CS399I Research

Adam De Broeck

January 3, 2024

The Objective

The key purpose of this project is to develop a technique that is implementable in a modern game development pipeline for generating procedural textures. This technique must maintain a handful of properties that allow for a seamless design pipeline. Firstly, the technique must be easy to use and iterate upon existing textures given an already formed texture. In order to satisfy these criteria, simply generating a whole texture in one algorithm, even when provided knobs and handles to tune the texture is not an adequate solution. Textures will need to be designable in a step by step fashion, similar to working on a texture within in a typical rasterization program.

Secondly, textures must be visualizable during each step of the creation process — this is because in order for the textures to have some hierarchical structure to them, tuning knobs must apply progressively. This is another reason why a simple input/output algorithm with tuning knobs will not suffice.

Lastly, the textures must be deterministic and seed-able so that they can be generated in a guaranteed way at runtime. While not all application will make use of this property, it's important to have for applications that have more sensitive requirements such as image matching.

Ultimately, this leads to three overarching requirements, instruction based file structure (for the iterative generation approach), a tooling application for use in designing the textures that will be used, and a library to load and compile textures at runtime. That being said, there are a couple avenues of research that will be covered which include: instruction based asset generation (Farbrausch), traditional texture generation methods (Texturing & Modelling textbook), ways to apply texture synthesis to tradition methods (UC Berkley paper), and existing tools for instruction based texture work. This article serves to give a summary explanation of the sources chosen and why they will be useful research to the project going forward.

Farbrausch: Runtime Assets

The inspiration for this project comes heavily from that of Farbrausch[1], a German group that are known for their visualizations and demos shown off at Demoscene and other graphical conferences/conventions. One such project in particular, .kkrieger, demonstrated a need for "compression" of asset data in order to fit the entire application within 96KB. While this project doesn't aim for compression goals, the tech that they developed in order to achieve this feat is particularly useful.

In order to achieve this compression Farbrausch came up with a method of generating the required assets at runtime instead of packaging those assets in with the application. By storing the instructions needed and the algorithms used in order to create the textures instead of the finalized asset itself, file sizes could be drastically reduced. The tool(s) that they developed for this would go on to be their program .werkzeug.

This concept of creating assets from scratch using essentially "scripts" or commands has useful applications for procedural generation as well. Instead of having a rigid set of instructions, a user could provide a range of values on certain parts of the instruction set in order to have the computer randomly create a range of values to use rather than a predefined one. By allowing ranges to be set in different steps in the set of instructions, constrained randomly generated assets could be created.

This project will limit the scope of those assets to textures but the idea is the same. The benefits of this technique for procedural generation are its clear hierarchical structure to asset generation as well as an easy to iterate upon pipeline which should make asset workflows incredibly smooth. That being said, Farbrausch's work is available open source on GitHub[2] and the source provided is a presentation of the thought process behind the tech. These sources will be invaluable as the foundation of the project.

Existing PCG Texture Methods

Traditionally, procedurally generated textures and/or images use well known combinations of algorithms, noise generation and mathematical functions to create some sort of input to output mapping of a given space. If Perlin Noise takes in parameters to build a continuous functional representation of noise via octaves, then this type of procedural generation as a whole does effectively the same but with less randomness and more predictable patterns. Typically articles on such techniques define a procedural texture

as something with infinitely scalability, similar to a fractal or Mandelbrot set. This property is actually not very useful for the application desired (video game graphics) and is even at times detrimental — retro games with intentional pixel art styles. Regardless of this, many of these traditional techniques provide very useful patterns and ideas for generating textures, albeit they are typically very "uniform" or "lacking in novelty".

A great start for looking into such methods is a survey performed and published by a group of computer science departments from various universities. The article, *Survey of Procedural Methods for Two-Dimensional Texture Generation*[3], describes a wide variety of techniques that can be utilized from cellular automata to Voronoi diagrams to even simulations. This source not only helps provide a variety of ideas to look at when it comes to traditional techniques, but it also gives an explanation and overview to each technique. As such, this article is a great starting place for work on traditional methods.

