Procedural Generation of 2D Terrain Environments with Watercolor Rendering

Sean Sullivan
sean.sullivan@yale.edu

May 4, 2017
This project combines the effects of a number of component implementations to create a game-like 2D terrain scene. The vision for these scenes incorporates a watercolor effect placed over cross-sections of a 3D terrain, such that unique backgrounds and foregrounds of the scene can be realistically generated as the character moves through the 2D space. The watercolor effect is achieved with a custom-written shader material applied in Unity onto the necessary terrain. 3D terrain, including base color scheme, is generated using an agent-based simulation.

The result of this project shows a number of successful elements, generating unique 2D scenes with an aesthetic matching the intended design. In particular, the project achieved the goal of generating almost all elements required to compose the game environment from scratch. The addition of platform-oriented gameplay elements and movement between scenes in future work can successfully incorporate the achieved aesthetic into a fully functional game.
## Contents

1 Introduction ................................................. 3

2 Shader ...................................................... 3
   2.1 Watercolor Effect ........................................ 3
   2.2 Texture Generation ....................................... 5

3 Terrain Generation ........................................... 6
   3.1 Agent-Based 3D Terrain .................................... 7
   3.2 Billboards .................................................. 9
   3.3 Color ..................................................... 12

4 Externals ...................................................... 12
   4.1 Character .................................................. 12
   4.2 Unity Assets .............................................. 13

5 Conclusion ................................................... 13

6 Acknowledgements ............................................ 13

7 References ................................................... 14
1 Introduction

This project combines the effects of a number of component implementations to create a game-like 2D terrain scene. The primary source of inspiration for this project was the work of artist Chiura Obata in Yosemite National Park. Shown in Figure 1 is an example of his work, which was often painted on silk and frequently used watercolor. The vision for the result of this project was to apply a simple watercolor aesthetic to a game environment simulating mountain scenes, using as few external assets as possible.

The project contains five coordinated primary tasks. The first of these is a fragment and vertex shader that produces the desired watercolor effect and outlines terrain appropriately. The second task creates the necessary noise textures to be used by the shader. The third task generates a 3D terrain environment with features to match the aesthetic vision. The fourth instantiates this 3D terrain into Unity as 2D “billboards” at runtime. Finally, the scene is attached to the shader with appropriate base colors for each feature.

Each of these tasks resides within one of two categories, either aesthetic work on the shader or procedural work on the terrain. As such, this paper will approach the methods, successes, and failures of each category in turn. Also important for this project was the learning element of these tasks, so in many cases the implementation was specifically chosen for educational reasons; these will be discussed in a case-by-case manner.

2 Shader

2.1 Watercolor Effect

The watercolor effect was achieved by implementing an algorithm discussed in Bousseau et al. (2005). The paper applies three layers of color modification to mimic the watercolor effect, implementing each using a noise texture with some scalar factor $d$. Each of these layers is
evaluated via a color modification equation on each color component $C = [0, 1]$:

\[ C' = C - (C - C^2) (1 - C') \]

The first of these layers represents the turbulent flow creating the “watery” effect of the image. As such, it requires a smooth texture, so Bosseau recommends Perlin noise. Whereas that implementation uses Perlin’s original implementation, I have adapted from more recent work, as discussed below. The second of these layers mimics pigment dispersion effects, so a texture of Gaussian-distributed noise was used. Finally, a third layer of a picture of paper is applied to give the appearance of a canvas of some sort.

The shader was implemented in HLSL using Unity’s ShaderLab. It uses two passes, the first drawing the model itself with the watercolor effect and then re-drawing the outline of the object with some offset. the first pass achieves the watercolor effect using a fragment shader, whereas the second pass extends the scale of each vertex relative to the center of the object and colors it according to an outline color parameter, which was always set to black. The shader was applied
to a new material at runtime using a random set of textures for the first two layers and one paper sample for the third. Also present in the shader is an input for the base color of the material, which is applied to the material of each terrain object at runtime.

While the shader achieved many of the desired effects, there are definite improvements to be made. The three layers of color modification achieve a unique aesthetic that fits the vision. However, future work would include further improvements by attempting to procedurally create canvas-like textures, rather than using a paper sample. Further, it would be better to use the sum of some Gaussian noise samples taken at different scales for the pigment dispersion effect, to smooth out some of the artifacts that show with increasing $d$ values. Bosseau also includes an edge-darkening effect using the gradient of an abstracted image, which was not included in this design; further work would include adding in edge-darkening effects and decreasing outline width, for a more vivid painting aesthetic.

