JTexGen Procedural Texture Library

Andy Gibson

JTexGen Procedural Texture Library

Andy Gibson

Table of Contents

Preface

One area I have always been interested in is computer graphics, whether it is generating images from 3D models, making images more realistic using radiosity and photon mapping, or emulating natural surfaces in terms of color, shape and texture using nothing but code.

I wanted to create some images using computer generated textures to try and generate some artwork to put in my house. Originally I was thinking of doing some marble textures, maybe the odd Mandelbrot set or some kind of flame artwork. The benefits of using procedural textures are many fold. You can continually reproduce the texture at different resolutions introducing more levels of detail the finer you go as well as tweaking parameters to get different results. Plus of course it’s pretty cool generating images and textures from numbers and code. What started as a simple project ended up growing into a much larger project which I have made available, as well as this article looking at how to use the library.

Chapter 1. Introduction

This section introduces the concepts and ideas of procedural textures and how they relate to the key parts of the framework.

1.1. Overview

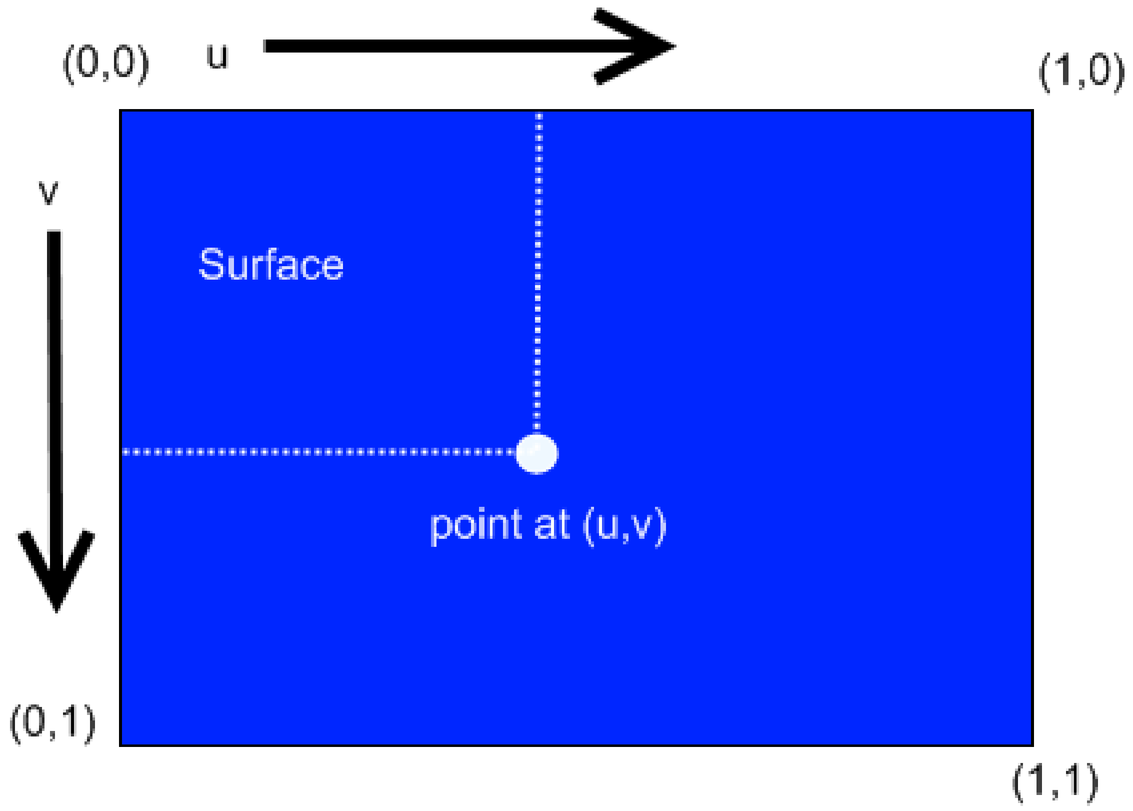
1.1.1. What is a procedural texture?

Put simply, a procedural texture is a procedure which takes input values derived from the point on the surface being textured, and returns a color value for that point on the surface. When the surface is considered as a whole (i.e. the collection of individual points), we end up with a textured surface.

Note that we can apply procedural textures in 2 or 3 dimensions, but for now, we are only considering 2 dimensions. The third dimension can often be represented by time creating an animated surface texture that changes over time.

1.1.2. Input Parameters

Since we are only dealing in 2 dimensions, our textures will mostly consist of 2D rectangles. In this case, our input points consist of 2 values, one going across the rectangle and one going down forming cartesian coordinates for any point on the rectangle.



Our procedural texture takes a `(u,v)` coordinate as an input parameter and returns a color based on the position on the surface.

The colors that we return are composed of values representing the red, green and blue components. We have an `RGBAColor` class that we use to represent a color. The 'A' part refers to the Alpha channel which indicates how transparent this color is. Transparency ranges from 0 to 1 with 0 being totally transparent and 1 being totally opaque. Colors can be specified using integers 0..255 or fractional values from 0..1. In both cases, the alpha is always specified as ranging from 0 to 1.

1.2. Implementing Textures

We start with a `Texture` interface that encapsulates the concepts so far. Our interface contains methods to return a color based on the input values.

```
public interface Texture {
    RGBAColor getColor(double u, double v);
    void getColor(double u, double v, RGBAColor value);
}
```

We have two methods because rather than keep creating `RGBAColor` instances to return from the texture, we just pass in a single `RGBAColor` instance that we re-use each time. If you consider a 100 X 100 image calls the texture 10,000 times, and since we could be chaining textures together, we could end up creating tens of thousands of `RGBAColor` instances. By re-using the same one and passing it around, we only create one and we get a 10-15% speedup.

One goal is to make the texture generation view independent. For example, we might display the texture in a small window when developing and testing it, but if we wanted to generate a texture for printing or saving, we would want to make it high resolution, and we probably don't want to display it. However, we do want to end up with the same image, but at a higher resolution. Therefore we decouple the view from the texture generation and render textures in view independent terms. We should be able to call our textures using any `(u,v)` values and it should be able to calculate the color from that point. It is this de-coupling that lets us render the texture at any size, rather than define the texture in terms of on screen pixels.

There is an abstract class called `AbstractTexture` which implements the `Texture` interface and contains a number of helper functions. The most important one is `calculateColorFromTexture` which takes the `(u,v)` coordinates, a texture and checks for null textures and puts the texture value in the target result color.

