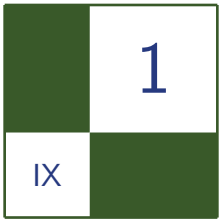


# Stylized Rendering in *Spore*

Shalin Shodhan and Andrew Willmott

As screen effects and deferred shading become standard fare in game engines, a good compositing tool becomes a key part of the rendering pipeline. In the game *Spore*, we used a scriptable filter chain system to process frames at runtime. A filter chain is essentially just a series of parameterized image processing shaders applied in order. Every frame in our game is processed and composited using this system. In addition to the standard art-directed look of *Spore*, we created a set of specialized filters to generate dramatically different visual styles for the game. Figure 1.1 shows a *Spore* rendered as an oil painting. These were so compelling that we shipped them with *Spore* as player-enabled cheats. In this article we would like to breakdown some of the visual styles we generated and share details about the design and implementation of our Filter Chain system.

## 1.1 Filter Chain System

GPU implementations of image processing techniques like blur, edge detection etc. are widely available. Our goal was to build a system with a palette of such filters that artists could use to author visual styles. Figure 1.2 shows how this system is placed in the rendering pipeline.

### 1.1.1 Design Considerations

Rapid iteration and scripting. Rapid iteration is key for artists working in accordance with art direction. So our scripts are hot-loadable in development builds. This means that an artist can tweak a parameter or change the script and see the updated results while the game is running. The scripting itself is very simple. In fact there are only two types of commands in the script! Listing 1.1 shows a simple bloom filter chain.

Figure 1.1. An airplane made in *Spore* rendered as an oil painting.

Filters. Each filter corresponds to a single shader applied to a screen aligned quad. It can take up to four textures and up to 16 constant float parameters as input. It is scripted using the following command:

```
<filterName>  <inTex>  <outTex> [<param1> ...]
```

The `filterName` corresponds to a pre-defined set of filters like blur, compress, etc., while `inTex` and `outTex` are source and destination textures for the filter.

A number of filter specific constant parameters can be specified.

Textures. Each filter chain can declare its own set of temporary textures to store intermediate results using the following command:

```
texture <name> -ratio <integer>
```

Then `<name>` can be used as `<inTex>` or `<outTex>` for a filter. The ratio parameter lets you control the texture size as a fraction of game resolution. Artist generated textures can be accessed by their resource IDs and game-generated or reserved render targets can be referenced by keywords; for example, the framebuffer is `dest` and the unfiltered scene is `source`.

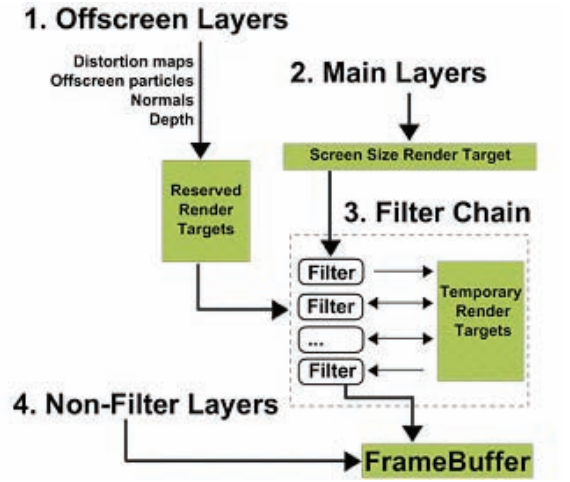


Figure 1.2. Filter Chain system overview.

Integration with effects. An easy win for us was to add this system as a component of our industrial strength effects system, Swarm. This way filter chains were well integrated into the rendering engine. We got scripting and hot-loading for free and our artists, being well-versed in Swarm, could easily get started with filter chains.

Figure 1.3 shows how the cell game's filter chain uses multiple input textures generated by other stages of the rendering pipeline and forms the final composite.

```
# Basic bloom
   filterChain
      texture blur1a -ratio 4
      texture blur2a -ratio 4

      compress source blur1a -bias -.25 -scale 1

      blur blur1a blur2a  -scaleX 2
      blur blur2a blur1a  -scaleY 2
      blur blur1a blur2a  -scaleX -2
      blur blur2a blur1a  -scaleY -2

      add source dest -texture blur1a -sourceMul .75 -addMul 1
   end
```

Listing 1.1. A simple bloom effect as a filter chain.

