GPU BASED **TEXTURE SYNTHESIS**

BY BOYC OF CONSPIRACY

zine BEHIND THE SCENE

I first experimented with texture generation in 2000. For a while I didn't really get far because there weren't too many resources on the topic - until I found an article written by Ile/Aardbei, which I still highly recommend to anyone new to the subject. The article you're reading is meant to be a follow up to Ile's - we now have graphical hardware that is basically built to do this sort of stuff, but using only the GPU to implement a texture generator can be tricky due to the nature of pixel shaders.

In this article I'm going to describe a solution that worked out pretty well for us. It is recommended that you already have some knowledge of texture generation and shader programming when reading this.

GETTING STARTED

When I set out to move our texture generator to the GPU, I had a few goals in mind:

- A common, simple rendering function to execute all filters (so optimally only the shaders change, and no extra cpu code is needed for any operator)
- Recreate each filter from the old texgen so we don't lose functionality
- The filters had to be as similar as possible to their old counterparts
- One pixelshader per filter, one vertexshader for the whole texgen
- For the sake of compatibility, it all had to fit into shader model 2.0

Most of these goals proved possible, with only small tradeoffs here and there.

When you're writing a filter for a CPU based texture generator, you're working with two kinds of input data: filter parameters and the outputs of previously executed filters. On the GPU this is not dealt with differently; you have to pass the same data to the pixel shader. The filter parameters are loaded as uniforms into the pixel shader, and the intermediate images are loaded as textures.

On the CPU side you'll of course need some code to execute the filters; we'll call this the rendering function. Most of the filters used in common texture generators can be recreated by rendering a single pixelshaded quad onto a texture, so the best idea is to start here and expand from this as needed.

> **USING ONLY THE GPU TO IMPLEMENT A TEXTURE GENERATOR CAN BE TRICKY DUE TO THE NATURE OF PIXEL SHADERS**

The simplest rendering function fills the whole rendertarget by rendering a single quad on it. The pixelshader draws an image using the texture coordinates and any additional data (parameters in the form of uniforms or images in the form of textures) you provide. Some simple generators (sinusplasma, gradients, etc), layer operations (colorize, blends, etc), and most distortion operations (twirl, sinedistort, rotozoom, mapdistort etc) can be handled this way. For example you can generate a gradient by setting the x texture coordinate to be the result of the shader.

This function will get you started, however there are some filters that can't be done from shaders at all - an image loader or a text writer are good examples. These filters require lookup textures calculated the old fashioned way on the CPU before they execute. The idea is to expand the rendering function

with the ability to create such textures when needed, and not use them most of the time. The lookup texture should contain any data that the filter shader needs to finish the job (such as a loaded image, etc - however note that the function that generates the lookup texture can't have access to intermediate images from the GPU without severe performance costs, so it's not practical to try and mix CPU and GPU based generation this way.) Such lookup textures can help with a lot of filters: for example a 256x1 image can make for a nice palette to colorize the image with.

Generators based on random numbers will also use such a lookup. There is no rand() in shaders, so the next best thing to do is to supply the shader with pseudorandom data in the form of a texture. A single precalculated 1024x1024 noisemap works nicely across the whole system, however you'll need to do some transformations on the texture to avoid repeating patterns and help randomize your results. Scaling with an x>1 nonround value combined with a random offset will do the trick most of the time.

The remaining filters (stuff like blur, perlin noise, etc) will need another expansion of the rendering function: support for multiple passes. Working with multiple passes requires a temporary rendertarget, which needs to be swapped between passes much like front/backbuffers are swapped between frames. This way the rendering function can leave any original input rendertargets intact for further use by other filters. Of course rendering with the same parameters in multiple passes will mostly do the exact same thing so the passcount should also be uploaded into the shaders as a parameter to use. If you need a filter that changes the number of passes based on a user given parameter, the function that would normally be overloaded to create a lookup texture can set the required number of passes before executing the filter.

