CS399I Project Results

Procedural Texture Generation

Adam De Broeck

January 3, 2024

Abstract

This abstract serves as a restatement of the original objective outlined in the previous research paper — that said, this will be a short summation followed by changes to the original design of the project.

This project has the intent of providing a pipeline friendly method of designing and implementing procedure textures into a modern game development environment. To do so, a couple of techniques were developed and/or adopted to better suit procedural texture generation — permutation based generation, a technique created as a result of this project, texture synthesis, a technique adopted to work alongside permutation based generation, and texture transfer, another adopted technique used to give users better control over texture synthesis. The formula for creating permutation based textures and by extension their file formatting/loading was inspired directly from Farbrausch's Werkkzeug program and their game .kkrieger. By using instruction based generation methods, users can have direct control over which parts of the texture are randomly generated (within constraints) while having the freedom and flexibility to make nearly any texture imaginable.

An example tool workflow, and a demo was created to show how these techniques could work in a practical setting. Naturally this tool is experimental and disregards performance and QoL features for sake of simplicity. Textures are entirely deterministic and can be randomized entirely by choice of the implementing program. Additionally, the tool created to work on PCG textures also serves as a commandline utility which can render the PCG textures on the fly.

Definitions of a Procedural Texture

In order to understand what techniques might be able to be utilized while creating procedural textures, it's important to understand the goal of a procedural texture and how to define one. The book *Texturing and Modeling: A Procedural Approach*[1] describes the advantages of a procedural texture in its second chapter. While not a direct definition, the descriptions here fit the purposes of what one might seek from such a texture. The book describes a procedural texture as being "extremely compact" in file size, which makes sense given that such a texture would be generated from a series of parameters in an algorithm rather than stored in a raster. Additionally, it mentions that a procedural texture has "no fixed resolution" and "covers no fixed area". What this means is that a procedural texture is in essence infinitely scalable and infinitely tileable with guaranteed resolution as an output image. This is similar in concept to the vector graphics format where the image is stored as shape and vector data so that it can be re-sized with infinite precision. Finally, a procedural texture needs to be able to "be parameterize" which allows its author to control the output of the algorithm via some variable input.

This all sounds good in theory, but in practice the "no fixed resolution" and "covers no fixed area" properties are not usually that useful in the game development setting. This is because in order for such properties to be useful, they need to be generated on the fly, and that doesn't mean just runtime startup. Any property that allows infinite scalability and scrollability loses those properties as soon as the texture is rasterized, thus in order to make use of the property the texture would have to be continuously generated per frame. Due to the complexity of such algorithms this is typically not a feasible feat to strive for — even if it is possible.

It should be noted that this is a bit disingenuous as even the book makes mention of such textures being finalized in raster form. However, a better definition would be to define a specific bounds that is desired for said texture. That being said, the final "definition" that was used for this project was, "extremely compact", "fixed, predetermined bounds" and "parameterized".

Instruction Based Assets: Farbrausch's Werkkzeug

A big inspiration for the design of the procedurally generated textures of this project comes from the group Farbrausch. In a 96k game competition (challenge to fit a game in under 96 kilobytes) back in 2004, Farbrausch submitted the entry *.kkrieger* which was a first-person shooter game that expanded out to over a couple of hundred MB at runtime[2]. One of the ways this was possible was due to the game's instruction based texture generation at runtime, where the step-by-step instructions needed to generate an asset were saved instead of traditional asset data like pixels.

Farbrausch showed this off in a presentation through the use of their tool, Werkkzeug, which they used to create instruction based textures, models and even animations[3]. By using instruction based texture generation, users can define specific steps in which a texture is generated and then apply randomized constraint values to specific instructions to vary the output. The basis of this idea is hierarchical in nature and easy to understand, which aids in user designed procedural textures. This idea would form the basis of permutation based texture generation later in the project.

Method Implementations

In order to create a tool which could support permutation based texture generation, two frameworks would be needed. The first of these frameworks was the raster framework, which needed to support operations being performed on it through the use of `raster functions`. This would allow all operations on a raster to be derived from a singular abstract raster class which could define an inheritable lambda used for applying said operations to the raster. Every function in the procedural generation tool works on a raster through a specified raster function.