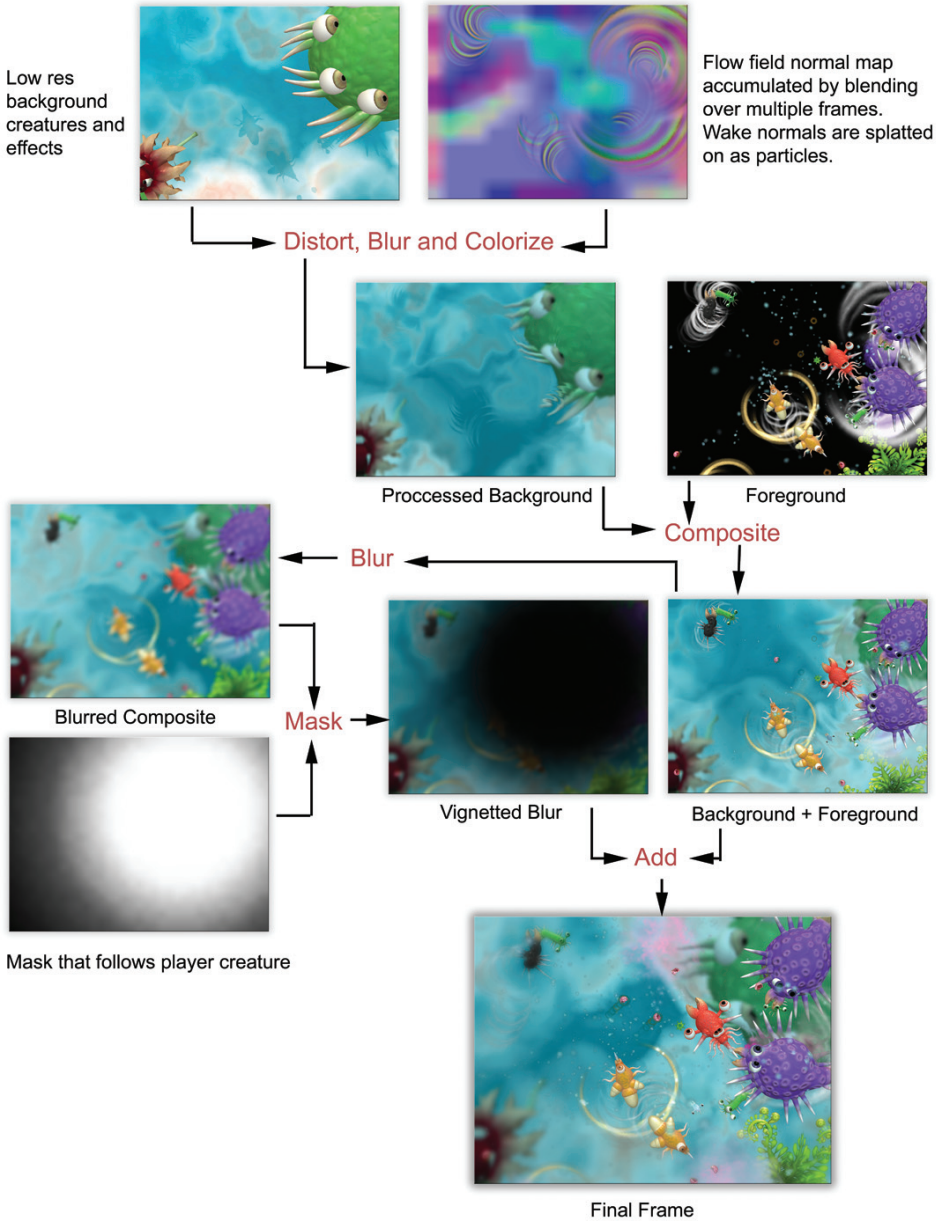


Figure 1.3. A complex filter chain for the cell game's fluid environment.

### 1.1.2 Implementation

The filter chain system we shipped with *Spore*, by and large, adheres to the design presented above. Some significant additions are given in this section.

Dynamic parameters. Spore's dynamic environment needs call for frame varying parameters. We added global parameters that update per frame and can be accessed by any filter. For example, we use camera height and time of day in doing atmospheric filters for planets as shown in Figure 1.4. In other cases the game needs to smoothly interpolate between two different sets of parameter values for a given filter. For example., whenever the weather system starts to rain, the global colorize filter's color transitions to an overcast gray. We added parameters that support game-controlled interpolation. Finally, we added faders that can smoothly change the strength of a filter.