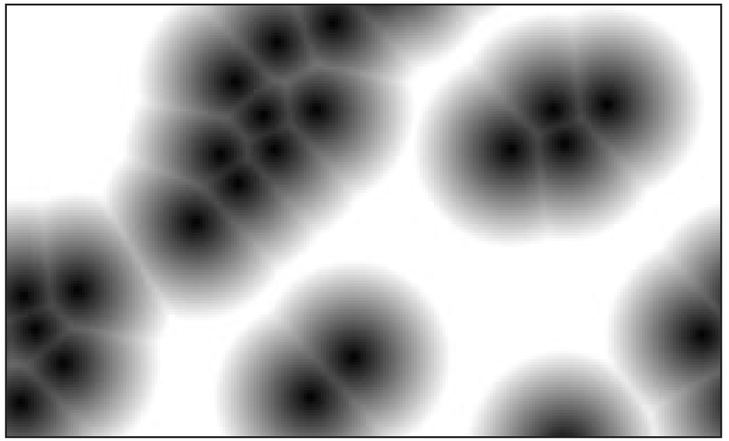
In some cases it might be desirable for a filter to have an undefined number of inputs (a combine filter for example, where the number of images combined can vary). These operations can be done by rendering in multiple passes and replacing the lookup texture in each pass to the current result.

COMPLICATED SHADERS

Let's have a look at how to do some of the more complicated filters with this system:

Blur: The blur filter can be divided into an x and a y pass. The idea is to take a fixed number of samples from the input and combine these by using a distance function (gaussian, linear, etc). The size of the blur can be varied by changing the distance between the sampling points; however this will add a ghosting effect after a certain distance due to the limited number of used samples. To fix this additional x and y passes can be added to the filter.

Subplasma: In his article Ile used catmull-rom interpolation for low resolution noise to get a nice subplasma effect. Unfortunately to do this in a single pass would require taking 16 samples from the noisemap and mixing them in a way that exceeds the limitations of the ps2.0 model. It's possible to get around this with the multipass approach with a single shader: The first pass samples the noise and interpolates it in the x direction the second pass interpolates the result in the y direction.