An example of a raster function looks like the following (written in `C++`):

```
void FillRasterFunction::RasterFunction(SelectableRaster *raster, const AbstractRasterFunction *instance)
{
    // Get this instruction's parameters
    Util::Color topColor = Util::Color(Util::DecodeUnsignedChar(instance->calculatePropertyVal("Color_Red")),
                                       Util::DecodeUnsignedChar(instance->calculatePropertyVal("Color_Green")),
                                       Util::DecodeUnsignedChar(instance->calculatePropertyVal("Color_Blue")),
                                       Util::DecodeUnsignedChar(instance->calculatePropertyVal("Color_Alpha")));

    // Apply this function using parameters to the raster
    auto selection = raster->getSelectedRasterIndices();
    for(int pixelIndex : selection)
    {
        Util::Color bottomColor = raster->getColorAtIndex(pixelIndex);
        bottomColor = Util::Color::BlendColors(topColor, bottomColor);
        raster->setColorAtIndex(pixelIndex, bottomColor);
    }
}
```

The second requirement of the toolset (also seen in the above raster function example) is the ability to wrap primitive data types in a wrapper that has knowledge of the primitive stored in it — a sort of pseudo-reflection. The reason for this was to be able to easily store, modify and add primitives of different types at runtime through a single generic function for each raster function instead of using specialized functions. This helped with both the cleanliness of the codebase but also in saving of values to disk later on in the project.

A number of basic operations which were specified in the prior research paper were implemented to allow for users to create any sort of image with relative ease. Every raster function implicitly supports randomized constraints on their property values due to the inherited property values in the abstract base class of each function. Because of this, any and all properties are eligible for permutation based texture generation. The basic operations implemented are raster selections (clear, rectangle and polygon/triangle), crystallizations (Voronoi cell generation), blurs (motion, Gaussian and fragmentation), gradients, HSL modification filters, shapes (line, rectangle, circle and triangle), fills and PRNG seeding functions.

Some operations that were mentioned in the research paper that were **not** added due to development time (and as an exercise to the reader) were the additional "full" generation functions like noise, Perlin noise, cellular automata and more. These types of functions would be useful and can be easily implemented due to the nature of the abstract raster function class (the class is even designed to automatically provide and register a static base class on runtime start without user intervention). With all the above frameworks and methods, permutation based generation was effectively complete. Below is an example of a permutation based randomly generated texture:

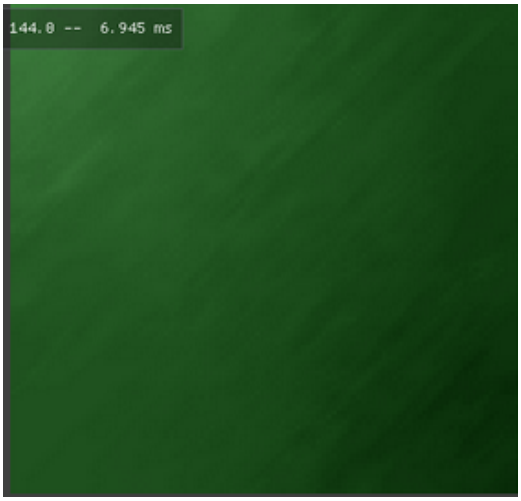

Figure 1: Permutation based randomly generated texture

The original research paper also mentioned a second method of random generation — property based generation, the idea of which was to use instruction macros to repeat and randomly apply sets of known procedures to a texture during generation. This idea was scrapped for a couple of reasons, the first of which was originally a time constraint. Developing a macro framework would take far too much time away from the rest of the project, and it also became quickly apparent that permutation based generation by itself was extremely tedious to get good results — something which property based generation would not fix, as it is effectively just shorthand permutation based generation.

To solve this problem (on both fronts) texture synthesis was added instead. The core idea of texture synthesis is to take known good parts of a donor image and use those parts to reconstruct a larger image. This works by searching the donor image for overlapping regions of the new image for a matching (or close to matching) image. By design, this allows the newly placed "quilt" piece to be a fairly good match with the already placed pieces. This method was described in a follow-up paper to a presentation at the 2001 SIGGRAPH written by Alexei Efros and William Freeman titled *Image Quilting for Texture Synthesis and Transfer*[4]. Below is an example of the same texture as before but using texture synthesis instead:

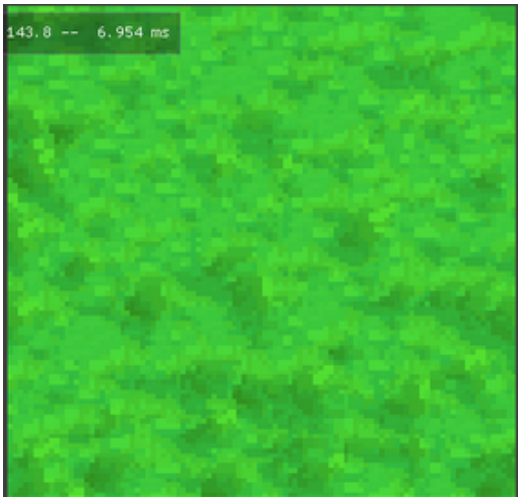

Figure 2: Texture synthesis based randomly generated texture

This technique, while usable by itself for satisfying results, works incredibly well with the aforementioned permutation based texture generation. Part of the reason for this is because of the latter's ability to create "features" as well as both stochastic and structured image data in a single pass (multiple instructions). This data works extremely well when fed into texture synthesis — as such, it's not only easier to implement than property based generation, it's also more practical. In principle, the idea of texture synthesis works very similar to wave function collapse in that it uses historical data in order to accurately generate new data. Unlike wave function collapse however, texture synthesis works on a pixel level instead of only in blocks. It is because of this, that the search must be done in realtime upon request instead of using a lookup n-gram table, as such a table would take far too much memory to be viable.

It should be noted that texture synthesis is only one half of the equation when considering image quilting. The other and arguably more useful technique outlined in this paper is image transfer. Image transfer works by also considering the data that is underneath the synthetic texture upon generation. By additionally matching a correspondence map, texture synthesis can effectively "re-map" a texture onto another texture. This is especially useful for "fully generative" image generation algorithms like Perlin noise which cannot feasibly take input image data and typically only overwrite existing data. By using texture transfer, any texture can be applied to any other texture, further increasing possible results for procedural texture

generation. Both of these methods, while inspired from the SIGGRAPH conference, were custom written for this project. Below is an example of the texture again, but using a correspondence map and applying texture transfer to it:

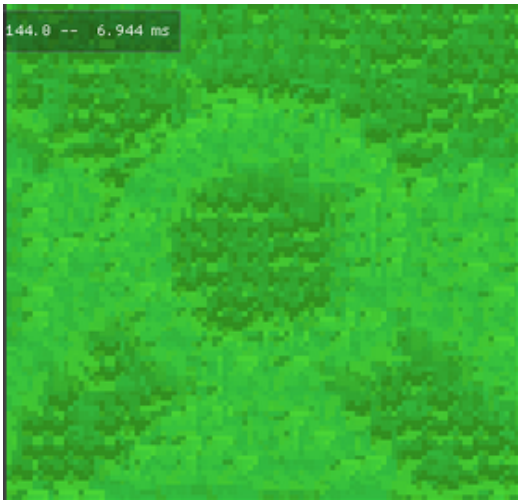

Figure 3: Texture transfer based randomly generated texture with an X and O correspondence map

Tool Breakdown

The following is documentation on how the procedural texture generation tool works. The tool is split up into three panels, and a menu bar. The left panel is for instructions for generating the texture. Instructions are generated in order of top to bottom. They can be selected by clicking on them, deleted by pressing the "delete" button of the selected instruction, and re-arranged by dragging the selected instruction to a new location.

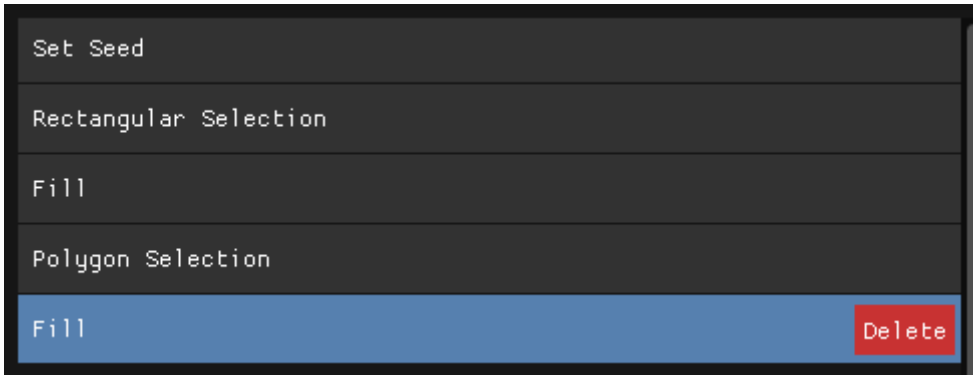

Figure 4: Selecting an instruction

New instructions can be added by using the dropdown below all active instructions, selecting the instruction to be added, and then clicking the "Add Instruction" button.