Custom filters. An important addition to the system is a *custom filter* that can specify its shader as a parameter. This means that a programmer can add a new image technique pretty easily by just adding a new shader to an existing

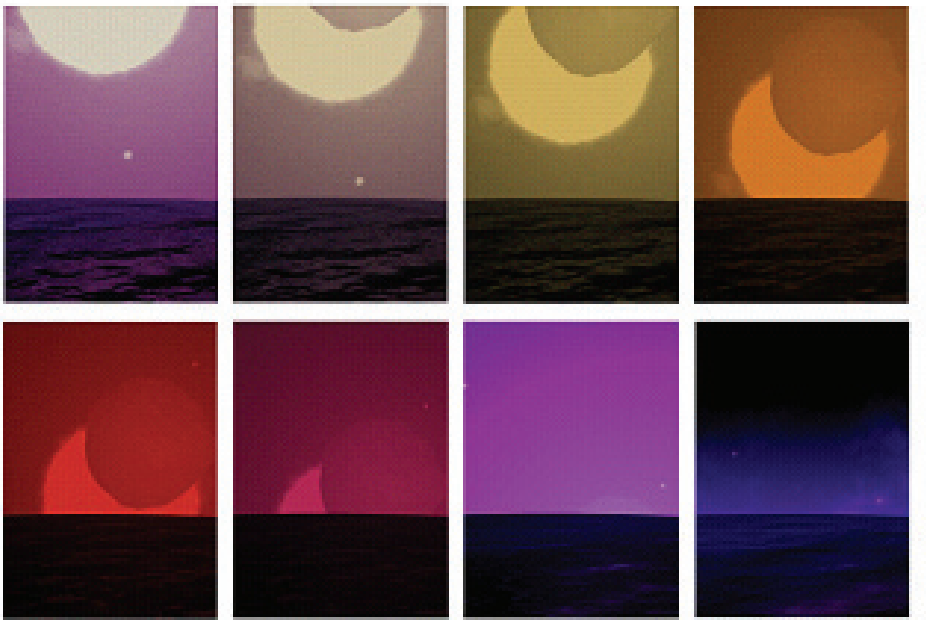

Figure 1.4. A time-of-day driven colorize filter. This colorized-compressed output is then blurred and added to the scene as bloom.

| Size | Count | Memory (KB) | Usage |
|---|---|---|---|
| $1024 \times 768$ | 3 | 9216 | One full screen buffer held the unfiltered scene. Two full screen and two half screen buffers were available for distortion maps, off screen particles etc. The remaining textures were used as intermediate buffers between filters. |
| $512 \times 384$ | 5 | 3840 | |
| $256 \times 192$ | 1 | 192 | |
| $128 \times 96$ | 2 | 96 | |

Table 1.1. Worst-case texture memory usage at $1024 \times 768$, 13MB.

build. Also, a programmer can optimize artist generated filter chains, by collapsing multiple filters into a single custom filter that achieves the same visual result.

### 1.1.3 Performance

Filter chains are fairly pixel shader heavy and choked our min spec low end Pixel Shader 2.0 cards. We wrote low, mid and high quality versions of all our filter chains to adapt based on target hardware. Custom filters are a great tool for collapsing multiple filters into a single pass and saving on fill rate.

Buffer size and texture memory usage. We heavily tuned texture sizes for scratch buffers, iterating rapidly with the `ratio` parameter for the `texture` command in filter chain script. After collecting data about texture usage for various filters we preallocated all our render targets. Since only one filter chain was allowed at a time, these textures could be reused. We tried to improve reuse by atlasing but ran into problems with texture addressing and filtering at boundaries for the atlased render targets (see Table 1.1).

## 1.2 Fun with Filters

Here are a number of experimental rendering styles available in *Spore* as cheats.

### 1.2.1 Oil Paint Filter

For an oil paint filter (see Listing 1.2), a brush stroke normal map is rendered first. This is used to distort the incoming scene. Then the same normal map is used to light the brush strokes in image space with three lights. Brush strokes can be driven by a ribbon particle effect to make the filter dynamic and more temporally coherent (see Figure 1.5).

The constants in Listing 1.2 control properties of the effect. Larger values of `kDistortionScale` make the effect more impressionistic. The `kNormalScales` values control how detailed the brush strokes look; `kBrighten` is a global adjustment. Light directions and colors can also be used to vary the look.

