Organic Tile Textures using Fragment Shaders

Luís Gustavo R. Moreira* Wallace S. Lages‡
Universidade Federal de Minas Gerais, School of Fine Arts, Brazil

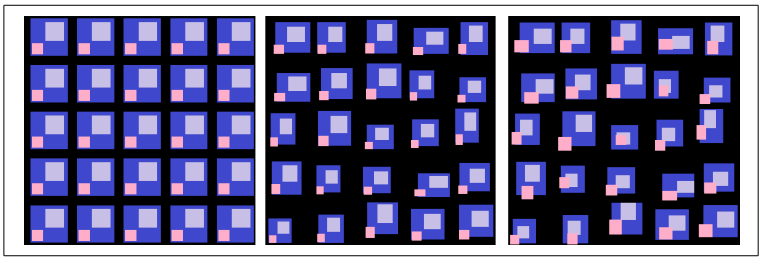

Figure 1: Example on how the code handles variation and parent-child relationship. From left to right: no variation; parent variation; parent and child variation.

Abstract

Tile textures are present in any modern 3D game. The basic concept is that of an image repeating itself in a seamless loop. However, the same pattern repeated over and over can look artificial, so variation on the texture is frequently needed. Variations are usually created by overlapping or alternating individual textures, a solution which is strongly limited by the number of textures and memory available. This paper describes a process for generating organic tiles using a small number of textures to generate infinite variations. With a custom fragment shader and a template created by the user, multiple textures can be combined and randomized in each tile. The result is a tile texture in which every cell is unique, without the need to design create each one individually.

Keywords: procedural texture, tile texture, fragment shader.

Authors' contact:
*luisgustavorm23@gmail.com
‡wlages@ufmg.br

1. Introduction

Texturing is a cheap way of generating detail in 3D models without the need to increase the underlying geometry. Big surfaces and screen areas need larger textures, so storage on the GPU starts to be an issue on high quality games.

A simple solution is to use a smaller texture repeated along the surface. However this often gives rise to repetitive patterns that distract the player from the game experience. Other techniques can be used to reduce memory footprint and at the same time add variation to the texture. Procedural texture generation [Perlin 1985; Lefebvre 2003] and synthesis from smaller samples [Efros 2001; Lefebvre 2006; Ashikhmin 2001] have been proposed as a solution to this problem.

This paper introduces a method that combines traits from procedural generation [LAGAE 2010] and texture synthesis [ASHIKHMIN 2001] to create tile textures with artist-controlled variation. The 3D artist needs only to design a template for one tile. This template will then be used as a model for each tile in the textured mesh but adding a pre-defined variation each time it is repeated. In a brick texture, for example, the artist first design a base brick, divide it into separated elements and give each one parameters that will guide the variation (position, scale, rotation, opacity, parent-child relationships). Then, the fragment shader will read those values, along with the textures that form the base brick, organizing each element into the final output. An example of the variation result can be seen on Figure 2. A simple program was developed to design the texture and output a GLSL fragment shader that is specific to the generated texture. This removes the need for parameter inheritance on the shader (with the exception of textures and UV coordinates).

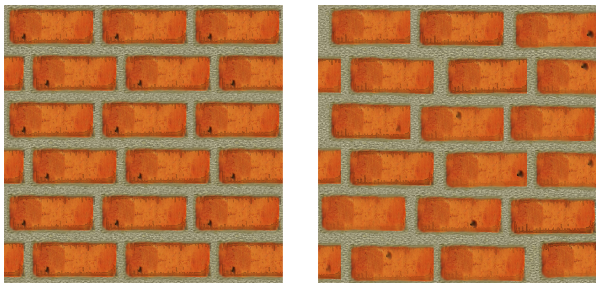

Figure 2: Left: brick texture without variation, Right: brick texture using variation, using the custom shader.