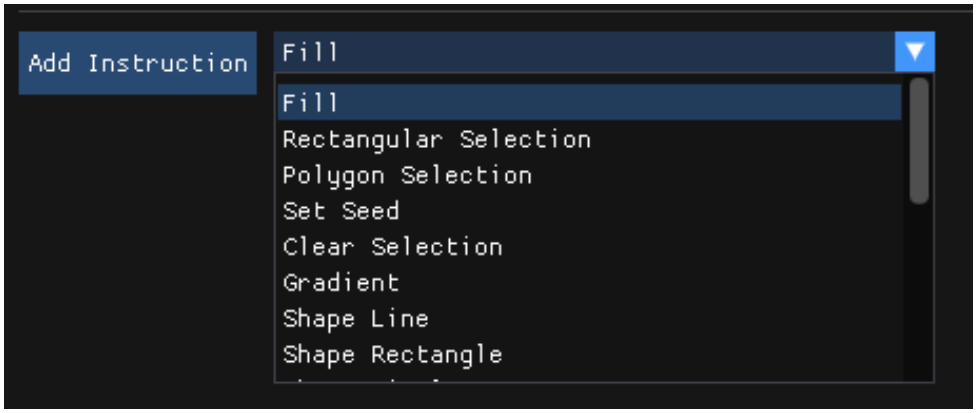

Figure 5: Adding an instruction

The panel to the upper right shows the currently generated texture. Textures that are to be generated are evaluated on a separate thread and will load in automatically after any change has been made. The panel to the lower right is the details panel and shows all the possible editable properties of a currently selected instruction. By checking the box that comes before the property's name, a randomly constrained offset can be added to the value. The min offset value describes how low the random offset will be applied to the current value, and the max offset value describes how high offset value describes how high the random offset will be applied. Random

offsets cannot generate offsets that would exceed a property's minimum or maximum values if a property has those values.

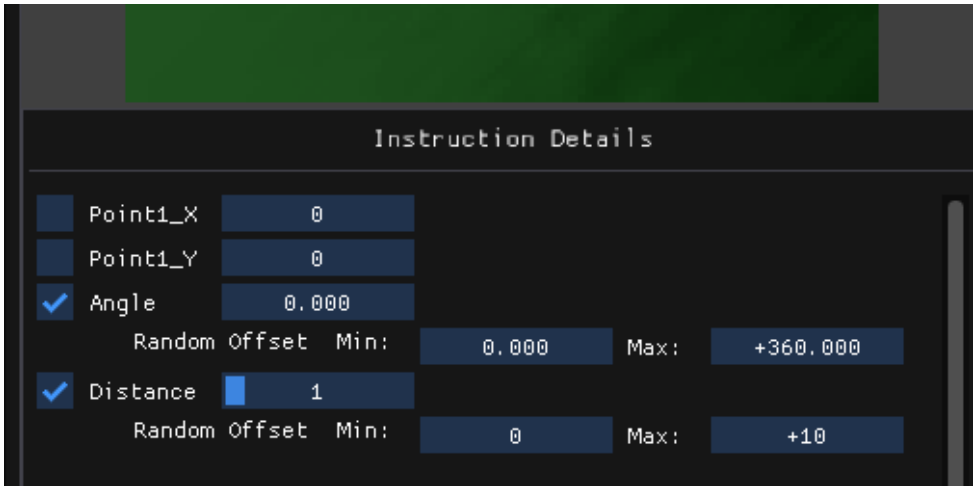

Figure 6: The details panel and how to set a random value for a property

Finally, the menu bar at the top offers the ability to create a new texture, open a texture, or save a texture.