1.3. Our First Textures

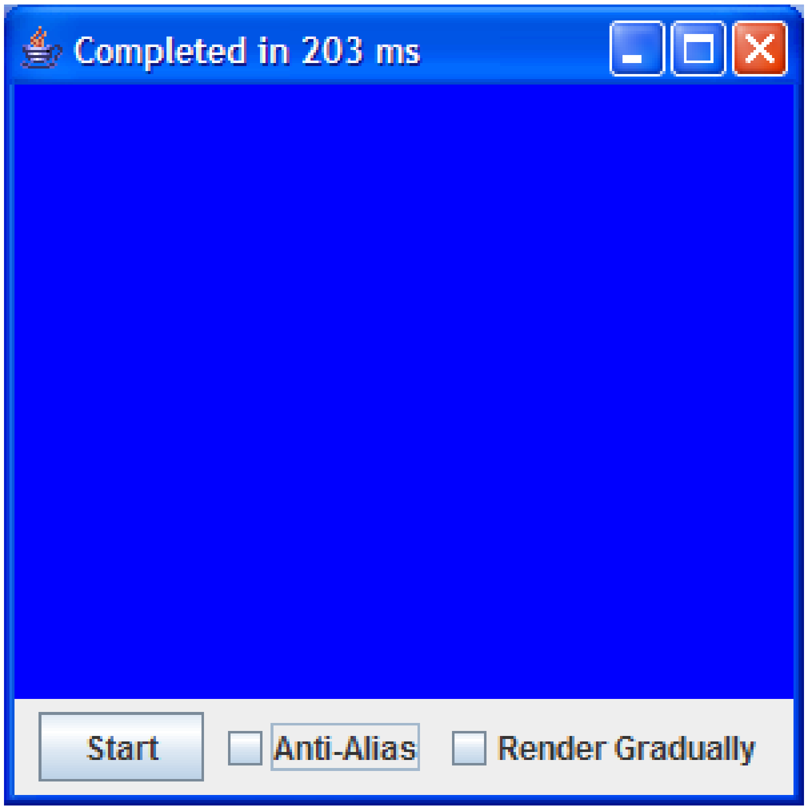
Let's take a look at writing our first simple texture which is just a solid color.

```
public class SolidBlue extends AbstractTexture {

    public void getColor(double u, double v, RGBAColor value) {
        value.setColor(0,0,255);
    }

}
```

Our simple example here just returns the same constant color no matter what we pass in. In order to view this texture, we use a class called `TextureWindow` which is a Swing window that lets you render textures in it.

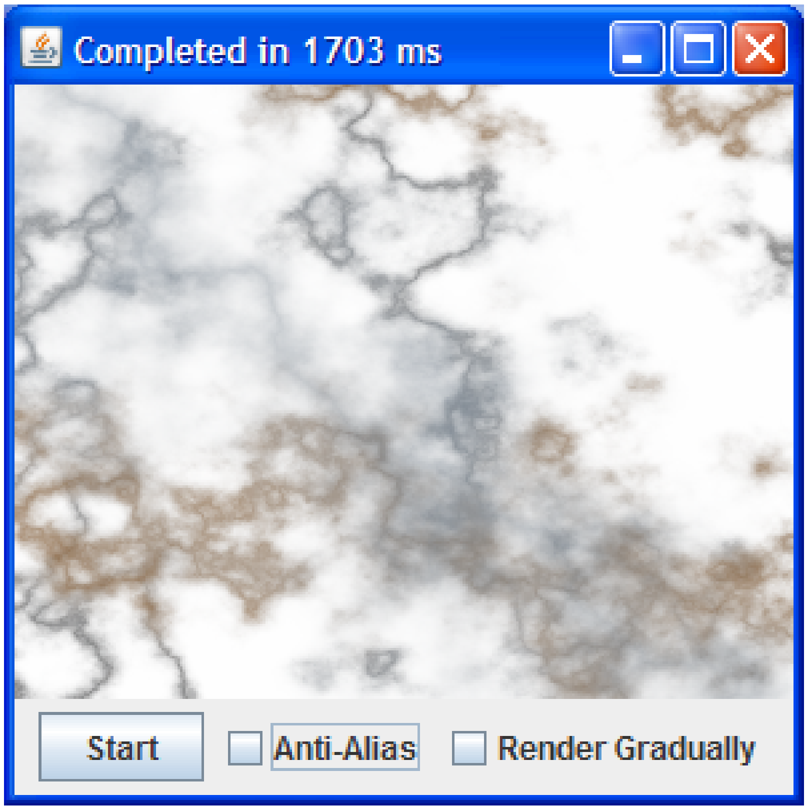
```
import org.texturemaker.gui.TextureViewer;
import org.texturemaker.textures.tester.SolidBlue;

public class SolidBlueDemo {

    public static void main(String[] args) {
        TextureViewer.show(new SolidBlue());

    }
}
```

This class can be run from the console or an IDE and should give you a Swing window which can be resized. Clicking the Start button starts the rendering process. Render Gradually produces really quick low res images which can be handy for complex textures. The anti-alias checkbox will use multiple samples per pixel. Neither of these effects can be seen in this demo because our texture is a single solid color.



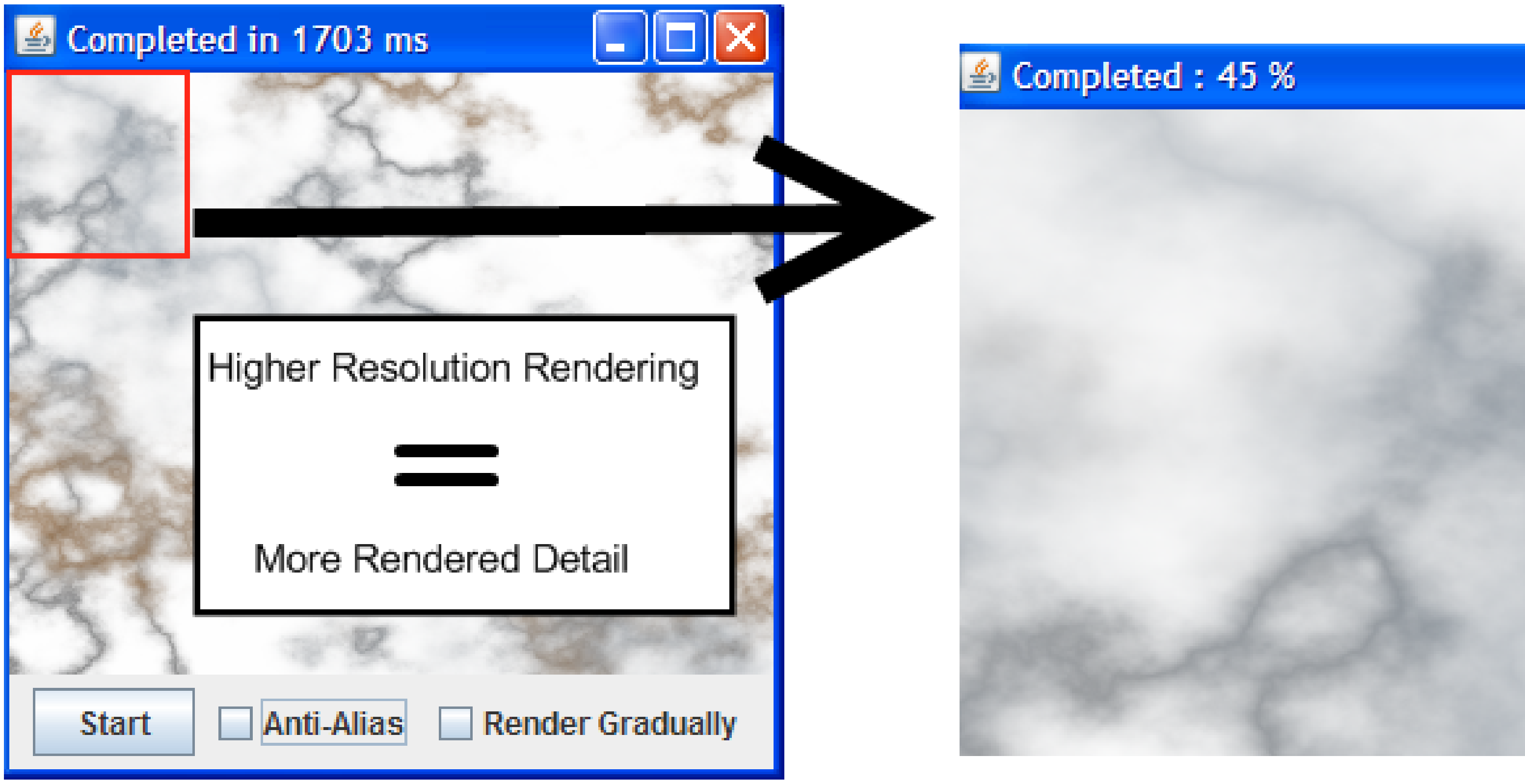
Not much really to look at here. Let's try something a bit fancier. Change our demo to use a different texture by passing a different texture to the `TextureViewer` class. Let's try the `org.texturemaker.textures.composite.ComplexMarble` class which is composed of a number of different textures to produce a nice marble effect.

```
public class SolidBlueDemo {

    public static void main(String[] args) {
        TextureViewer.show(new ComplexMarble());

    }
}
```