The next source is a web article by Shea McCombs[4] as an introductory in procedural textures. It focuses on the functional side of texture generation and shows an in-depth explanation of how combining functions in a 2-Dimensional space can create images — very similar to the discrete cosine transform which is used in JPEG compression. The article also explains a little behind Voronoi diagrams and noise functions from a functional perspective. Even though the scope is very narrow, this source provides great information about combining functions which is very useful for creating new functional methods of texture generation by adding together older ones.

The last source for traditional methods is a textbook simply titled, *Texturing and Modeling: A Procedural Approach*[5]. This book goes into extremely fine implementation detail for a large number of the mentioned traditional generation techniques above. If the first source describes the *what* for procedural generation, this source describes the *how*. It's no surprise that one of the contributing authors happens to be Ken Perlin, whose noise algorithm is the backbone of many of these techniques. This source will be invaluable during the implementation process for each tradition generation method.

Unfortunately, while all the methods described above are fantastic approaches to procedural textures, they don't align with the projects goals very well. For starters, each method takes some parameters as an input and provides the built texture from scratch in one pass. Given that the project's goal is to design an iterable instruction based toolset, these techniques could realistically only be used as starting points for a given texture (generating after some work is already done would simply replace that work).

At this point there appeared to be a disconnect with no way of linking the concepts of both traditional PCG textures and the projects goal of creating instruction based PCG textures. That is, until stumbling across a small slide deck from Western Kentucky University.

Texture Synthesis

The aforementioned slide deck, simply titled *Texturing: Procedural Texture Generation*[6], describes similar methods to those above — namely noise based generation — but provides an additional field of study called texture synthesis. The basic applications of texture synthesis involve expanding a texture/image to a larger and seamless image based on a smaller original image — e.g., if the original image is a small portion of a chain-link fence, then the synthesized image would be the whole fence. Typically, this is done by image quilting, which involves copying and pasting portions of the original image and shifting them in order to make them fit together in the final image. On its own, this technique sounds nice to use as a supplementary method for the project's instruction based generation, however, the real use-case comes when this technique is used to replace parts of an existing image.

By forcing an image quilting algorithm to always match the quilted image best within the center of the image and not just the edges, a donor texture can effectively be "mapped" onto a receiving texture. The result is the ability to "apply textures" to other textures. By doing this, all the previous traditional texture generation approaches suddenly become valid and usable with the instruction based generation method by texture mapping a tradition texture onto a region of the new texture. The slide deck above is actually a smaller presentation of the original larger proceeding from a SIGGRAPH conference — *Image Quilting for Texture Synthesis and Transfer*[7].

An additional source of texture quilting/synthesis that contains multiple other papers on the subject (should the need arise) is a web page maintained by Stanford's Graphics Department. The page is simply titled, *Texture Analysis and Synthesis*[8]. While the first source really just obtains information from the second source and won't likely be used, the second and third source contain an abundance of information regarding the texture quilting and texture mapping techniques.

Other Possible Techniques

One other final technique that is at least worth mentioning is wave function collapse. Given the grid-like structure of image quilting, it's possible both wave function collapse and image quilting could be used at the same time in order to create a greater depth of variance in the resulting texture. A video titled, *Procedural Generation with Wave Function Collapse and Model Synthesis — Unity Devlog*[9], made by a creator that goes by DV Gen explains some pros, cons, and possible problems when working with wave function collapse. As an example of how this algorithm can be useful: during the image quilting process, if a different texture is swapped out mid-quilting (depending on a set of parameters), or the quilting order is changed (depending on surrounding data), a more organic and less uniform texture could be generated. This idea takes a lot for granted however, so any attempts to do so should be treated as an extension to the original project.

