Generating Procedural Environments using Masks

Layered Image Document to Real-time environment

Emil Eldståhl

Computer Graphic Arts, bachelor's level
2019

Luleå University of Technology
Department of Arts, Communication and Education
Preface

This bachelor thesis was written by Emil Eldstål. It concludes my bachelor studies in Computer Graphics at Luleå University of Technology. I have been working on this thesis while doing my internship at The Future Group in Oslo.

My goal with this assignment was to create a suite of tools to automatically and procedurally generate real-time environments using Layered images (Photoshop files) as the input. The goal was to produce a tool that enables the end user to paint out simple level design ideas and quickly get a good-looking result in a game engine.

*Emil Eldstål*
Sammanfattning


Abstract

This paper will explore the possibilities of using an automated self-made procedural tool to create real-time environments based on simple image masks. The purpose of this is to enable a concept artist or level designer to quickly get out results in a game engine and to be able to explore ideas. The goal of this thesis was to better understand how you can break down simple ideas and shapes into more complex details and assets. In the first part of this thesis, I go over the traditional workflow of creating a real-time environment. I then go on and break down my tool, what it does and how it works. I start off with a Photoshop file, make tools in Houdini and then utilize those in Unreal for the end result. I also argument about the time-saving possibilities with these tools. From the work, I draw the conclusion that these kinds of tools save a lot of time for repeating tasks and the creation of similar environments.
# Table of Contents

1 Introduction 5
   1.1 Background 5
   1.2 Problem 5
   1.3 Purpose 5
   1.4 Limitations 6

2 Theory 7
   2.1 Procedural generation 7
   2.2 Splines 7
   2.3 Parameter based generation 7
   2.4 Decals 8

3 Method 9
   3.1 Tools and programs 9
   3.2 Creating data from PSD files 9
   3.3 Terrain generation 9
   3.4 Spline generation from bitmap 10
   3.5 Spline creation in Unreal 12
   3.5.1 Power Lines tool 12
   3.6 Foliage diversity 12
   3.7 Reactive foliage layers 13
   3.8 Method critic 14
   3.9 Flowchart 16

4 Result 17
   4.1 Terrain 17
   4.2 Roads 18
   4.3 Cars 20
   4.4 Farm 20
   4.5 Input/Output 21
   4.6 Animals 22
   4.7 Editor Scripts 23

5 Discussion 25

6 Conclusions 27

7 References 28
1 Introduction

1.1 Background

The visual effects industry is growing and the demands of content creation rise as well. Consumers want more and more content. So a lot of companies are turning to a more procedural workflow to get out more content more quickly. Instead of having 10s of artists working on repetitive tasks you can have one technical artist doing a procedural workflow tool that does the same thing. The initial time investment and skill required are usually required for these types of tasks. But when you have it in place, it is easy to pump out lots of content and make variations easily. Repetitive tasks and tasks that follow a ruleset can be automated or proceduralized in various ways. I have been interested in procedural generation the last year or so. The GDC talk\(^1\) about the terrain tools and generation for Ghost recon wildlands sparked the interest in making my own procedural terrain generator with props and foliage based on image inputs.

1.2 Problem

The time to get in-game results decreases steadily in the VFX and games industry. But the workflow has to improve as well to keep up with the demands. While the team for Ghost recon hand-made a hugely detailed world map with giant proportions for their game I wanted to test whether I could produce reliably consistent world maps at a smaller scale. Their approach was to use tools to help populate their map which inspired this work.

1.3 Purpose

The purpose of this paper and the development of a suite of workflow tools is to get a better understanding of procedural workflows and how you can group techniques to get advanced results. The focus is to get a game-ready environment using simple texture masks as the input. Several tools for achieving these goals will be created. The prototype will create different types of rural forest areas with roads, cars, and foliage. As the work progresses iteratively on the tools, results in different conditions will be tested to make sure they work large-scale and with different types of input data from the user.

