Pixel Quilting for Random Textures

Berke Boz, Alec Grover

Overview

Pixel Art has a significant presence in game development for a variety of reasons, but one of the most common is its accessibility for programmers to create game art without an art background. By using so-called tilemaps, developers can array grids of small self-repeating textures to create large fields of arbitrary size with only a relatively small amount of time dedicated to asset creation. Tilemaps are fantastic tools, but they struggle when used to represent inherently stochastic textures, such as foliage or chaotic patterns. While in small scale the repetition is not extremely obvious, in a large space the human brain's exceptional ability to identify patterns can cause a ghostly grid to appear. Obviously, this effect is not desirable, but to create entirely original textures is almost always prohibitively expensive.

Our goal was to turn to image generation techniques in the field of computational image manipulation to create a system that can take a single tile and produce an arbitrarily sized texture field that does not contain the grid aliasing caused by tilemapping. The challenges with this lie in two primary areas; extraction of enough details from a tiny input to produce quality results and enough efficiency to dynamically produce textures in real time. These challenges meant that late in development we were forced to move away from our target floor of 32x32 input tiles and move to a 64x64 input. While we had some success in generating convincing textures with even the tiny 32x32 samples, the number of patches that needed to be stitched became problematically computationally complex. With a 64x64 input, this problem is eased, as the patch size can be large enough to substantially reduce time complexity, while also remaining small enough to avoid significant structural repetition.

Once we had an effective synthesis system, our focus turned to creating a functional demo of such a system running inside a game environment in real time. Originally, we had targeted a Unity based demo, but the amount of overhead required to operate our system inside Unity proved to be a steep enough challenge to justify an entire project of its own. Instead, we used the PyGame library for Python to produce our simple game environment at a very low level by effectively writing directly to the camera. With this in place we then needed to tweak the image quilting algorithm to produce larger sections of the terrain in an order that generates playable space near the player first. In both of our existing work from the first assignment we had generated the left column and top row, then iterated row by row to determine and integrate each patch, however, this meant that the player could easily outrun the generation by travelling perpendicular to the direction of generation. To fix this we altered the generation order to produce patches diagonally such that the terrain is generated at an equal rate on both primary axes to the player. This ultimately boosted both performance and appearance dramatically, and beyond the actual goal of the project it is one of the most interesting theoretical result in isolation we produced.

Experimentally, we tried using a variety of different inputs, and ultimately found that it works best with a 64x64 texture. The one used for the demo uses a patch size of 20 and an overlap region of 2, but those numbers will change based on the nature of the input image. In an ironic twist, by creating

a program to improve upon tilemaps for random textures we have broken the part that tilemapping does very well, and our results on highly structured tiles have been less than desirable. While it seems that the image quilting algorithm can be extended quite well to noisy inputs with small resolution, it does lose a great deal of its efficacy with the small patch size on grid patterns. We believe that it may be since a single pixel of offset on a tile where the seams are a single pixel wide is far more obvious than on a higher resolution input where a solitary pixel error is not as immediately identifiable. To an extent, a very carefully selected patch size should work, however, this is only the case for perfectly regular patterns. One such test we performed used a wooden floorboards tile that had alternating board widths between rows. With our patch sized the algorithm was unable to learn this, however, it still did produce a structure that came close to being orderly.

The Future and Existing Errors

In the future, to extend this project beyond theoretical there are a few key areas in which it needs to be improved. The first and foremost is in improving how mapped tiles are saved. Currently, each tile saves its basis texture as a pickle, and upon instantiation when it is loaded it first queries the patches folder to check for a pre-generated version, using that one if available. The better system would be to save all the basis textures as a dictionary, pickle that, then create a dictionary that maps each generated index to the key to its basis texture in the dictionary. This would significantly reduce the amount of complexity dedicated to file IO and should provide a sizable performance boost. In the current implementation there is a bug with the multithreaded design that causes tiles to occasionally be skipped when adding them to final display set. We mitigated this skipping in the presentation by harshly slowing the camera to avoid generating too far ahead of the rendering portion, but this would need to be addressed in the long run. Additionally, some inverse of existing logic would allow the program to generate to the left and right of the starting position. This portion is already fully capable of being added, however we elected to focus on getting the existing two directions of travel correct for the demo before branching to all directions.

Division of Labor

Due to parallels to Assignment 1, the base code for image quilting was taken from Berke's implementation as he was further along at the time of starting. Berke adapted his code into a PyGame environment and extended his generation system to allow for expansion. Berke also implemented the multithreading used to allow the game logic and texture generation to run in parallel.

Alec had originally been working on the Unity adaptation, however that route ended up being scrapped. Alec's work on the current implementation included integrating the diagonal logic, setting up texture serialization and deserialization, and setting up the final demo. Alec also worked on developing various tile-able textures for use in testing and demonstration of the system for our Pixel Art use case. Finally, Alec produced the papers and video for the submission and proposal.

Resources

Python Libraries

- NumPy
- PyGame
- Pickle
- OpenCV Python

Textures and Art

- Custom textures were made in Pyxel and Aseprite
- Furniture artwork is from Pixel Art Medieval Interiors by Acasas on GameDevMarket.net, Alec owns a license to use the assets for both commercial and non-commercial products.