This is much more interesting! It also took much longer to create as it is more complex. If you resize the window, you will notice that the pattern stays the same, it scales to the window size and adds more detail to fill the space. Now when you generate a high resolution texture, you know you will get the same basic image, but with more detail.

Here is another view of the same texture. This time, we expanded the window to fill the screen causing the texture to be rendered at a higher resolution. The picture below shows the top left hand corner of the texture. If you compare to the previous full image of the marble, we can see that where we have zoomed in at a higher resolution, our procedural texture has produced more artifacts and details.



This means that if we write our textures properly, we can make them resolution independent. In which case, we can produce very high resolution versions which can be printed or used in other situations where we need high resolution detailed textures.

We use a decorator type of pattern quite often when we want to transform the inputs to and outputs from signals or textures. We can wrap another signal or texture around the existing one, and adjust the inputs or outputs to the signal or texture. For example, we might have a signal from 0 to 1 which is passed to a gradient texture. We can wrap the signal in a noisy signal so the output from the signal is jiggled around to produce not such a smooth gradient.

So now we have an easy way to test and view our textures, lets start looking at how we write them.

Chapter 2. Texture Basics

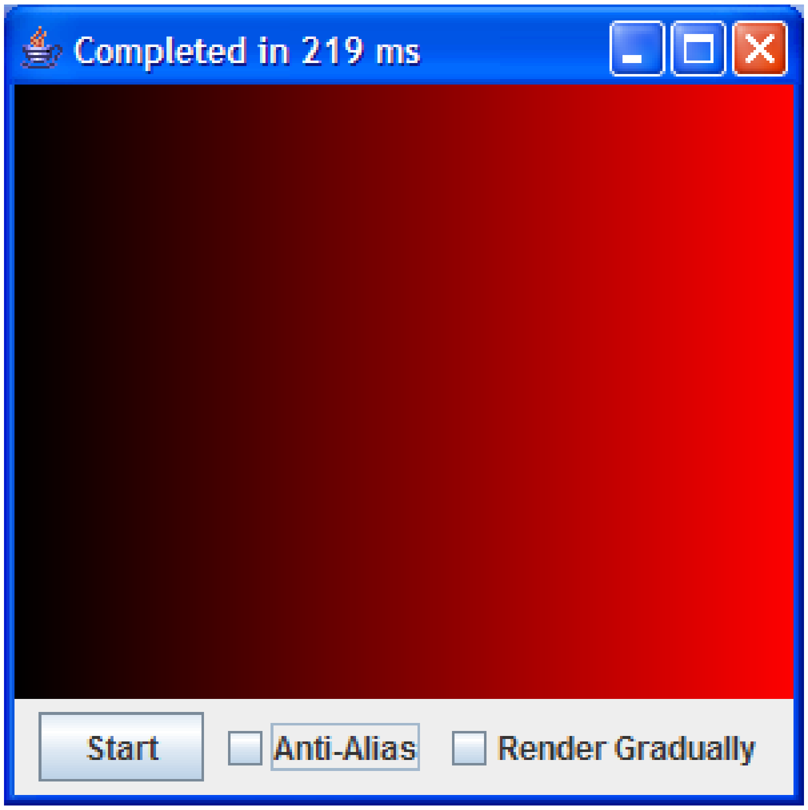
From here on, we focus mainly on the texture class itself, and assume that you are using the `TextureView` class or something similar to display the texture.

2.1. Simple Gradients

We'll start by looking at some simple gradient textures based on the u or v values, and then start looking at the `ColorGradient` class which provides us with some fundamental building blocks.

A gradient is simply an interpolation from one point to another, or in this case, one color to another. As a simple example, let's make the red component of the resulting texture color equal to the u value.

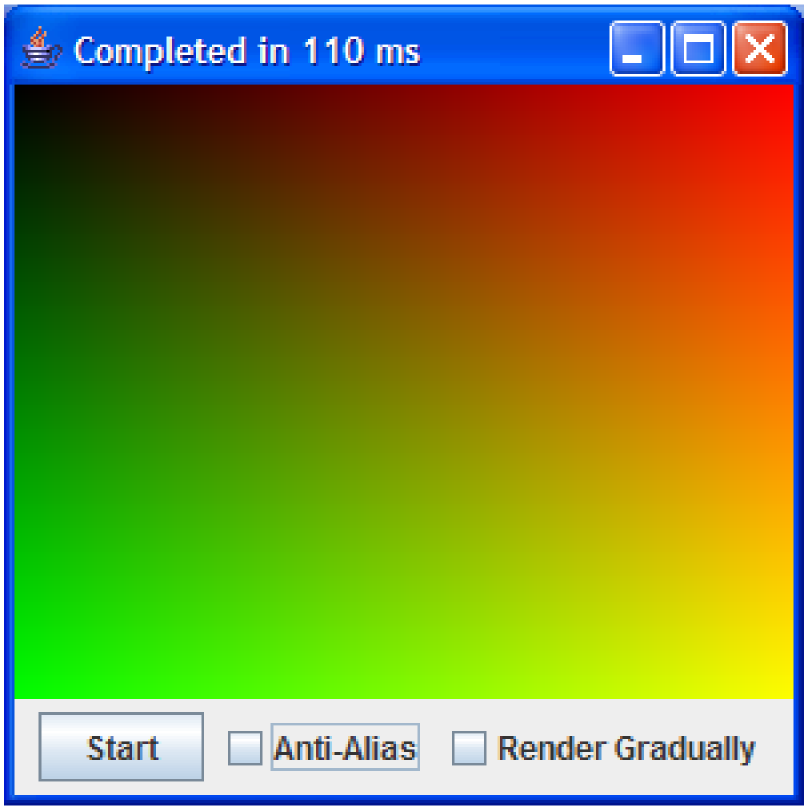
```
public class UGradient extends AbstractTexture {

    public void getColor(double u, double v, RGBAColor value) {
        value.setColor(u,0,0,1);
    }

}
```



Fairly simple, the red component of the resulting color increases as we go from left to right. This is because the U value increases in value from 0 to 1 as we go from left to right.

Let's extend this and set the green component to the v value of the input.