\(^1\) [https://www.gdcvault.com/play/1024029/-Ghost-Recon-Wildlands-Terrain](https://www.gdcvault.com/play/1024029/-Ghost-Recon-Wildlands-Terrain)
1.4 Limitations

Some limitations that were set for on this project were the type of features the generator would require as well as the type of environment. The decision to use Scandinavian forests/rural areas as the theme were made. This limitation was set to focus on the different tools themselves and create a proof of concept generator. It was also chosen this as there would not have been enough time to create a plethora of house assets etc. So at a glance, these features deemed sufficient to create procedural rural Scandinavian maps:

- Forest areas with varied instanced foliage
- Farm agricultural growing areas
- Power lines with power cables
- Roads with cars animated along the path
- Basic heightmap (optional input) with procedural details added

This also aligns with the definition of a rural area: “Typical rural areas have a low population density and small settlements. Agricultural areas are commonly rural, as are other types of areas such as forest.”

Where possible, the amount of coding will be limited and will be tried to be done with visual scripting and simple scripts. Proper programming languages would be beyond the scope of this project and writer. To further reduce the scope a bit, the use and downloads of free assets will be utilized to be able to focus on the tools and not manually creating assets.

---

2 [https://en.wikipedia.org/wiki/Agriculture](https://en.wikipedia.org/wiki/Agriculture)
2 Theory

2.1 Procedural generation

Procedural generation is when you create data such as meshes, music or textures using different kinds of algorithms\(^5\). Another common word for this is random generation. The level of proceduralism can range from small rotation offsets in grass meshes to the creation of entire worlds. A big advantage of using these methods is that file sizes can be reduced significantly. Content can be generated at run-time which then can be loaded into the RAM instead of being saved to the hard drive.

For games, procedural aspects extend the content for players and free up time for creators. Procedural dungeons are commonly used in action RPGs. A procedural content system usually takes a longer time to develop than normal content. But when the system is made, getting hundreds or thousands of iterations become easy. Open world games are among the most popular use cases where procedural generation play a huge role in the amount of content and replayability.

2.2 Splines

Splines are used in many tools to create content. A spline is a mathematical function defined by polynomials\(^6\). Splines are basically curves with different kinds of methods to generate them. It is defined as a parametric curve. In level design and procedural content, splines can at ease be used for content such as roads, fences, animation along a curve among other things. Most game-engines have this supported and various help-functions such as getting a point at a certain distance along a curve. This allows the creation of complex shapes with little data compared to having linear curves with many sample points instead. The word spline itself refers to the many different kinds of functions and data interpolations that create these curves.

2.3 Parameter based generation

To better shape and tweak the results, parameters are usually utilized. For example, there can be a parameter for the color of a car, the width of a road, the type of fencing and so on. The technical artist exposes certain parameters for the level designer or artist using the tools. There can be a few simple parameters exposed to the user, or a lot of advanced parameters. Usually, parameters can be hidden depending on checkboxes used. So if a toggle switch for a fence is

\(^5\) https://en.wikipedia.org/wiki/Procedural_generation

\(^6\) https://en.wikipedia.org/wiki/Spline_(mathematics)
disabled, all parameters for that can be hidden. This removes some clutter in the User interface.

2.4 Decals

Decals are special materials/shaders that get projected onto meshes in an existing level\(^7\). This feature exists in most modern game-engines and is used primarily to add more detail in a game or environment easy. Due to the nature of how decals work, they are performance-friendly. With the use of deferred renderer, decals can be added in the same GBuffer pass (Geometry pass) without the need of recalculating the lighting. The most frequent use cases can be bullet holes, water puddles affecting multiple meshes, graffiti and so on.

The decal implementation in Unreal Engine allows you to choose which material attributes to utilize so that only the roughness pass would be used for the asphalt tire wear.

\(^7\) [https://docs.unrealengine.com/en-US/Resources/ContentExamples/Decals/1_1](https://docs.unrealengine.com/en-US/Resources/ContentExamples/Decals/1_1)
3 Method

3.1 Tools and programs

From initial pre-research, Unreal Engine 4 was chosen as the real-time engine due to experience and required features such as in-editor scripting capabilities of assets. To drive/create the data Houdini was chosen as the program to create these different sub-tools for generating assets/features. Photoshop was used to create the layered masks as Houdini can read the layer system and names from Photoshop documents.