2. Related Work

Much work has been done in the past years to enhance texture mapping. Procedural generation solve the problem of texture variation and memory but are generally limited to a few types of natural textures, such as wood and marble [Perlin 1985]. Texture synthesis techniques can generate large textures without patterns by sampling smaller textures.

However, the use of complex algorithms make most of them too heavy for realtime rendering. Intermediary solutions try to pre-compute part of the problem to increase efficiency and keep texture variation.

Regarding variations on tile texture, Wei [2004] describes a method which uses alternative tile variations packed together in a single texture. Compatible tiles are then chosen at random using a fragment shader to create organic variation. This method focuses on reducing the memory passed to the GPU, using a small texture to generate a virtually infinite image. The visual output has the same image quality as the base texture but with more variation, thus, more organic.

Ashikhmin [2001] describes the synthesis of natural textures using sample images and user interaction to create a new and bigger texture. In many ways the newly generated texture is better than a tiled smaller texture. It is organic, does not show recognizable patterns and has a reliable user control. However, the final result is a flattened image. Since it is not designed to be tile texture, it would consume a large amount of memory on the GPU if used on a big structure.

[Risser et al. 2010] presents an interesting proposal. The idea is to use sample images to create hybrids with variations, taking coordinates from the samples, mixing and correcting them. The result is an image that is a blend of the original samples, but with enough variation to avoid the recognition of patterns. This hybridization process can be used to create various similar textures from a few samples. But just as other methods [ASHIKHMIN 2001; KWATRA 2005; LASRALM 2012] this synthesis is not able to be precessed in real time.

Allegorithmic, with its Substance Designer software, proposes a new mix between procedural generation and standard texture mapping. The artist starts working with basic code generated structures, and then applies various transformations onto them. The transformations add up on each other and an intuitive interface enables the user to control the process to get the desired output [Kerr Pellacini 2010]. In the final step the result is baked into a texture, so all the control the artist had while designing the texture is lost at that point. The method proposed by this paper is able to keep a level of control over the texture even after the artist is done, and the shader itself is able to interfere into the texture.

3. Texture Synthesis from Model

Our method is another take on the concept of tile texture generation. Instead of just mapping a pre-computed set of tiles or trying to synthesize a bigger texture from one flat sample image, we propose that the texture be generated in realtime from a model that describes its structure and the base textures used by the artist.

The idea is based on the fact that most game textures are created by artists methodically compositing several layers by hand in an editing software. When the artist is satisfied, these layers are flattened and exported to the game engine. Before export, however, these files contain information about the structure of the image that can be used to create a model suitable for procedural generation. Based on this model parameters, a fragment shader can generate infinite tiles variation.

3.1 Fragment Shader Generation

The fragment shader itself is texture-specific. The number of base textures and each of their parameters are not passed as varying, or fragment shader input, but are written into the code itself. For this purpose, a simple program was developed to generate the code for the fragment shader. This program enables the user to organize the base textures in layers, define the base values for each parameter as well as parent-child relationships. The program output is a text file with the code for the shader, written in GLSL, ready to be used by the Graphic Hardware with support for OpenGL 4.0 or greater. The code can then be edited by simple text editing programs, if needed.

3.2 Fragment Shader Description

Once generated, the fragment shader follows a simple pipeline, processing parent-child relationship, defining offsets, scaling factors and other parameters.
The pipeline is defined by:

- Input textures definition,
- Global variables declaration, such as number of rows and columns,
- Global variables definition,
- Fragment color definition for each input texture, using the parameters given by the user while generating the code,
- Alpha color mask (if needed),
- Texture's fragment color blending,
- Fragment color output.

The definition of the input textures is a simple inheritance of the texture data into sampler2D variables and UV data into vec2 variables. The code use UV textures as a guide for position values, so they do not work as they normally would.

Some global variables are declared to control the code. As mentioned before, the number of rows and columns are global variables, as well as a copy of the UV textures, parent and child indexes and some

functions for pseudo-random number generation. The seed used for this random number generation is related to the specific row and column of each tile, so each fragment keeps the same variation value while in the same row-column combination. Those variables are defined at the start of the main function and can be edited at any time, even though the generation program will not change these variables. For example: for a brick shader, the second row of bricks has an offset on coordinate X. This offset has to be changed manually in the fragment color definition stage of the pipeline.