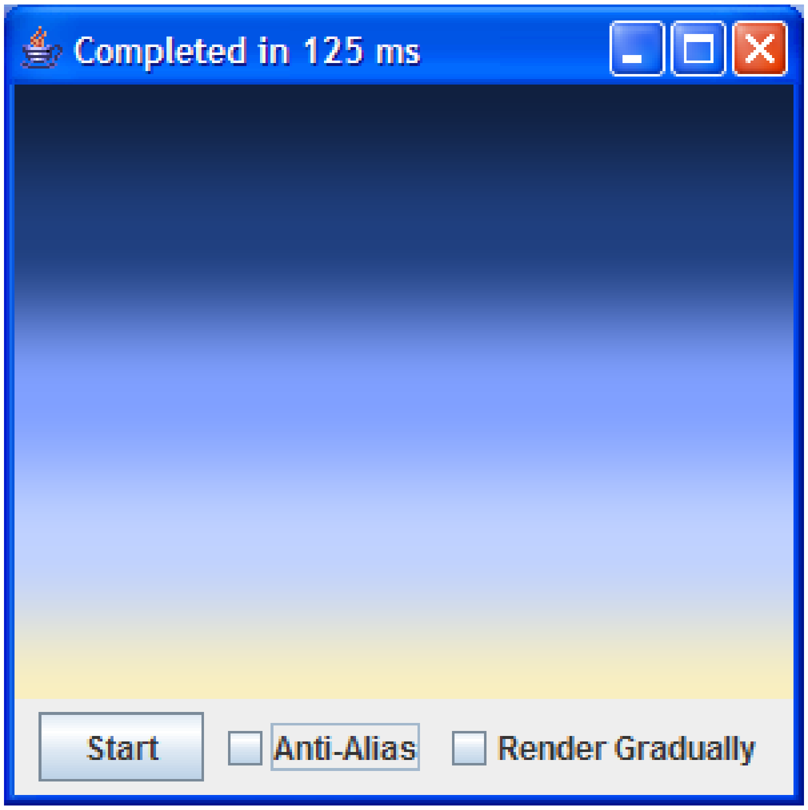
```
public class UVGradient extends AbstractTexture {

    public void getColor(double u, double v, RGBAColor value) {
        value.setColor(u,v,0,1);
    }

}
```



Nothing we wouldn't expect, but we need something a little more interesting. More complex gradients can be achieved by using the u or v value to interpolate through multiple points or colors. The `ColorGradient` class lets us easily perform this task.

```
public class SunsetTexture extends AbstractTexture {

    private ColorGradient gradient;

    public SunsetTexture() {
        gradient = new ColorGradient();
        gradient.add(new RGBAColor(16, 32, 64))
                .add(new RGBAColor(32, 64, 128))
                .add(new RGBAColor(128, 160, 255))
                .add(new RGBAColor(192, 210, 255))
                .add(new RGBAColor(250, 240, 192));
    }

    public void getColor(double u, double v, RGBAColor value) {
        gradient.interpolate(v, value);
    }
}
```

Our `ColorGradient` class takes a list of colors and when we are rendering our texture, we use those colors to calculate the resultant color based on the v value. This produces a sunset like effect in our final texture.



Gradients are fairly simple components to use and are often building blocks for more complex textures.

2.2. Cum on feel the Noise()

On key element of procedural textures is the ability to introduce controlled randomness to the texture. This is so don't have to worry about individually placing elements (clouds, dirt etc) and also so we can render larger textures without worrying about repeating elements. However, we don't want to just use a random number generator, we want some pre-determination that will allow us to calculate the same result for the same inputs.

Also, if we have two points, for which we can always calculate the same value, as we move between the two points, we want our random value move between the two points correspondingly. To achieve this, we look at Ken Perlin, a graphics guru and inventor of Perlin Noise in 1985. He has a page describing Perlin Noise at http://www.noisemachine.com/talk1/ [http://www.noisemachine.com/talk1/] .

I'll try and give a brief description of the technique of generating noise here, but Ken Perlin's page has a much more technical overview.

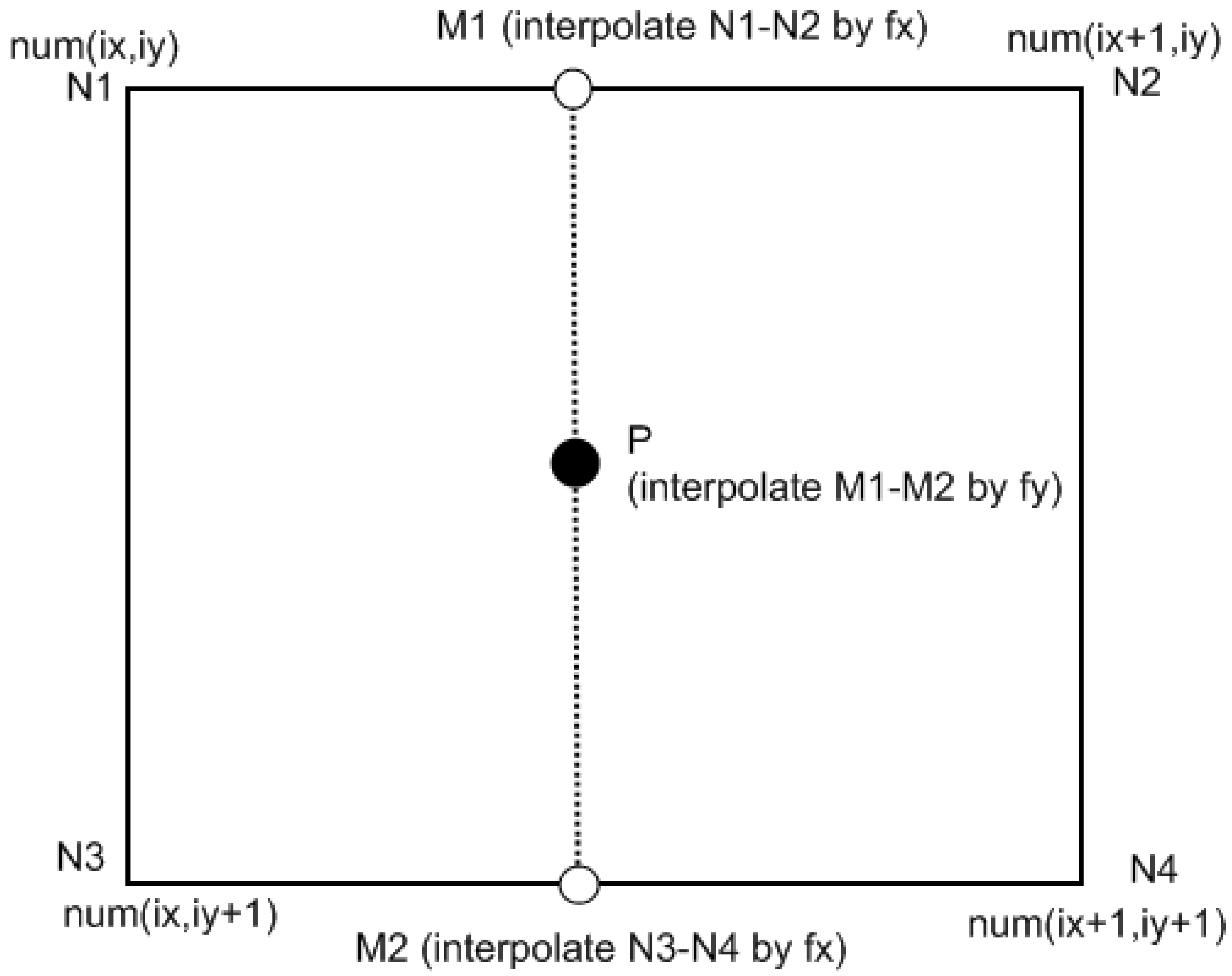
Let's start by considering 1 dimensional noise where we have a list of pure random numbers. We generate noise from a single floating point value by using the integer part of the index to look up the first value and using the integer value plus one to get the next integer value. Once we have the two integer parts, we use the floating point value to interpolate between the two points.

```
//Pseudo Code for a 1 dimensional noise function
//assumes a numbers[] array with 255 elements
function getNoise(double input) {
   int ix = floor(input); //get the integer part of the input value
   double fx = frac(input);  //get the floating point part of the in
   double v1 = numbers[ix && 255];  //get the first number
   double v2 = numbers[(ix+1) && 255];   //get the second number
   double result = (v1 * (1-fx)) +  (v2 * fx);
}
```