Figure 1.5. Oil paint filter.

```
# Oil Paint Effect
# kDistortionScale 0.01, kBrighten 2.0
# kNormalScales (1.5, 2.5, 1.6)

# Get the normal map and bring normals into [-1,1] range
half4 pNormalMap = tex2D( normalMap, fragIn.uv0 );
half3 nMapNormal = 2 * pNormalMap.rgb - half3( 1, 1, 1 );

# Distort the UVs using normals (Dependent Texture Read!)
half4 pIn = tex2D(sceneTex,
                  saturate(uv - nMapNormal.xy * kDistortionScale) );

# Generate the image space lit scene
half3 fakeTangN = nMapNormal.rbg * kNormalScales;
fakeTangN = normalize(fakeTangN);

# Do this for 3 lights and sum, choose different directions
# and colors for the lights
half NDotL = saturate(dot(kLightDir, fakeTangN));
half3 normalMappingComponent = NDotL * kLightColor;

# Combine distorted scene with lit scene
OUT.color.rgb =   pIn.rgb * normalMappingComponent * kBrighten;
```

Listing 1.2. Pixel shader snippet for oil paint effect.

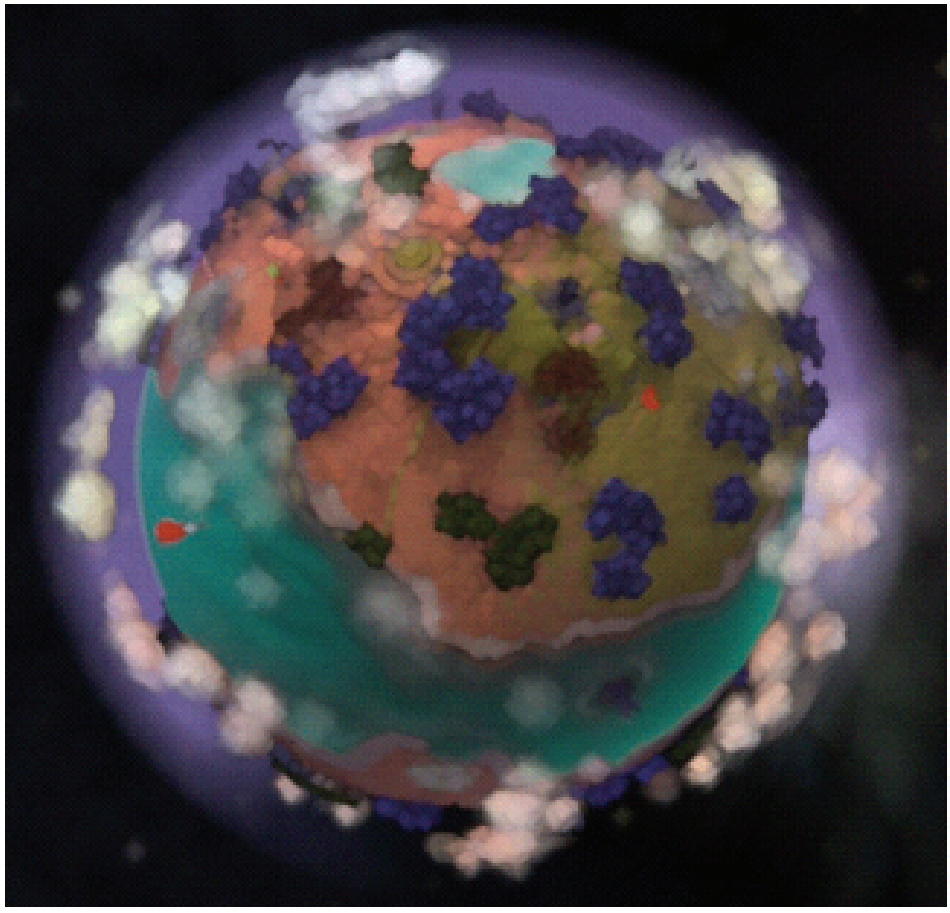

Figure 1.6. Watercolor filter.