2.2 Texture Generation

Three texture samples are required as inputs to the watercolor shader—a Perlin noise sample, a Gaussian-distributed noise sample, and a “paper” sample, as listed above. Whereas the third sample requires a more photographic input, the first two types of textures have distinct ways of being generated. The Box-Muller transformation maps uniformly distributed random variables to random variables with a Gaussian distribution. In particular, the algorithmic representation of the polar form of this transformation was used to convert random variables in the range $[0,1)$ into Gaussian-distributed random variables with mean 0 and standard deviation 1. This was scaled accordingly for export. Two dimensions of Perlin noise were generated using a simplified version of Perlin’s Improved Noise, which at each point performs a weighted average on the dot products between a random gradient vector and the distance vectors to nearby grid points. This creates a smooth, periodic 2D noise function $N(x, y)$ which was then sampled at
different frequencies to generate the final texture.

Since these textures are randomly generated, there was an initial hope to include them in the runtime environment so that each game instance had a unique set of textures. The original implementation in C# was attached to a Unity GameObject to run at game start, but this proved to be extremely slow. As a result, the work was converted into python scripts which output a number of textures that could be imported into Unity and selected from randomly. This was done as opposed to moving some of the calculation onto the shader itself in spirit of these performance concerns. The scripts, named `perlin.py` and `gauss.py`, can be found in the project folder and run using

```
$ python perlin.py
$ python gauss.py
```

which will create 19 of each texture in an existing “/noise” directory.

## 3 Terrain Generation

For the terrain generation, I wanted to invest in the idea of a full 3D environment, like that of Yosemite, so that each “painting” (scene) in a single game instance could exist within the
same, consistent world. Initially there were competing ideas for 2D vs. 3D presentation, but limitations in Unity on the types of shaders that can be applied to 3D “Terrain” objects guided the project in the current direction. This includes the generation of a full 3D environment to fit the desired design qualities, as well as down-sampling this environment by cross-section to create 2D billboards.

3.1 Agent-Based 3D Terrain

With the emphasis in Obata’s paintings on distinct landmarks such as specific mountains or waterfalls, it seemed appropriate to enforce the creation of landmark features uniquely, rather than with alternative options like noise textures. Thus, this algorithm followed the design of Parberry (2010), which enables software agents to simultaneously affect areas of the landscape, thus generating a full terrain. Software agents are algorithmic constructs which spend allocated tokens on terrain operations. The process includes two relevant stages: coastline generation and feature generation. First, the coastline is generated by simultaneous agents that uniquely identify attractors and repulsors, and then push areas of the current coast out based on this. To do so, new areas added to the coast are chosen based on those proximity to those points as well as the edge of the map. Next, features are generated randomly throughout the allocated island area. These features include beaches, mountains, hills, and rivers.\(^1\)

Of paramount concern are the coastline agents, which are supposed by design to run simultaneously. Thus, this seemed to be a feasible exercise in process forking and memory locking. As a result, I implemented this task in a C project which shares the terrain memory and adapts the coastline using multiple different processes (agents). Although the coastline could be generated in this way, the feature generation was not. Features were instead applied to the already-raised

\(^1\)While Parberry’s paper includes algorithms in pseudo-code, it leaves out details on how certain functions (wedge raising, beach flattening, etc.) operate. Source code for Parberry’s implementation was not available during the course of this project, so the implementation includes personal solutions to some of these missing elements.
coastline afterwards, randomly selecting either a feature-based agent or a smoothing agent with some probabilities.

This portion of the project succeeded in creating an interesting coastline and some basic terrain shapes but failed to realize fully lifelike 3D terrain. This is in part due to difficulty in implementing functions such as wedge-raising for mountains and hills, which creates staggered-looking changes in height. The beach agent seems to work acceptably well, but parameters to allow it to function properly and create lifelike beaches also did not occur. The river agent works quite well, but it sometimes creates unrealistic paths (e.g., up mountains). Although these feature issues are significant, they do not detract from how well the project achieves its vision, since unique and interesting shapes resulted in any case. Further, the effects are not always immediately problematic in the 2D view since many of the odd-looking shapes of the terrain are due to abnormal gradients in 3D, which the 2D environment naturally does not show.