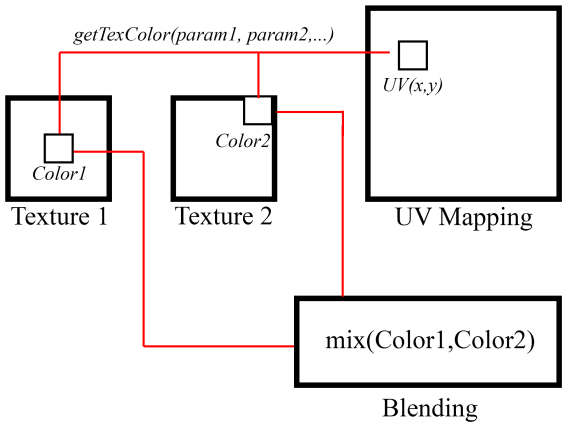

Figure 3: How the shader selects the color for its output

As illustrated on Figure 3, to define the color to be used for each texture, the shader uses a custom function that takes the parameters for each texture as arguments, then applies the transformations, returning a vec4 color. The default transformations are position, scaling and rotation, and are randomized following the user-defined minimum and maximum values. More transformations can be added to the function by manual scripting. The function uses the black color (0,0,0,1) as a transparency mask. This color can be changed or removed simply by editing the code. The blending of each texture is defined by the user at the generation of the code, and can be customized beyond the program presets (mix, add, subtract and multiply). At this stage any texture can be applied as a mask, allowing further shader customization. After that, the shader will output a single vec4 color, to be displayed on screen.

3.3 Performance

Performance can be an issue if too many textures are used and the texture has too many pixels on screen, since all the code runs on the fragment shader. The shader is most useful when applied to an extensive mesh that will not be on the screen entirely, such as walls, floors, and big objects. It can be used in small objects that have small number of pixels on screen, but this would devalue the variation of the tiles.

A program was developed to test the shader in a simple mesh that is always entirely on screen. Performance was measured by the medium stable FPS obtained. The results are shown in Figure 4. No manual code optimization were made during the test.

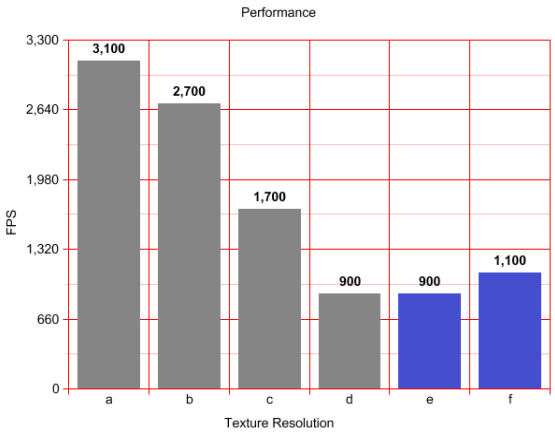

Figure 4: Performance comparison. The gray columns corresponds to standard shaders and the blue ones to the custom shader from this paper. From left to right: (a) one 256x256 texture; (b) one 512x512 texture; (c) one 1024x1024 texture; (d) one 2048x2048 texture; (e) four 256x256 textures; (f) one 256x256, two 128x128 and one 64x64 textures.

4. Usage Scenarios

As stated above, the shader has some clear uses. Some example scenarios that could use this shader are:

- A long hallway in an old building, with cracks and holes randomly positioned into the wall. Not-functional doors and windows can be randomized, if needed.
- Tiled floor with a specular mask as a child texture, making some tiles reflect more light than others, by randomizing the specular mask opacity value. The tiles can have randomized offsets and scaling values to add variation.
- Extensive mesh of desert floor, using sand texture as a tile. Rock formations, solid areas, grass and other elements are randomly placed and scaled, without need to map the entire mesh. The elements can use different rows and columns values.