**Unreal Engine 4.22.1**
Extremely popular game Engine used in AAA and indie games. PBR-based renderer

**SideFX Houdini 17.5.221**
Powerful 3d DCC with the extensive support of procedural generation methods and excellent data management

**Adobe Photoshop**
Photo editing and manipulation program by Adobe

**Megascan assets**
PBR Photo-scanned asset and texture library by Quixel

3.2 Creating data from PSD files

So the first steps of this project are to create and read the layers and data from the Photoshop document files. A template Photoshop document was created with these naming conventions of layers and color swatches for consistency.

layers/features

- FOREST - Green
- FARM - Yellow
- ROAD - Black
- POWER - Brown
- HEIGHT - black/white data - (Optional)

3.3 Terrain Generation

For the terrain generation, the tool detects if there exists a height-map in the Photoshop document and uses that as a baseline. It then adds some noise and a
quick blur effect on it. If there is no input height-map the tool instead use a bit stronger noise as the base and utilize the same blur effect afterward.

3.4 Spline generation from bitmap

The idea here is to take the layer for the power lines or the road and convert that bitmap into a spline for use in the game engine. This turned out to be a difficult task, there is no common simple way to convert a bitmap shape to a spline. This took a lot of time to figure out but a method to do so was used. The bitmap was split up based on connectivity (if there’s more than 1 road brush stroke for example) and then converted into a flat polygon. The polygon edges were then resampled to get a constant distance between vertices (Figure 1).

(Fig 1 - Vertex spacing evened out)

The mesh then gets triangulated to get a topology (figure 2): With that done, groups of the outer and inner edges were made (figure 2). Then, the middle vertices were divided in the middle, then triangulating them again. Finally, they were grouped and the outer vertices were removed (figure 2).
After that, the vertices were grouped based on connectivity to get them separated. Those were then looped through each “line” and the point numbers were rearranged based on the vertex order. After this, each point was raycasted against the terrain so that the points attach to the surface (figure 3).

With that done, splines were derived from the bitmap images in Houdini.
The splines then get added together. But as of writing, there is no way to get splines from Houdini into Unreal Engine as they use different systems. A custom solution was created to circumvent that problem. After each spline is made in Houdini, a vertice with 0/0/0 in position was added for each spline. A “dummy” value as a separator was used for this later on. The combined splines then get exported as a .hscv file for use in Unreal. This is a modified CSV file with more usability in game engines. The .HCSV file consists of rows of every vertice and columns with x,y and z values.

3.5 Spline creation in Unreal

With the exported .HCSV file done, the implementation to create splines in Unreal begins. To create splines from the .HCSV, several custom functions were made. The first function parses through the file and counts the number of rows with 0/0/0 in value to get the number of splines to create. The next step was to create an array of Vector2 values. This is to get the start and end vertices to use for each spline.

With these steps done a for-loop goes through the number of splines, lookup which rows to use in the .HCSV files as position data. It then adds those as spline points for each spline and generates functional Unreal Engine splines. Now there is a master script with the splines in memory for further use and a generalistic method. It can be decided where to send these splines and in what context they should be used in blueprints.

3.5.1 Power line tool

In Houdini, the spline gets taken along with the terrain as input. Then they get ray-traced up/down and so that the points get placed right on the terrain surface. With this, instances of power line poles can be placed at each point. It also creates 2 points at the end of each pole for the location of the power cables. This is then packaged into a .hda tool for use in the game engine. When the tool is dragged into a game engine level, Houdini Engine creates and spawns instances of power line poles on the landscape. It then at run-time creates the .HCSV file so that the use the spline script from the previous step can be used. This then creates a blueprint with Unreal’s built-in cable component which is a physics-animated cable. The distance between each power line gets measured and a customizable amount of additional length is added randomly to each cable. This, along with the physic simulation enabled creates cables hanging in semi-curves realistically.