Files generated by the tool are saved in a binary serialized format that specifies a header with information about the texture up to a maximum of 260 bytes. The instructions are then serialized to the file in order, and their properties are added by name along with their values and random constraints. The tool also serves a dual-purpose by being usable from the command line via program arguments. Doing so will generate a single passed-in texture. The format for these command line arguments are as follows:

```
GAM399I.exe [path_to_file] [seed1] [seed2] ...
```

Demo Results

The demo, ProceduralDungeons, is a past semester project in which a level is procedurally generated upon load in. Due to the nature of this game, it served as the perfect candidate to test the procedurally generated textures on. In order to incorporate the PCG textures into the game a function was built to load multiple random variations of all the textures that would be needed at runtime. This function created a process call with the correct arguments (the .pcg file to be loaded along with a list of seed values) which then loaded the resulting .png when the procedural generation tool was finished rendering the texture. Since the demo already featured a biome system for different parts of the dungeon, each biome was given a procedural texture to use on generation.

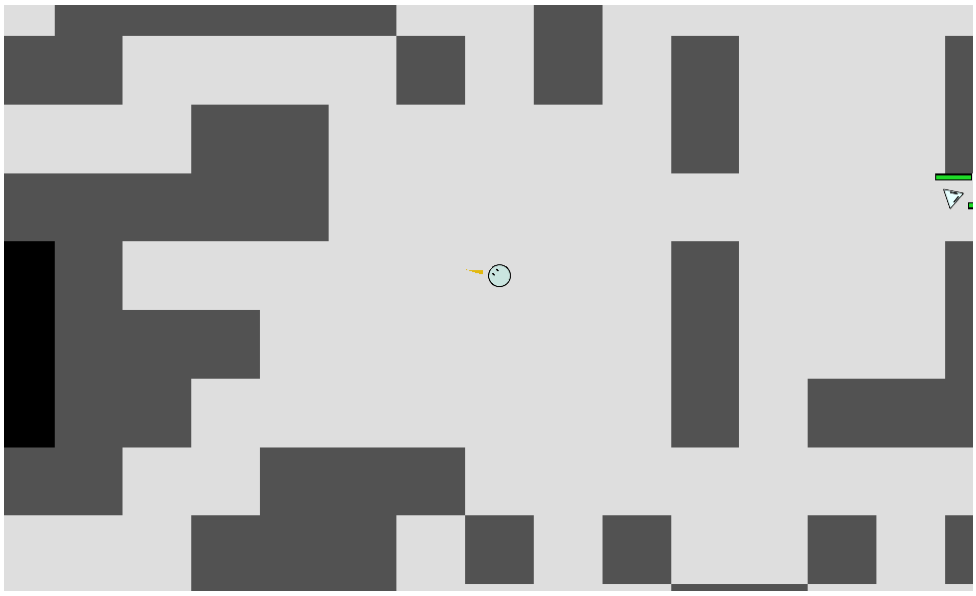

Figure 7: The original Procedural Dungeons game

The results looked remarkably well as the textures added new detail to the previously flat terrain. Each biome was also given a distinct feel and biome transitions, while gradual, felt more impactful and noticeable.