Figure 5: Brick texture, wood texture, and tile texture.

4. Conclusion and Future Work

This work presents a technique to create large non periodical textures from tiles, using an artist created template and a small set of base textures. Instead of precomputing the tiles, the final texture is generated in real time, opening new possibilities for texture design, since the artist can focus in each element separately instead of the whole tile. Tile appearance can also be modified on the fly as a level of detail technique.

For future work, performance improvement is a major concern. Currently, to keep a reasonable performance one has to optimize the code manually. Breaking the shader into two parts may be a possible solution: one part would be processed by the GPU and the other by the CPU. This could make algorithms such as [Risser et al. 2010] possible to be executed by the program in real time, adding another layer of variation into the final output.

Vector textures [WANG 2010] are an interesting concept to expand on, since they would not lose quality when scaled, and could be implemented as another type of layer. Since the algorithm would return a color anyway, those new kinds of layers would not interfere with the rest of the process.

A software release is intended for the shader generator program, but it's interface needs improvement to be more user-friendly and permit user-code tweaks, to enhance the control the artist has over the final product.

Finally, a game should be made using the improved version of the shader. The game will be used as a testing ground for the technique, helping us to detect points for improvement.

Acknowledgements

The authors would like to thank CAPES and the program “Jovens Talentos para a Ciência” for the support that made this work possible.

References

ASHIKHMIN, M., 2001. *Synthesizing natural textures*. In Proceedings of the 2001 symposium on Interactive 3D graphics (I3D '01). ACM, New York, NY, USA, 217-226.

EFROS A. A., FREEMAN W. T.: *Image quilting for texture synthesis and transfer.* In Proceedings of ACM SIGGRAPH 2001 (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 341

KERR W. B., PELLACINI F.: *Toward evaluating material design interface paradigms for novice users.* Transactions on Graphics (2010)

KWATRA, V., ESSA, I., BOBICK, A., KWATRA, N., 2005. *Texture Optimization for Example-Based Synthesis*. Proc. ACM Transactions on Graphics, SIGGRAPH 2005.

LAGAE A., LEFEBVRE S., COOK R., DEROSE T., DRETTAKISG., E BERTD., LEWIS J., PERLIN K., ZWICKER M.: *State of the art in procedural noise functions.* In EG 2010 -State of the Art Reports (2010)

LASRAML, A., LEFEBVREL, S., DAMEZ, C., 2012. 2012. *Procedural texture preview*. Comp. Graph. Forum 31, 2pt2 (May 2012), 413-420.

LEFEBVRE, S., AND NEYRET, F. 2003. *Pattern based procedural textures.* Symposium on Interactive 3D Graphics, 203-212 – 346

LEFEBVRE, S., HOPPE, H., 2006. *Appearance-Space texture synthesis*. In ACM SIGGRAPH 2006 Papers (SIGGRAPH '06). ACM, New York, NY, USA, 541-548.

PERLIN K.: *An image synthesizer.* In Computer Graphics (Proceedings of ACM SIGGRAPH 85) (1985), vol. 19, pp. 287–296

RISSER, E., HAN, C., DAHYOT, R., GRINSPUN, E., 2010. *Synthesizing Structured Image Hybrids.* In ACM SIGGRAPH 2010 papers (SIGGRAPH '10), Hugues Hoppe (Ed.). ACM, New York, NY, USA, , Article 85 , 6 pages.

WANG, L., ZHOU, K., YU, Y., GUO, B., 2010. *Vector Solid Textures.* ACM Transactions on Graphics (SIGGRAPH 2010), 1-8

WEI, L.-Y, 2004. *Tile-Based Texture Mapping on Graphics Hardware*. In ACM SIGGRAPH 2004 Sketches (SIGGRAPH '04), Ronen Barzel (Ed.). ACM, New York, NY, USA, 67-.