### 1.2.2 Watercolor Filter

For a watercolor filter (see Figure 1.6), a simple Sobel edge version of the incoming scene is multiplied with the original scene. The result is then smoothed using four passes of a smoothing filter that finds the brightest value per channel from four surrounding taps. The edge detection based outlines add some definition that are lost in the smoothing. The offset values and scales in Listing 1.3 let you vary the size of the paint daubs.

```
# Water Color Smoothing
# kScaleX = 0.5, kScaleY = 0.5
# offsetX1 = 1.5 * kScaleX  offsetY1 = 1.5 * kScaleX
# offsetX2 = 0.5 * kScaleX  offsetY2 = 0.5 * kScaleY

# Get the taps
tap0 = tex2D(sceneTex, uv + float2(-offsetX1,-offsetY1));
tap1 = tex2D(sceneTex, uv + float2(-offsetX2,-offsetY2));

tap2 = tex2D(sceneTex, uv + float2(offsetX2, offsetY2));
tap3 = tex2D(sceneTex, uv + float2(offsetX1, offsetY1));

# Find highest value for each channel from all four taps
ret0 = step(tap1, tap0);
ret1 = step(tap3, tap2);
tapwin1 = tap0* ret0 + tap1 * (1.0 - ret0);
tapwin2 = tap2* ret1 + tap3 * (1.0 - ret1);
ret = step(tapwin2, tapwin1);
OUT.color.rgb = tapwin1 * ret + (1.0 -ret) * tapwin2;
```

Listing 1.3. Pixel shader snippet for a smoothing filter to do watercolor.

### 1.2.3 8-Bit Filter

To create an 8-bit filter (see Listing 1.4), use the round function in the pixel shader and draw to a low res buffer $1/4^{th}$ the size of game resolution with point sampling. This is a really simple effect that makes the game look like an old 8-bit game (see Figure 1.7).

```
# 8 Bit Filter
# kNumBits: values between 8 and 20 look good
half4 source = tex2D(sourceTex, fragIn.uv0 );
OUT.color.rgb = round(source.rgb * kNumBits) / kNumBits;
```

Listing 1.4. Pixel shader snippet for an 8-bit filter.

### 1.2.4 Film Noir Filter

In creating a film noir filter, first, the incoming scene is converted to black and white. It is then scaled and biased. Some noise is added, A rain particle effect is a nice finishing touch (see Figure 1.8). In Listing 1.5 kNoiseTile can be used to adjust the graininess. kBias and kScale serve as parameters of a linear contrast stretch.

Figure 1.7. An 8-bit filter.

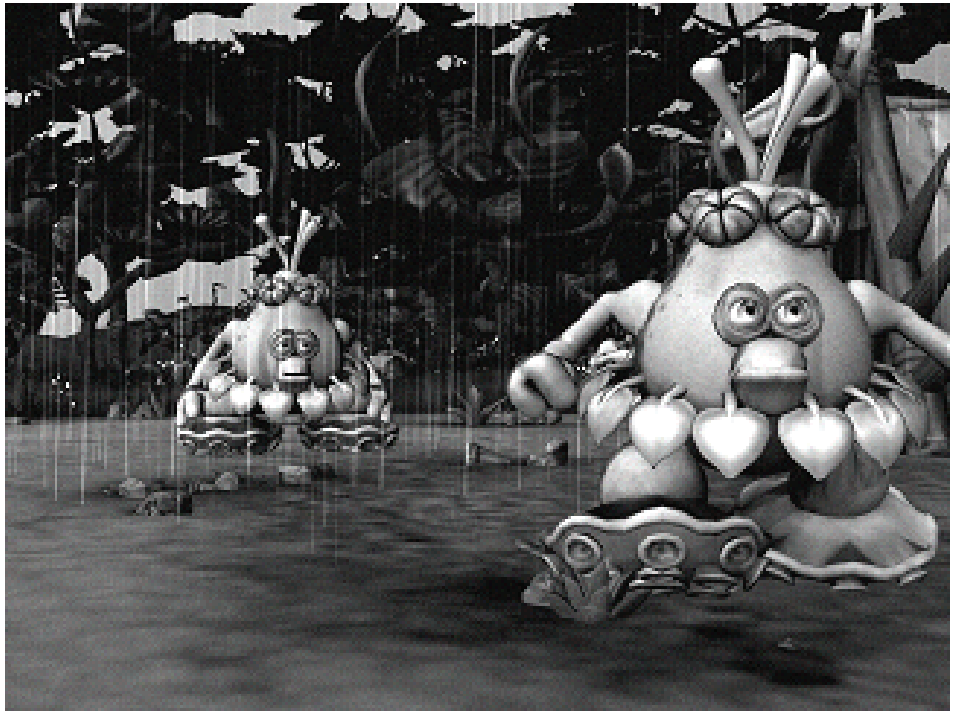

Figure 1.8. Film noir filter.

```
# Film Noir filter
# kNoiseTile is 4.0
# kBias is 0.15, kScale is 1.5
# kNoiseScale is 0.12
pIn = tex2D(sourceTex, uv);
pNoise = tex2D(noiseTex, uv * kNoiseTile) ;

# Standard desaturation
converter = half3(0.23, 0.66, 0.11);
bwColor = dot(pIn.rgb, converter);

# Scale and bias
stretched = saturate(bwColor - kBias) * kScale;

# Add
OUT.color.rgb = stretched  + pNoise * kNoiseScale;
```

Listing 1.5. Pixel shader snippet for film noir filter.