In practice, for the `PerlinNoise` classes used here, I use a function which combines several prime and large numbers to create a pseudo random sequence of numbers for a given integer input rather than an array of pre computed random values.

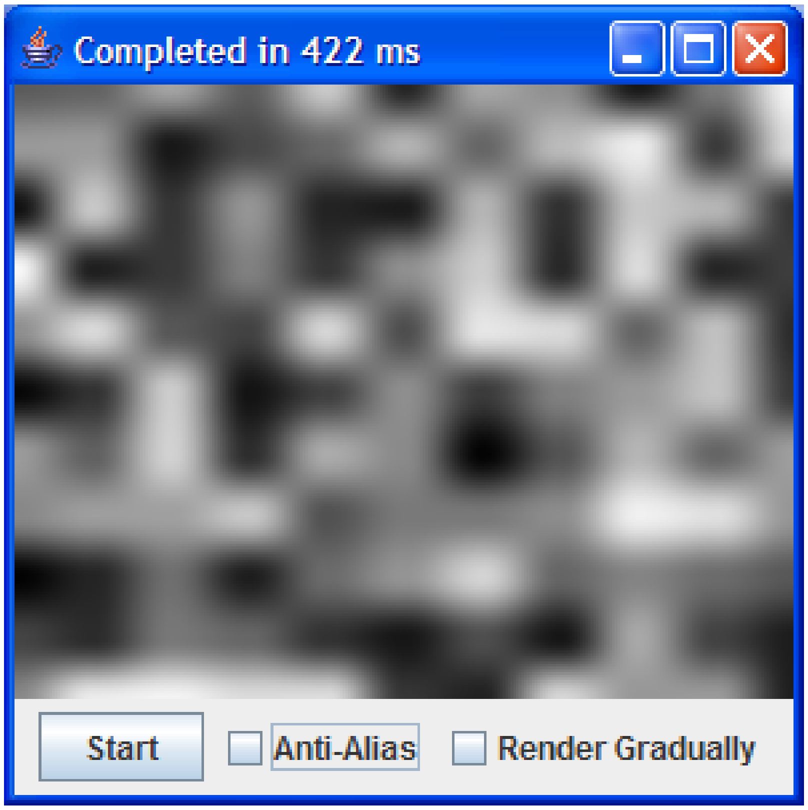
Two and Three dimensional noise can be achieved my using multiple input values and interpolating multiple times to achieve the correct value. Consider 2 dimensional noise which is calculated by `noise(x,y)` since we need 2 inputs for the 2 dimensions. We determine the 4 'random' numbers for points we use to determine the final value. Logically, we can think of this as a 4 corner square which as part of a big grid where each point in the grid is indexed by the integer part of the input values and values plus one and each points has a random value associated with it. The fractional parts determine how far along the way to the next point we are.

Given our input (x,y), we determine the integer parts (ix,iy) and the fractional part (fx,fy). Using the integer parts, we calculate the points of the square using ix,iy and ix+1,iy +1 to form the 4 corners. We label the 4 corner values as N1,N2, N3 and N4. We take these corner values and pair them together, and interpolate them using the `fx` value. This produces two mid points from N1 to N2, and N3 to N4. We then interpolate these two points based on the value of `fy` which determines how far between the two points we are. A diagram provides a much clearer example of what is going on. Note that we can also use this technique to perform bi-linear interpolation of images.


num(ix,iy)
N1
M1 (interpolate N1-N2 by fx)
num(ix+1,iy)
N2
P
(interpolate M1-M2 by fy)
N3
num(ix,iy+1)
M2 (interpolate N3-N4 by fx)
N4
num(ix+1,iy+1)

Regardless, you don't need to fully understand how the noise functions work to see what they do and how you can use them in textures. Here is a simple noise texture with the greyscale value set to the noise result for the `(u,v)` values of the texture.

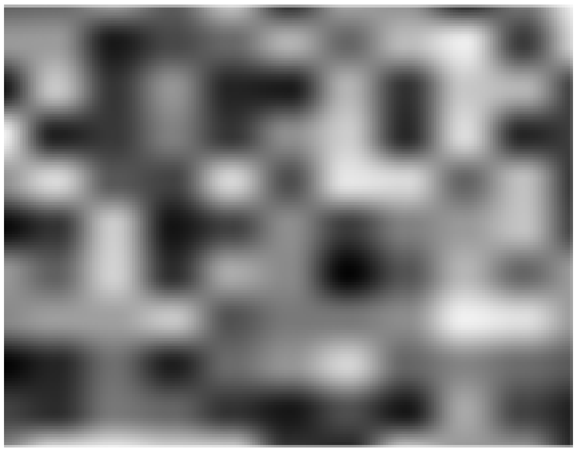
```
public class GreyNoise extends AbstractTexture {

    public void getColor(double u, double v, RGBAColor value) {
        double noiseValue = noise.noise2(u*10,v*10);
        value.setColor(noiseValue,noiseValue,noiseValue,1);
    }
}
```

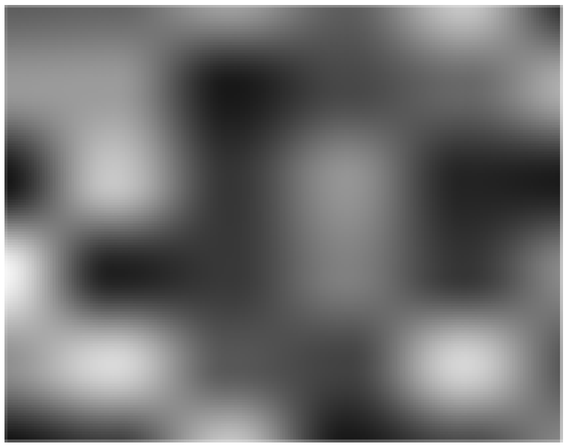


As you can see, this looks fairly blocky and unappealing. The noise itself is just an interpolation between various single points. What we can do is sum the noise at different frequencies, scale them with different amplitudes, and add them to the result to create a richer noise function. To do this, we can use the `PerlinNoise.fbmNoise()` method which takes the input parameters and the number of octaves to use. It then calculates the noise at different frequencies and sums them together.