In the future, I would aim to sharpen up most of the problems I have previously listed, as well as extending the simultaneous nature of these agents to the feature-specific agents.

Files for this program are found in the terrain subdirectory. Instructions for compilation and running the program are in the README.md file in this subdirectory.
To create the 2D painting environment, the 3D terrain must be dissected into flat planes that can be presented into the scene. To do this, a flat plane mesh with 101 vertices along the top was generated in Maya and imported into the Unity scene. Then, the Unity scene has a GameManager object which an attached script set to generate a number of these flat planes (the Land object) at runtime. Next, a single, 101-unit-long cross-section of the 3D terrain is selected, and the mesh of the flat planes is updated such that the top vertices of the plane match the shape of the cross-section. Appropriate materials and shader interactions occur from here. Billboards were all placed in the same initial location stretching across at the bottom of the screen.

While the results of this implementation are encouraging, there are some undesirable elements remaining. For example, there exists the unfortunate result of mesh objects being shared from import and not easily instantiated anew, meaning that features created on earlier billboards also show on later-created ones. Also, while the mesh was updated to fit the terrain, the collider component of the Land object does not match this, so any character manipulations that occur in the scene occur on the same flat plane. Problems also arise when considering the random nature of selecting cross-sections from the environment. Although the algorithm checks to start each billboard at some feature, it is possible to use smarter design to make features more identifiable pieces of the scene, for example by placing features in each third of the image (according to the rule of thirds). The last and possibly most important issue that arises with this presentation of the billboards is that because UV coordinates are not modified when vertices on the billboard meshes are changed, the watercolor effect appears stretched for taller features. Future implementations would look to scale each texture coordinate along the height axis to match the increase in vertex height.
Figure 5: Sample game environment screen capture with lighter background option and three billboards.
Figure 6: Sample game environment screen capture showing significant texture stretching.
3.3 Color

Because each billboard needs a different identifying base color representative of the type of terrain being visualized, the 3D terrain generation was supplemented with identifiers for each agent, which then tags each location of the terrain as it reaches them. In this way, the code creates a texture whose color information corresponds to a hash of each type of agent. Each billboard identifies the type of terrain at the first point in its path and randomly chooses a color from a range of colors designed for that terrain type.

4 Externals

4.1 Character

The player-character included in the images and the Unity scene was created in Maya for a separate project. He is included in the game environment as a prototype to show off the effects of the watercolor effects on a character. Due to some differences in mesh details, the shader had to be duplicated, with some changes so that the outline would display properly. Although animations are not yet included, he has a placeholder movement script which allows him to move left and right across the flat plane off of which the terrain is lifted. While I am pleased with the watercolor effect on the character, I am interested in both increasing the quality of the

Figure 7: The most recent agent touching each pixel of the terrain (brightened for visualization).
outline so that he is more easily visible in-game and providing a base wood texture as opposed
to either the main color parameter or the paper variations parameter to empower his toy-like
design.

4.2 Unity Assets

A few other assets are necessary to comprise the Unity scene. One of these is the Background
prefab and attached script, which simply assigns the watercolor shader to a larger background
image. There are two small ranges of color which the background can take on, representing
day and night with lighter and darker shades. The GameManager object also plays a significant
role since it controls generation of the Land objects. Prefabs can be found in the Unity project
folder under Assets/Prefabs and Assets/Scripts.

To build the full Unity project, in the source code, unzip the archives in the “unity” subdi-
rectory and merge the two “resources” folders. Then, replace the Assets folder in your Unity
project file with this folder, open the scene file and build.

5 Conclusion

Overall, the project successfully demonstrated a number of uniquely implemented strategies.
The visual presentation of the ultimate game environment was satisfactory—although improve-
ments can be made, it is clear what those improvements must be, and the current visual still
achieves the desired style.

6 Acknowledgements

I would like to thank Holly Rushmeier and the Yale Computer Science Department for allowing
me to pursue this inspiration as my final project. I would also like to offer my thanks to Lining
Wang and Sam Levatich, who offered helpful peer insights over the course of the semester.
7 References


Image Reference