### 1.2.5 Old Film Filter

For an old film filter (see Figure 1.9), a simple sepia colorize is combined with a sharpen filter. Scratches and vignetting can be done with particle effects (see Listing 1.6).

```
# Old Film Filter
# offsetX and offsetY are 2 pixels. With such wide taps, we
# get that weird sharpness that old photos have.
# kNoiseTile is 5.0, kNoiseScale is 0.18
# kSepiaRGB is (0.8, 0.5, 0.3)

# Get the scene and noise textures
float4 sourceColor = tex2D(sourceTex, uv);
float4 noiseColor = tex2D(noiseTex, uv * kNoiseTile);

# sharpen filter
tap0 = tex2D(sceneTex, uv + float2(0, -offsetY));
tap1 = tex2D(sceneTex, uv + float2(0, offsetY));
tap2 = tex2D(sceneTex, uv + float2(-offsetX, 0));
tap3 = tex2D(sceneTex, uv + float2(offsetX, 0));
sourceColor = 5 * sourceColor - (tap0 + tap1 + tap2 + tap3 );

# Sepia colorize
float4 converter = float4(0.23, 0.66, 0.11, 0);
float bwColor = dot(sourceColor, converter);
float3 sepia = kSepiaRGB * bwColor;

# Add noise
OUT.color = sepia * kTintScale + noiseColor * kNoiseScale;
```

Listing 1.6. Pixel shader snippet for old film filter.

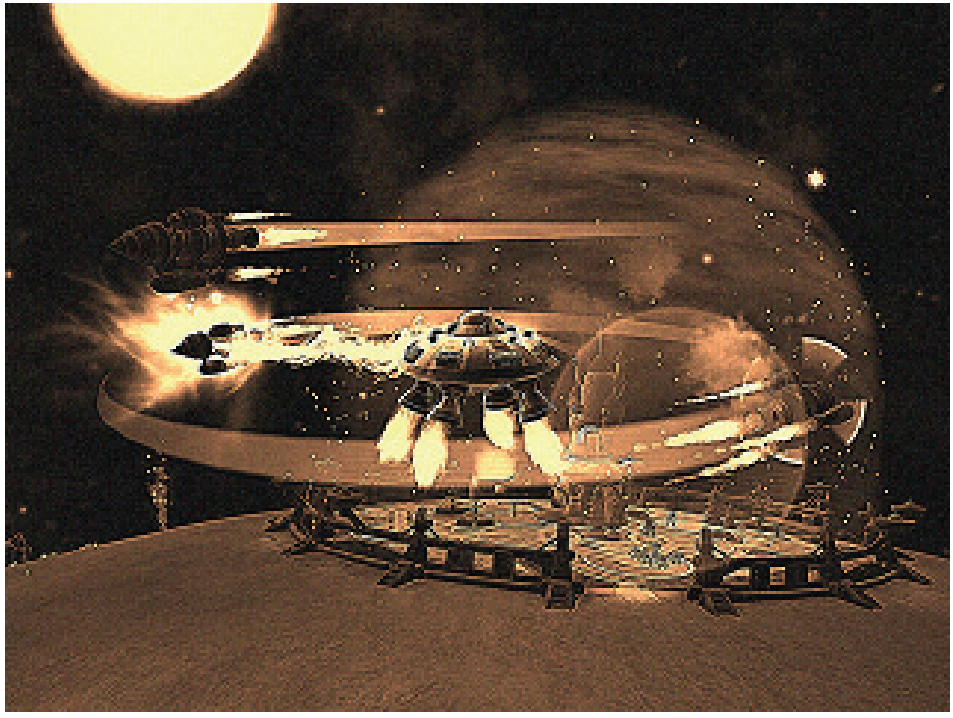

Figure 1.9. Old film filter.

## 1.3 Cheats

A list of *Spore* cheats to do stylized rendering can be found at http://www.spore.com/comm/tutorials.