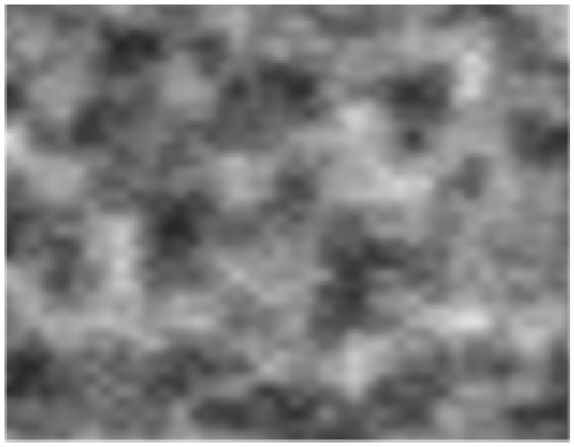
```
Noise = noise(i) + 1/2noise(2i) + 1/4noise(4i) ...
```

We can demonstrate this using multiple images for different noise coefficients and then merging them to create a final complex noise texture.


Scale = 10

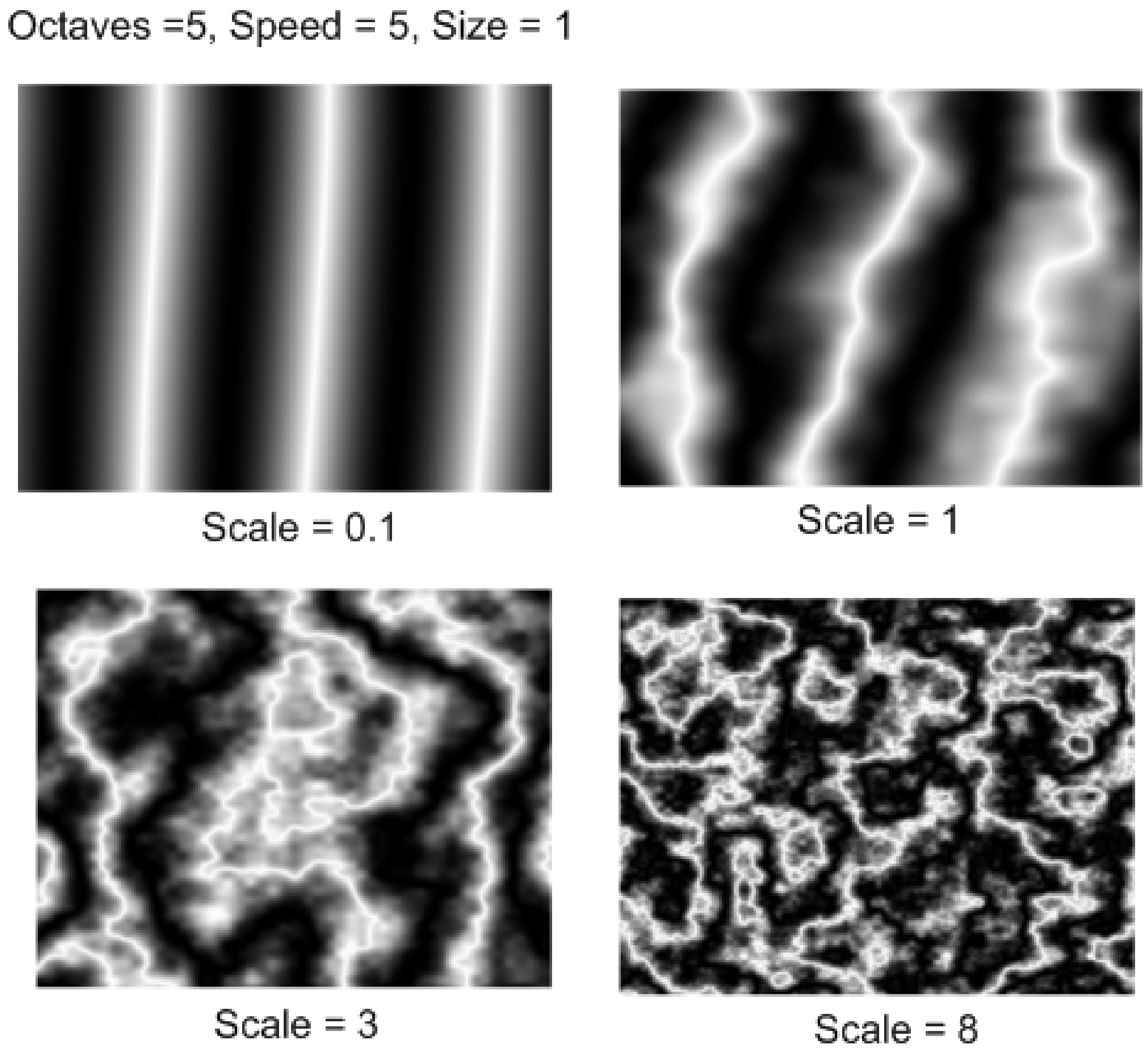

Scale = 5

Scale = 2.5


Combined Noise

2.2.1. Marbellous Textures

Another key basic texture is a marble type texture that is calculated by summing up noise functions at different frequencies and then passing the value through a sine function. In the noise class, there is a `noiseSine()` method that calculates this. The parameters for this are speed, which determines how fast the sine wave moves. Scale determines how much noise we apply to it, and octaves determine how many frequencies we use to sample the noise.



These images were generated using an anonymous texture class around a `MarbleSignal` signal to get the marble value.

```
    public static void main(String[] args) {

        Texture texture = new AbstractTexture() {

            private ChannelSignal signal = new MarbleSignal(0,0,5,1,

            public void getColor(double u, double v, RGBAColor value
                double val = signal.getValue(u, v);
                value.setColor(val,val,val,1);
            }
        };
        TextureViewer.show(texture);

    }
```

2.3. Bringing it all together

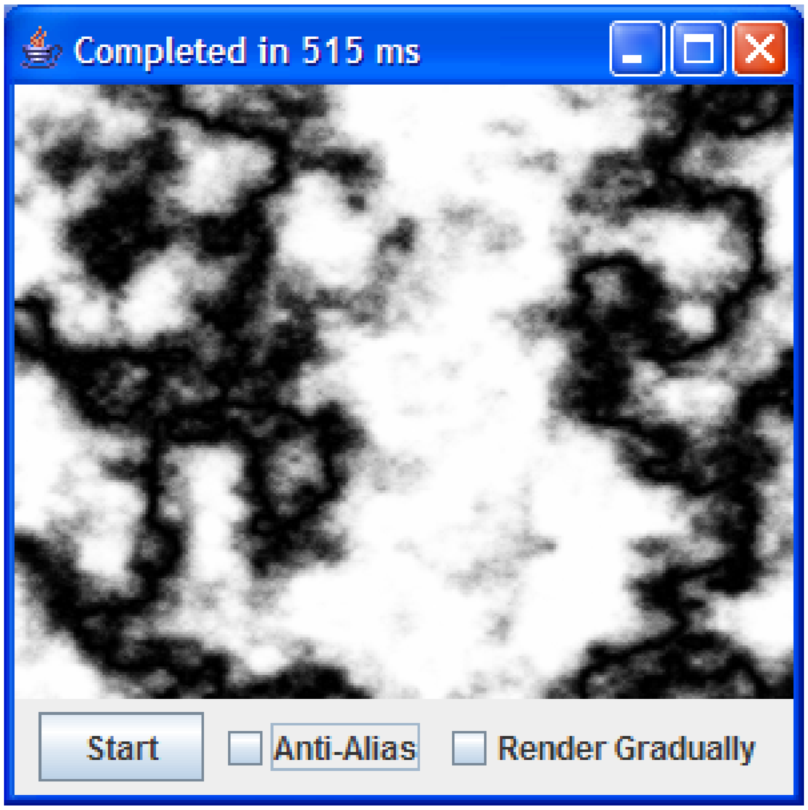
Now we have a number of textures to create, we need a way to bring them all together, to take multiple textures and combine them to form one single texture. To do this, we have several composition textures that take one or more textures as input and produce an output based on the those textures and optionally based on the `(u,v)` input parameters. Usually, we need to consider the alpha channel as it is used when compositing one texture over another. We composite textures in such a way that the visibility of the background texture is dependent on the alpha value of the texture laid over it.