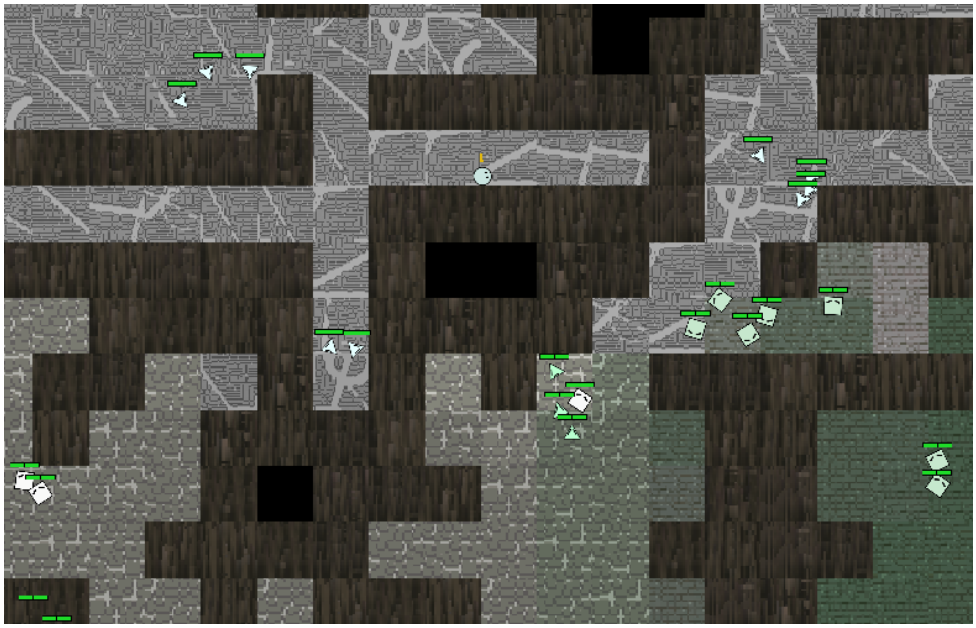

Figure 8: The Procedural Dungeons game with procedural textures

Where the procedural textures needed work was the connections between random textures. Due to the edges of each texture not having a transition, the biomes had a noticeable cutoff and look very "grid-like". Fortunately, this could be fixed with the same technique that made the textures in the first place — texture synthesis. Another problem was the design of the walls and the color values applied to the textures via the biomes; both felt a little flat and out of place.

It should be noted that no artists worked on any of these textures and, in experienced hands, even better results could be achieved. Some addition textures that came out very well while experimenting with the generation are below:

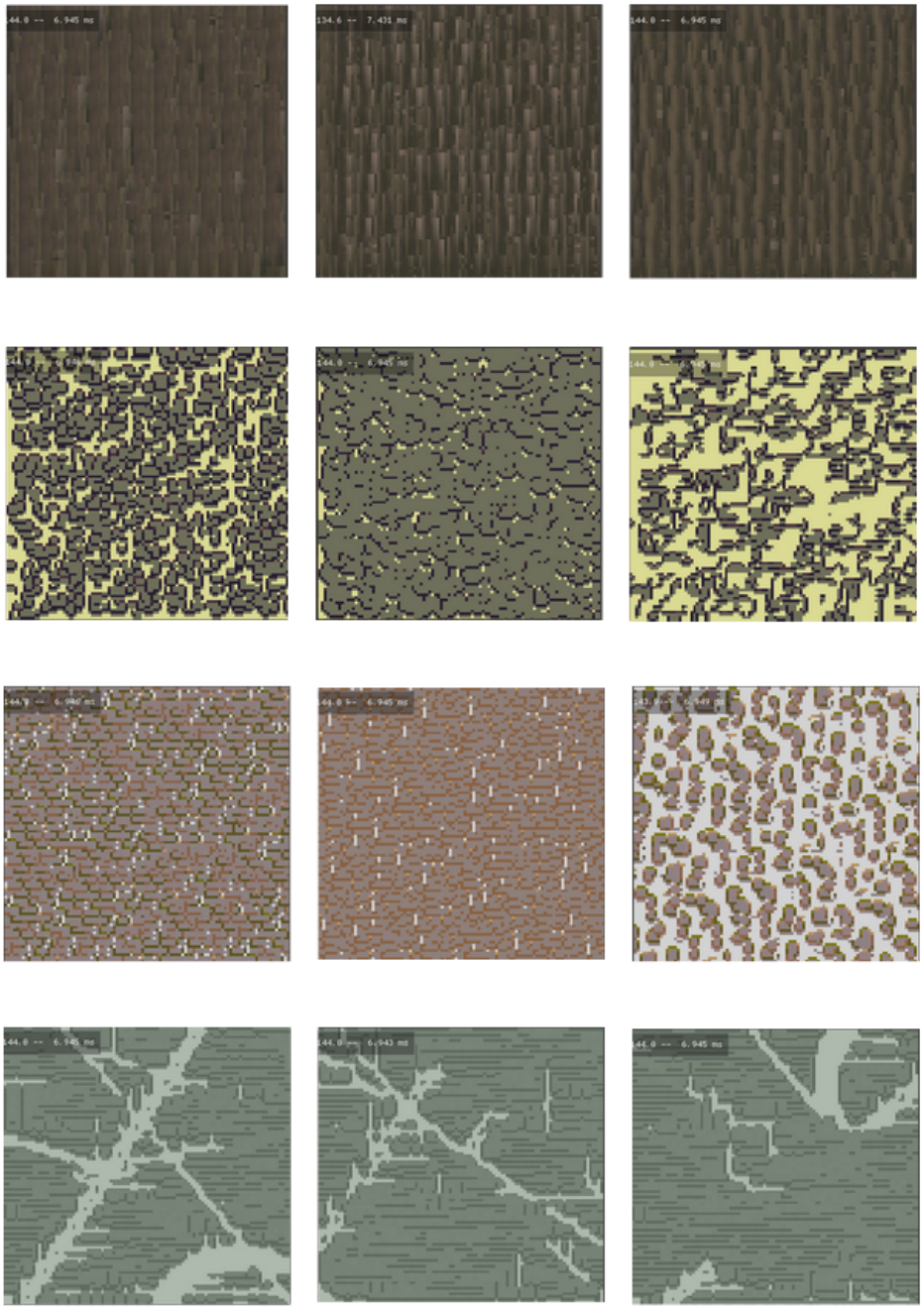

Figure 9: Additional procedural texture examples

Further Extensions

There have been a couple of further improvements already mentioned in previous paragraphs such as the use of more "fully generative" texture generation functions, as well as the use of more experienced artists working with the tools. On the topic of generative methods, the survey titled *Survey of Procedural Methods for Two-Dimensional Texture Generation* put together by a group of computer science departments from various universities lists a number of additional generative methods worth trying. Among those are techniques such as cellular automata (previously mentioned), reaction-diffuse algorithms, wavelet noise functions and even physical based simulations[5]. If combined with the already implemented texture transfer technique, a wide variety of texture patterns/feature would become available to this type of texture generation.

In the tool demonstration portion of this paper, it was mentioned that an alternative use-case of the tool existed via the command line for use in programs to load procedural textures. While this works, it's a single threaded and rather limited approach to loading textures. Developing a runtime environment for every single language and platform is far to much work however, so the properties of this being achieved in an existing program are rather desirable. A possible solution to this is to use memory mapped files to share data between the loading program and the client. This would allow for not only a parallel approach to loading procedural textures but also allow for more options and parameters to be passed to the loader program from the client.

A big pitfall that this implementation runs into is optimization. One of the biggest sources of slow down in the project's implementation is the brute force attempt at searching for matching images in the texture synthesis function. By using k-means clustering and a more optimized search algorithm, this process could be sped up dramatically as well as decreasing its operational complexity from $O(N^2)$ to $O(N)$. Multithreading most of the functions including the texture synthesis function would also dramatically improve the techniques' performance. Additionally, by moving the raster functions to shader code the entire program could take advantage of GPU hardware acceleration which excels at this type of operation which could increase the generation speed by several orders of magnitude. This would also make the program easier to be cross-platform as the functions themselves would not need to be re-written.

Another big problem with the tool itself is the UI/UX design of the tool. Currently, it's very difficult to use efficiently due to a bad layout, lack of

undo/redo functionality and other hiccups that get in the way during use (layout design leaves much to be desired as many of the UI elements don't afford the user with any information about how to use the program).

Possibly the biggest take-away from this entire project however, is the possibility of reversing the entire process. Instead of generating images from instructions, what if the instructions could be generated from an image? In this case, any passed in image could be drastically reduced in file size as it's broken down into its core drawing components. Imagine taking an 80GB game with 70GB in assets and shrinking it down to 5% of its original size. It doesn't stop there however, once an image is broken down into texture instructions it could be "riffed on" and permuted to a similar texture just by randomizing some instruction values.

There's already a technology that excels at this type of weighted constraint problem-solving — machine learning. It just needs someone to apply the concept and try it out.

References

[1] David S. Ebert, Kenton F. Musgrave, Darwin Peachey, Ken Perlin, and Steve Worley. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 3rd edition, 2002. ISBN 1558608486.

[2] Nostalgia Nerd. kkrieger: Making an impossible fps — nostalgia nerd, 2021. URL https://www.youtube.com/watch?v=bD1wWY1YD-M.

[3] Dirk Jagdmann. The farbrausch way to make demos: Procedural generation of textures and 3d-objects, 2008. URL https://llg.cubic.org/docs/farbrauschDemos/.

[4] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. SIGGRAPH '01: Proceedings of The 28th Annual Conference on Computer Graphics and Interactive Techniques. Association for Computing Machinery, 2001. URL https://people.eecs.berkeley.edu/~efros/research/quilting/quilting.pdf.

[5] Junyu Dong, Jun Liu, Kang Yao, Mike Chantler, Lin Qi, Hui Yu, and Muwei Jian. Survey of procedural methods for two-dimensional texture generation. *Special Issue Networked Sensing for Autonomous Cyber-Physical Systems: Theory and Applications*, 20, 2020. URL https://www.mdpi.com/1424-8220/20/4/1135.