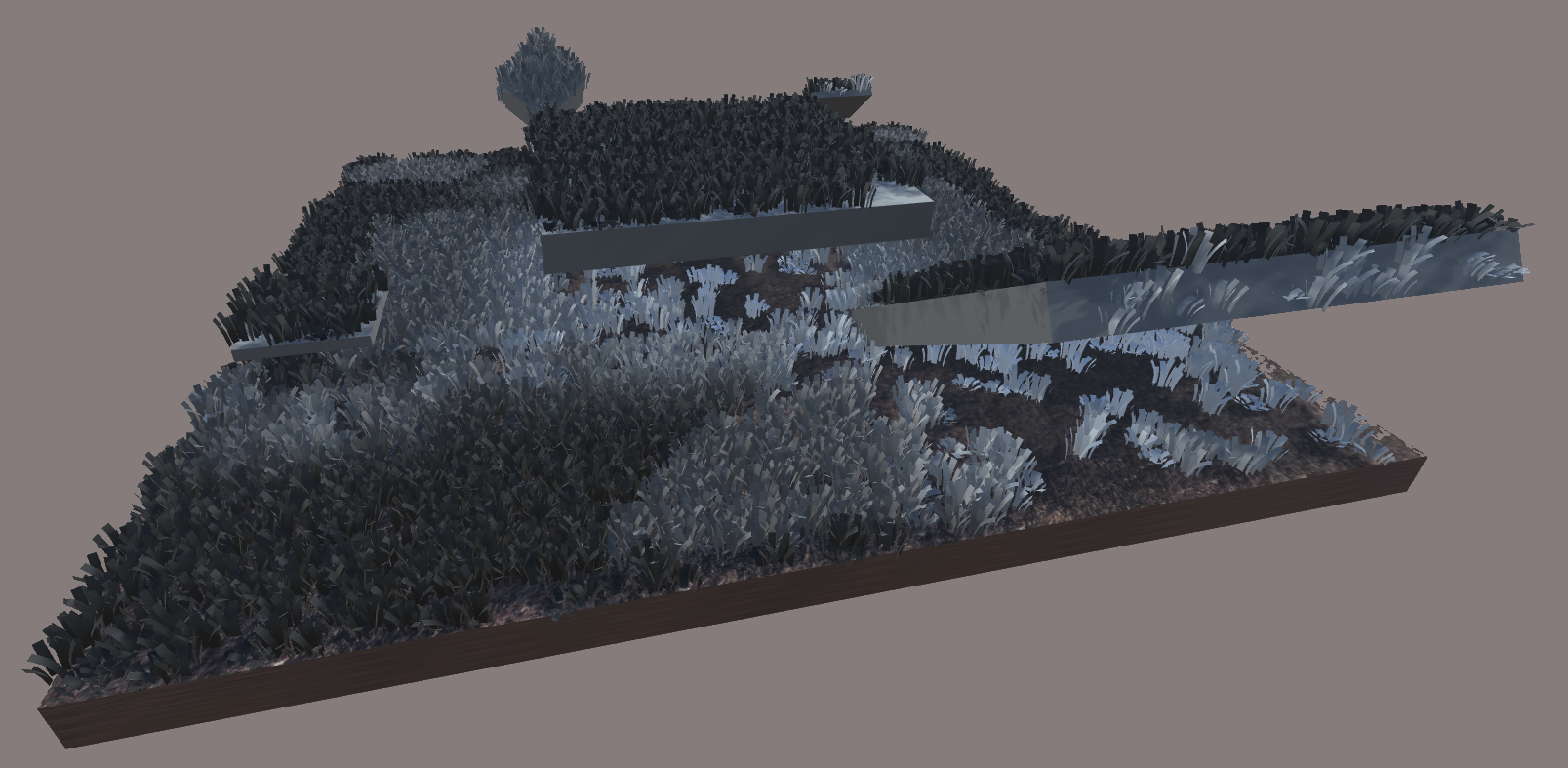
[3] Oregon State University. Discuss the basics of grass growth. Retrieved 1 October 2019, from the Oregon State University website: <https://forages.oregonstate.edu/nfgc/eo/onlineforagecurriculum/instructormaterials/availabletopics/management/growth>

[4] Oregon State University. How does grass grow? Developmental phases. Retrieved 7 October 2019, from the Oregon State University website: <https://forages.oregonstate.edu/regrowth/how-does-grass-grow/developmental-phases>

[5] Schacht, W., Sandall, L., Fick, W., Perennial Plants, Retrieved 7 October 2019, from: <https://passel2.unl.edu/view/lesson/50d18f39cff4/2>

[6] Boulanger, K. (2008). Real-time realistic rendering of nature scenes with dynamic lighting (Doctoral dissertation). Retrieved from <http://www.kevinboulanger.net/publications/thesisKevinBoulanger.pdf>

# Appendices

  
**Extra Figure 1: Test where all grasses are set to have all of their colour values equal to the shadow factor on their location. The higher the shadow factor, the more difficult it is to grow there.**