3.6 Foliage diversity

Now that the foliage and grass get spawned procedurally it looks a bit bland. One might add more types of foliage but the spawning itself gets repetitive and no
area gets any special focus, it is just evenly distributed across the entire map. That is why the use of splat maps was used to counteract this. It could look something like this:

(Fig 4 - Texture used for splatting different types of foliage)

It is a tileable texture with splashes of colors put in separate R, G, and B channels in a texture file. This can be used in different ways to tell the game engine to spawn normal uniform grass in the black areas for example. Small rocks can use the red channel to only spawn in those areas in small clusters. Bushes spawn in green areas and so on. This texture map can be scaled across the map/multiplied by noise or combined. This, in turn, produces a much more interesting result of foliage and small details, as things cluster hierarchically in nature\(^8\). Small rocks often group in nature with different sized ones.

### 3.7 Reactive Foliage layers

With access to all the layers of the terrain, they get utilized to create different foliage masks for the different kinds of foliage. The mask for the power lines gets taken and subtracted that from the forest mask so that no trees are spawned in

\(^8\) [https://en.wikipedia.org/wiki/Hierarchy#Life](https://en.wikipedia.org/wiki/Hierarchy#Life)
the close vicinity of the power lines. This new mask gets reversed and used so that only small bushes and grass spawn there instead (figure 5).

(Fig 5 - Power Lines running through a forest with foliage override near)

3.8 Method Critic

Trying to do everything procedurally without much user input proved to be a difficult task. To automate things I had to do a completely different approach than normal for most parts. The spline generation was especially cumbersome. Normally you would quickly create splines in-engine and generate content by hand.

The tool parses the outer borders of the bitmap layer and then subdivides and find the middle vertices. This was only reliable working with a brush size of 30px with high hardness. The vertex orders and spline point order would also not be correct sometimes. So this method was maybe not the best one, but after extensive research, no other really obvious solution was found and applicable to this project/thesis in particular.

Procedural foliage spawner

Another method for spawning the foliage could certainly be used. A feature called Procedural foliage spawner is available in Unreal Engine⁹. The method and idea of that tool are to place rectangular box volumes, set up spawning rules and then spawn foliage. This method was tried initially, but when the complexity of the different kinds of foliage increased, the simulation times became huge. With enough density, the tool would also stop working, it would only spawn wheat foliage for example and not the forest areas or grass at all. There was also no way as of writing to procedurally spawn these volumes using blueprints or editor

⁹ https://docs.unrealengine.com/en-us/Engine/OpenWorldTools/ProceduralFoliage
scripting. The simulate button itself could not be accessed and had to be pressed manually. As the only realistic solution to this was to place several volumes by hand for each area and simulating them one by one, it was discarded.

**Houdini instancing**

The idea of using Houdini instances to spawn foliage was also considered as an alternative method. With the slow implementation of terrains in the current version of the Houdini plugin, this seemed like it would slow down iterations further compared to doing it natively. The Houdini engine feature to use .HDAs and spawn and use unreal’s foliage and not normal instanced meshes did not exist at the time. That feature got added in a Houdini Engine update about half-way into this project. At that time, however, enough time had already been spent on implementing and improving the use of Unreal material-based foliage spawning. The decision to keep that implementation was made.

**Road system**

With the procedural spline-based method of spawning roads and the underlying implementations, the feature to have roads cross and intersect is not possible. With another method, that could perhaps be possible, such as path-finding algorithms used in Ghost Recon Wildlands. That method, however, would remove the hand-painted mask possibility with the method that was used in this project.
3.9 Flowchart

(Fig 6 - flowchart showing the data flow and types of data in the tools)
4 Result

This is the resulting features/sub-tools and map variations I made to demonstrate this procedural workflow using simple masks as a basis. You draw several layers of features in a Photoshop document file. That file then gets used in Unreal Engine with the Houdini assets. With those, a real-time environment that is 1x1km in size gets generated. The layers from photoshop get used to designate the 4 different zones in the terrain; Grass, Forest, Farm, and Road. The terrain master material uses those to add different textures and blend the borders. These then get used to procedurally spawn foliage such as bushes and trees in different natural-looking hierarchical patterns.