The simplest method of compositing is to use the `MergedTexture` texture. This texture takes two source textures, a background and an overlay texture and merges them. It does this by calculating the point for each texture at point `(u,v)` and returns the color based on the source inputs and the overlay's alpha value. If the overlay is anyway transparent, we will see the background texture through the overlay texture.

As an example of this, lets create a composite texture that takes a `Marble` texture and merges it with a white background. In this case, the `Marble` texture only produces the veins of the marble in a specific color with varying alpha transparency. It is supposed to be laid over another texture to provide a background which is what this texture does.

```
public class MergeTest extends AbstractTexture {

    private MergedTexture mixer;

    public MergeTest() {
        Marble marble = new Marble(RGBAColor.black());
        SolidTexture background = new SolidTexture(RGBAColor.white()
        mixer = new MergedTexture(background,marble);
    }

    public void getColor(double u, double v, RGBAColor value) {
        mixer.getColor(u, v,value);
    }
}
```

You can see the results below. The marble texture always returns a solid color, but the alpha value varies between 0 and 1 depending on the value returned from the marbling noise method. When the alpha value is less than 1, then it is mixed proportionally with the background texture.



Note

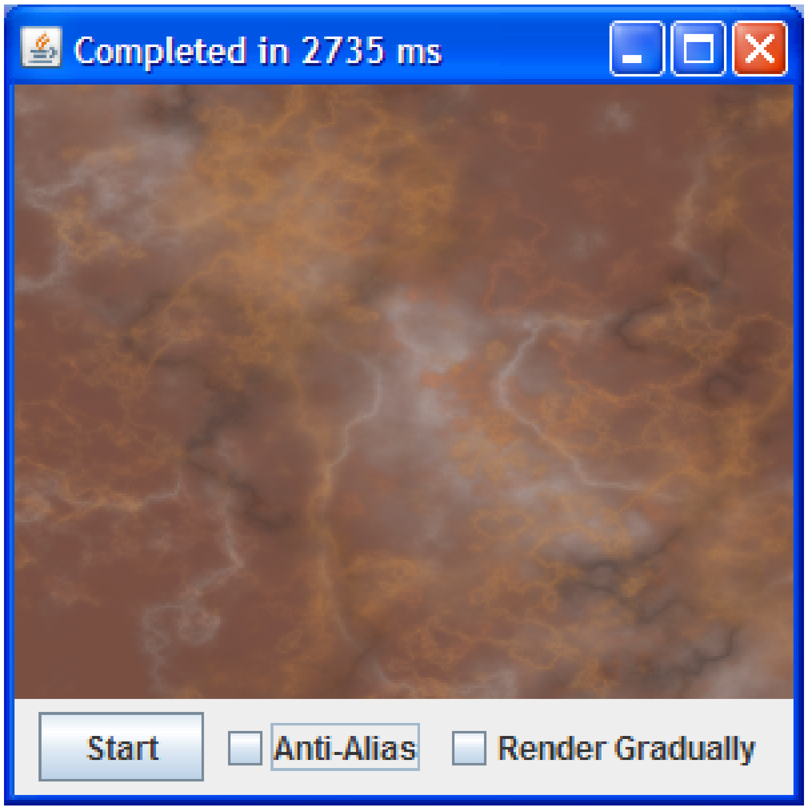
When creating textures like the basic marble which is meant to be used in other textures, it is better to return a color with varying alpha rather than mixing the white and black in the marble texture. This is so that it can be placed over other textures and let the texture show through. This lets us create more re-usable textures.

Another alternative is to take one or more input sources and return the average of the mixed results. We do this by summing up the colors and then dividing by the number of colors mixed into the result. We can do this with the `TextureMixer` class. A simpler version of this class is the `MixTexture` class which takes two textures and and optional mix level (defaults to 0.5) and the result is based on :

```
Result = (SourceA * level) + (SourceB * (1-level))
```

The `MultiMergeTexture` can be used to compose multiple texture elements onto a final texture which can then be composed onto a background. These textures are merged in the order that they are added. This next example takes a number of marble textures and puts them into a `MultiMergeTexture` and composites the multiple marble textures into one complex marble.

```
    public static void main(String[] args) {
        MultiMergeTexture mixer = new MultiMergeTexture();
        mixer.getTextures().add(new UVRotate(new Marble(new RGBAColo
        mixer.getTextures().add(new UVRotate(new Marble(new RGBAColo
        mixer.getTextures().add(new Marble(new RGBAColor(200,200,200
        mixer.getTextures().add(new Marble(new RGBAColor(0,0,0,0.4),
        mixer.getTextures().add(new UVRotate(new Marble(new RGBAColo
        mixer.getTextures().add(new Marble(new RGBAColor(255,176,80,
        TextureViewer.show(new Background(mixer,new RGBAColor(122,80
    }
```



Probably the easiest way of putting a texture onto a background is to use the `Background` texture which takes a texture and overlays it on a solid white background. You can use other colors in the constructor, but white is the default.

```
  Marble marble = new Marble(RGBAColor.black());
  mixer = new Background(marble);
```

If you want to merge textures and layers using gradients, specifically by using gradients with the alpha channel, then the next section on filtering should help you create some more interesting effects.

2.4. Filtering And Signals

The previous section looked at merging two textures based on the calculated alpha value for one of them, however, we can also use the `ChannelSignal` interface to merge textures together (as well as much more). Channel signals are similar to textures in that they take `(u,v)` input values, and return a result which is consistently the same each time it is called for that `(u,v)` value. For signals however, we only return a single double value instead of a color. That value can then be processed by a texture that is able to use the value for a number of things.