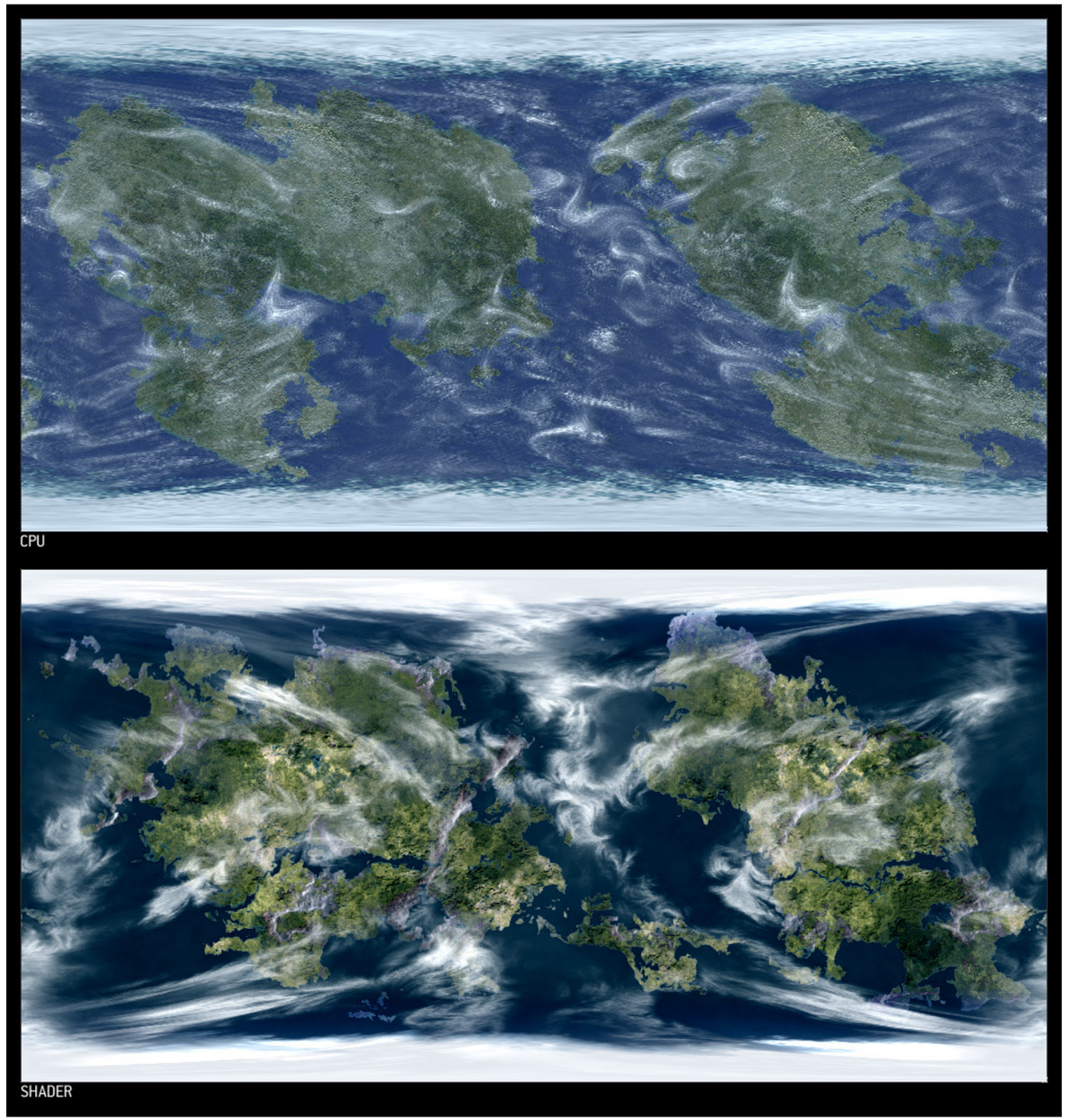


A cell clustering

Perlin noise: To mix several of the above described subplasma effects and create perlin noise that way is not possible in this

system. An extra temporary buffer would be required to do that, which I find is a bit of overkill for a single filter to work. It's possible to do linear interpolated noise instead of catmull-rom in one pass though, which will work just as well.

Cells: Traditionally, to create a cells effect one takes a number of points, and for each point of the texture takes the distance to the closest one. The problem with implementing this approach in a shader is that the number of points will be fixed and fairly limited. The same effect can be achieved by taking a simple distance map (the distance of the current texture coordinate from the center), adding a random offset, and combining it with itself with a min() operation. This can be done for n points in n passes, or it can be done recursively which is a lot faster but will produce weird clustering effects with some random seeds (as seen on the picture). The catch is that effects that would require the second closest point (cells borders for example) aren't possible this way.



PROS AND CONS COMPARED TO CPU

Pros:

» A lot faster thanks to hardware that is built to do this

» Smaller - the shader code is a lot less complicated than its cpu counterpart (in numbers: cpu texgen = 11743 bytes, gpu texgen = 6959 bytes, both with all the filters included, kkrunchied)

» Texture filter editor can be easily implemented in a demotool - no more need to recompile the texgen to add/remove filters

» A lot of the filters can be reused as video postprocessing effects

» Quality - it's easy to switch from 8 bit rgba to float textures if needed

» Texture sizes can vary between filters, the video card takes care of that

» Almost realtime speeds even on more complex operator chains and the graphician can work easily due to immediate feedback

Cons:

» Less compatibility (requires ps2.0 videocard)

» Memory management requires more work than on the cpu (especially when working with big textures)

» Multithreading can be a problem

Any questions are welcome at boyc(at)conspiracy(dot)hu

I'd like to thank ryg/Farbrausch and jimmi/TGD for their help and ideas.

PSEUDO CODE FOR THE FINAL RENDERING FUNCTION

```
void execute(Input, Output)
{
 // create lookup texture and/or change passcount according to input parameters
 lut=prerendersetup();

 Target1=Output;
 Target2=create_temporary_rendertarget();

 // ensure correct output
 if (passcount&1) swaptargets( Target1, Target2 );

 for (x=0; x<passcount; x++)
 {
  // send pass specific data for the shader
  setpassdata(x);

  // replace the lookup texture if needed
  lut=replacelookuptexture(x);

  // render quad to the current target
  if (!x) render( x, Input, Target1, lut );
  else render( x, Target2, Target1, lut );

  // swap target buffers
  swaptargets( Target1, Target2 );
 }

 // free temporary textures
 delete lut;
```