4.1 Terrain

Even with the blurred cloud height-layer from the .psd file (fig 6.1), the resulting height-map was noisy when imported in Houdini(fig 6.2). A blur was applied(fig 6.3) along with low-frequent terrain noise to get a more interesting heightmap(fig 6.4).

(Fig 7 - Input heightmap > terrain > blur > noise filter )
This is the final terrain look from Houdini (figure 7) before sending it to the game engine. The splines from the roads were converted to meshes and projected on the terrain to flatten out the surfaces for later use.

4.2 Roads

With splines for the roads, a blueprint was created that takes a spline as input and then applies road detail decals (fig iv). The road also gets outputted as a landscape layer so that it does not spawn any foliage or grass on the road itself. The tool also scatters road signs, small rocks and of course the cars to the roads. The speed limit, the distance between decals, the number of speed signs and lamp posts can all be tweaked via exposed parameters to the level designer. By default, the length of the road is measured so that we get an even distance between stop signs among other things.
Here we see the final result of a road, with light poles, correct speed signs, road lines, animated cars and small tire wear details on the asphalt.

The use of decals made it easy to create road systems. With access to the main road spline, new ones could be made that gets offset sideways for further use. At certain distance intervals, points were sampled determined by distance parameters. With this, the position/rotation of the point could be extracted to be able to place a road line decal there. A distance threshold was set for each type of line. A horizontal offset was used with this to get the number of decals to generate. The points were then looped through, multiplying each point along the spline with the distance to get an even pattern. All of this was driven by different sets of grouped parameters so that the user could easily tweak the sizes and distances to get good results. With all the blueprint logic in one road script, the information can be used to have different lines and offsets depending on the speed limit on that particular road. This will also change the material on the speed signs to reflect that value. So from a single spline, enough information and a collection of scripts, an entire road system can be made working.
4.3 Cars

The cars are instanced blueprints that have a speed parameter, random color assignment, and spline. A copy of one of the road lane splines from the road gets sent to the car scripts to be used for animating them along the roads. By default, there is also a small chance that one of the cars becomes a police car instead, with blinking sirens. The car gets a start position on the spline and then follows the spline road until it gets despawned.

4.4 Farm

(Fig 11 - Farm area generated with wheat plants, a tractor, and fencing around)

Here we see one of the farm areas generated, it spawns semi-chaotic grids of wheat plants, the occasional tractor, and hay balls. Fencing around the border of the farm areas gets procedurally generated. The farm area also gets higher priority when deciding what feature goes on top. If a farm area were to overlap with a painted forest area, the farm area gets priority. It overrides the layering and there are no trees in a farm area. Power poles do get put on top of wheat plants though. Roads get the highest priority no matter what and will override both farm areas and forest/grass areas.
4.5 Input/Output

(Fig 12 - Input map used for all this generation)

(Fig 13 - Top-down look of the above map in the game engine)
With the simple input image (figure 12), the next image (figure 13) becomes the result in the game engine with only a few button presses and automated scripts and math. Below we see an image rendered closer to the ground to give a visual representation of the results that gets auto-generated.

(Fig 14 - close-up render of the generated map)

4.6 Animals

To give that little extra touch of life the decision to add animals was made. When the map gets generated, an option to spawn in crows and birds that fly around exists. They then get a random position from inside a box volume with a procedural straight-like flying path. The other option is creating random points in a circle-like shape and then creating a closed spline from that for infinite flying animations. Each crow gets a random speed value derived from a min-max value. This gets divided with the length of each spline to have a consistent flying speed. This multiplier is added to the animation speed so that the faster flying crows get faster flapping animation and vice-versa.
4.7 Editor scripts

With a plugin for Unreal Engine, the possibility of scripting and extending the editor gets possible. This further reduces the steps and clicks needed to generate all of the content made in this thesis. A simple custom UI window was created (Fig 15) with buttons and custom text that can be docked in the editor. This enables these scripts easily within the editor along with other features. Since the spline generation method relies on custom .HCSV files some way to import these to the project were needed. When the terrain gets generated, a file path parameter was exposed so that the .HCSV gets automatically exported from the Houdini plugin. This file gets exported to the import folder in the Unreal project. With an editor setting enabled, the ability to auto-import assets becomes possible. When a sub-folder in the import folder receives a new file, Unreal Engine detects it and imports it into a specific folder in the project. With sub-folder, you can choose which files get imported and where to automatically.
The first script in the UI allows the user to simply select the auto-imported .HCSV file in the asset list in the engine, press one button and have it automatically spawn road scripts for each found spline in the .HCSV file as described earlier. Here, there is also a custom button script for spawning crows.