Planned Generation methods

Now that it's possible to use both Farbrausch's iterative generation technique and tradition texture generation techniques with the help of texture synthesis, the question that remains is how PCG will work in the project. The goal of the project is of course to create a sort of "script" of commands used to generate a texture. Those commands include traditional generation techniques as well as some more common operations like selecting portions of the image, basics shapes, basic effects like distorts or blurs, and more. In order to let users control which aspect of the texture is procedurally generated, each command will have an option to provide a random range of values instead of a concrete value. This lets a user chose where in the generation pipeline the texture gets a randomly applied element.

While a user is allowed to set everything to be random, it's very likely that only specific values should be set to random parameters in order to achieve like-ness and/or predictable textures. Given the "riff-on-it" nature of this method, where the algorithm takes a known deterministic texture and changes it within constrained parameters, this method will be called the permutation method. A different approach to this is to instead define a texture solely by its properties. In this case properties would be tagged onto a texture and would be applied as small "macros" called `properties` which are predefined by the user. Each macro would perform a constrained random operation as a sort of function to an existing input texture. The order in

which properties are applied to some base texture is randomly decided by the generation algorithm.

Naturally, this method will be called the property method. Creating a tool and structure for performing both methods simultaneously is the final goal of this project.

Tooling Requirements

There are already some tools which exist that perform similar types of operations to the ones needed in this project. Of course, none of these tools (that are known) have the capabilities for adding constrained randomness to their generation methods. However, it should be useful to look at existing tools for inspiration when creating a smooth and easy to use workflow for a prospective user. One article titled, *TextureLab—Open Source Procedural Texture Generation*[10], provides a video and some links regarding existing texture creation tools.

This project's tool has some required properties in order to support each generation method. The tool must first have some sort of flow to the instructions; like a scripting language, operations need to have an execution order. Tools like Unreal's material graphs and blueprints use connectable nodes so show flow direction. Unfortunately, this is out of the scope of this project, so a method of ordered call blocks will be used instead.

The tool must also have a preview interface and a method of "compiling" the texture to show current changes without immediately applying them on parameter change. Additionally, the tool (and textures for that matter) must be deterministic and seed-able to allow for users to guarantee a randomized texture is the same upon texture compilation should they desire that. The texture files must be written to a file as a simple set of instructions and/or properties and the tooling must support creating functions/groups/macros as is required by the property based generation method. In order for texture synthesis to work, the tool must also support the creation, movement, duplication and deletion of layers for each texture.

The basic required texture instructions to include are as follows: Selections (rect, poly, clear, add/override/subtract), Fill, Gradient, Line, Rect, Circle, Triangle, Poly, Motion Blur, Gaussian Blur, Fragmentation, HSV Modify, Crystallize, Erode, Dilate, Perlin Noise, Uniform Noise, Transformation (rotate, translate, scale), Texture Quilting (seamless edge, expand), Texture Mapping, Dent/Bulge/Warp, and a variety of tradition generation methods (Voronoi, cellular automata, functions, etc).

References

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

[2] Farbrausch. Farbrausch demo tools 2001-2011. URL https://github.com/farbrausch/fr_public.

[3] 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.

[4] Shea McCombs. Intro to procedural textures. URL http://www.upvector.com/?section=Tutorials&subsection=Intro%20to%20Procedural%20Textures.

[5] 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.

[6] Western Kentucky University. Texturing: Procedural texture generation. URL http://people.wku.edu/qi.li/teaching/446/cg13_texturing.pdf.

[7] 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.

[8] Li-Yi Wei and Marc Levoy. Texture analysis and synthesis, 1999-2003. URL https://graphics.stanford.edu/projects/texture/. Stanford University.

[9] DV Gen. Procedural generation with wave function collapse and model synthesis — unity devlog, 2023. URL https://www.youtube.com/watch?v=zIRTOgfsjl0.

[10] Texturelab—open source procedural texture generation, 2020. URL https://gamefromscratch.com/texturelab-open-source-procedural-texture-generation/.