(Fig 16 - Custom UI window)

(Fig 17 - Scripts for custom UI)
5 Discussion

Due to the many steps required to get a reasonable result and the intertwining mechanics, I feel that the visual fidelity is not high enough for an end result, but good enough for concepting a map/environment idea, as set out in the goal of this report. The terrain itself only got a blur and minor noise added from Houdini. This was due to the extra calculation time it took to add and simulate erosion from Houdini. I, therefore, decided not to use erosion to more quickly get the results in-engine, as blur and noise are relatively fast operations to perform. One could add the option to disable the built-in blur and instead use a higher-quality height-map for more realistic results.

5.1 Time-saving elements

There are several areas where this workflow saves a lot of time compared to the normal hand-made workflow. The biggest time-saving element from my reasoning is getting the actual terrain mesh/textures out to a game engine. Typically you would create your terrain in an external program, simulate erosion, bake out splat maps and then export it to the right resolution/ratio. With this workflow instead, these pipeline steps are reduced to only drawing out shapes in a .psd file and procedurally create the environment in only minutes. And this time gets reduced even further when you have everything set up. Making a change in the concept image including changing the entire concept gets you a result in the game-engine in under a minute. So a level designer/map artist would be able to test out dozens of variations or ideas in under an hour. Normally this could take hours for each map variation with the conventional workflow.

5.2 Similar work

The decently similar work I have been able to find is mostly for a specific game with specific features. Those were of course created over many months or years with a dozen technical artists or more. With that said I have barely scratched the surface of what is possible but my results turned out better than I anticipated. The only notable negative thing I encountered that I cannot get rid of is the seemingly slow implementation of Houdini Engine for Unreal. Due to this, iterations take a bit longer to do than I would have wanted. The other thing is the procedural foliage generation, Unreal does it in grid blocks. This went pretty fast in the beginning but as I added more rules and different kinds of foliage the generation time have skyrocketed away to several minutes sometimes.
5.3 Weaknesses

Due to the short time and focusing on core functions, certain features that one would expect are not implemented fully. This included the crossing of roads, the road generation would break if one were to draw overlapping roads in the .psd file.

Many of the steps I took were specific to the particular game engine and Houdini, but the concepts can be applied to other game engines. The Houdini part however I believe is hard to accomplish with other programs. Some parts could be replicated in basic ways in programs such as Autodesk Maya and other 3d programs, but I believe Houdini is a really extremely powerful tool and have a lot of unique features.

5.4 Lack of diversity

There’s no real diversity for several of the features due to time constraints. I originally planned to have different types of farm area growings such as potatoes and other plants. I ended up only using wheat plants instead. Ideally, one would develop a list so that you could select the plant for each farm area. But as this relies on landscape layers it is a bit more complicated to get that working out of the box. It would require more development time in Houdini.

5.5 The road problem

The road problem could be solved by adding support in my tool for multiple layers of roads, therefore allowing separate splines of roads that would overlap. This would, however, mess up the decal generation and road lines. A more advanced method could be developed for finding these intersecting areas and block the normal decals and/or add a special intersection mesh/road part.
6 Conclusions

From my result and discussions I conclude that while these types of tools can be made and are highly useful for creation of multiple environments with similar features, it is hard to make a generalized toolset that gets good results. It is easier to specialize the tools for a purpose like with this report, with a predefined list of features. With that said though, when you have the base features nailed down it takes less time to add new types of sub-tools and new purposes. If you get solid core-pillars of tools and scripts it is easier to create new features that derive from them.
7  References