The `ChannelSignal` interface is as simple as the `Texture` interface we saw earlier. This interface is implemented in an `AbstractChannelSignal` class.

```
public interface ChannelSignal {

    public enum Channel {

        RED, GREEN, BLUE, ALPHA;
    }

    double getValue(double u, double v);
}
```

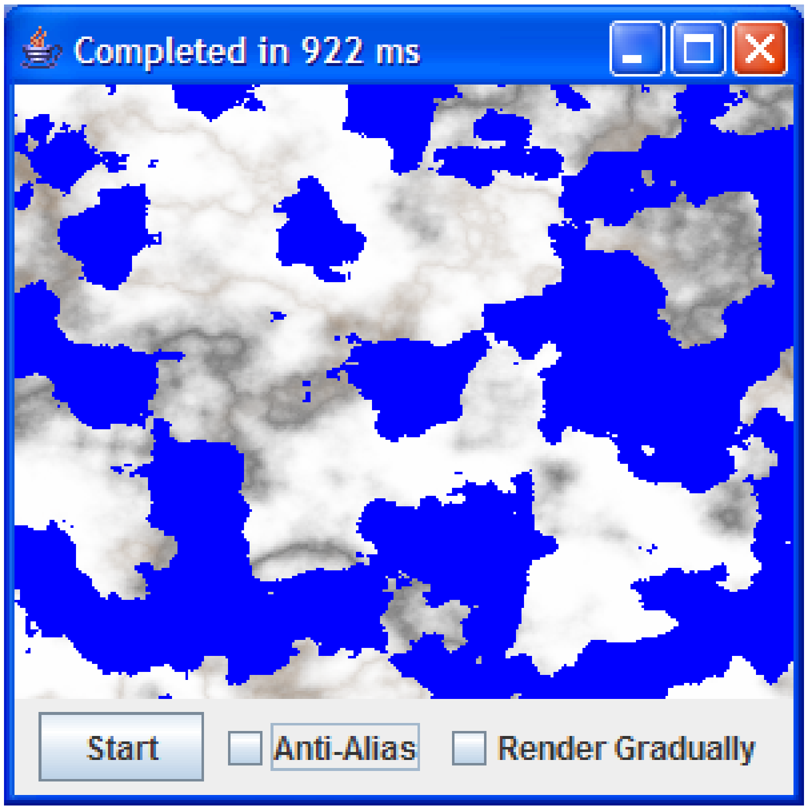
Let's jump right in and create a simple channel signal. For our example, it will just return the u value that is passed in. That means that as we process the texture across the surface from left to right, the signal will change from 0 to 1.

```
public class ChannelTester extends AbstractChannelSignal {
    public double getValue(double u, double v) {
        return u;
    }

}
```

We will test this with our `Threshold` texture which takes two source textures and a `ChannelSignal` and returns one texture if the signal is below a certain threshold value, and returns another texture if the signal is above a certain threshold value. The default switch level is 0.5, which means that since our `ChannelSignal` just returns the `u` value, the output should switch from one texture to the other around the middle of the surface texture.

```
public class ChannelMergeTest extends AbstractTexture {

    private Texture texture;

    public ChannelMergeTest() {
        Texture marble = new ComplexMarble();
        SolidTexture background = new SolidTexture(RGBAColor.blue())

        texture = new Threshold(marble, background, new ChannelTeste

    }

    public void getColor(double u, double v, RGBAColor value) {
        texture.getColor(u, v,value);
    }
}
```

This isn't a very good example, but if we change the source signal in the `ChannelTester` class to one of the built-in ones, e.g. the `NoiseSignal()` signal which returns a noise value for the given `(u,v)` values, we can create a slightly more interesting texture.



Again, still not a very good texture, but it does offer us some possibilities. By using a Noisy signal with a threshold, we can let textures poke through other textures at random points.

2.4.1. Reusing Filtering

Using Channel Signals has actually expanded the ability to create re-usable components drastically. For example, the `GradientSignalTexture` texture takes a `ColorGradient` and a `ChannelSignal`. For each `(u,v)` we calculate the output of the signal which results in a double value. This value is then used to interpolate the gradient to obtain the final texture color. This texture has been used to rewrite a number of the existing textures such as the horizontal and vertical gradients (we use a `USignal` and `VSignal` to interpolate the gradient across the `(u,v)` values). We even used it to generate a `Mandelbrot` texture using a Mandelbrot `ChannelSignal` that returns the calculated fractal value for the point on the texture.

We refactored a `MarbleSignal` signal from the original marble texture and we now use it not only in the refactored `Marble` texture but in the `Flame` texture. The difference is that one takes a signal and assigns it to the resulting color's alpha channel while the other uses the value to interpolate a gradient giving a colored marble effect. We can easily re-use our signals for different types of textures and they will probably be used more in the future to create a more pluggable system.

2.5. Transforming Inputs

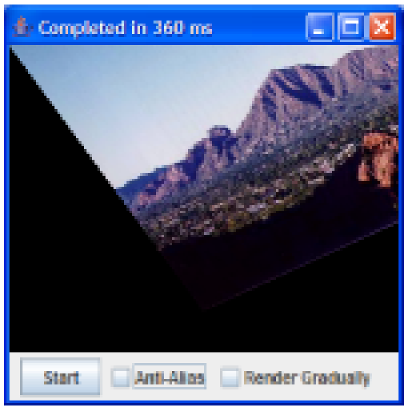
There are times when we want to modify the `(u,v)` inputs to reflect either scale, translation or rotation. Scale multiplies the `(u,v)` value by a scalar value. Rotation rotates the values around the point 0,0 and translation adds values to the `(u,v)` values. Each of these can useful effects on our textures. Scaling can allow us to zoom in or out of the texture, while rotation can give us a new angle on some textures which by default look more vertical or horizontal in nature. Translation can help us move the texture around on the surface which can be used to move more interesting parts of the texture into the center, or also reduce the visibility of patterns based on noise functions which are shared by two different textures. Also, we can apply noise to the `(u,v)` inputs to disturb the values slightly.

Most of these textures are passive in that they obtain the actual texture color value from a source texture we pass in to the transformation texture using the `(u,v)` values once they have been transformed.

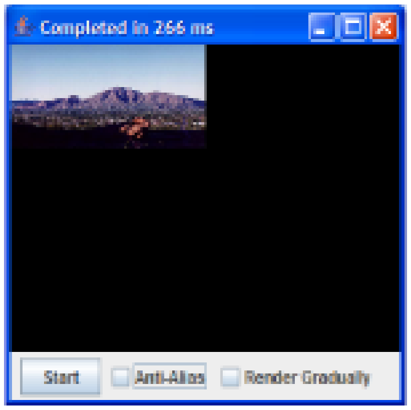
We can demonstrate the use of these textures using a pattern based texture such as the `DirtyBrick` , `Checker` , or an image based texture.

The example below demonstrates the different types of transformations that can be performed.

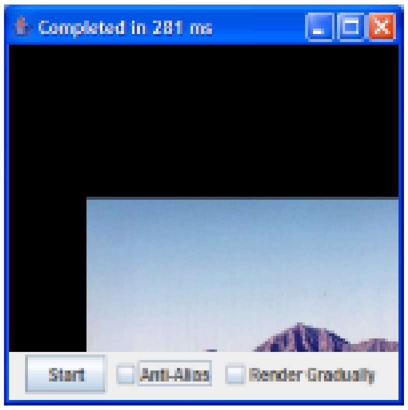
```
// Rotation
  Texture texture = new ImageTexture("c://camelback.jpg");
  TextureViewer.show(new UVRotate(texture, 45));


// Scale
  Texture texture = new ImageTexture("c://camelback.jpg");
  TextureViewer.show(new UVScale(texture, 2,3));


// Translation
  Texture texture = new ImageTexture("c://camelback.jpg");
  TextureViewer.show(new UVTranslate(texture,0.2,0.5););
```



Rotated



Scaled

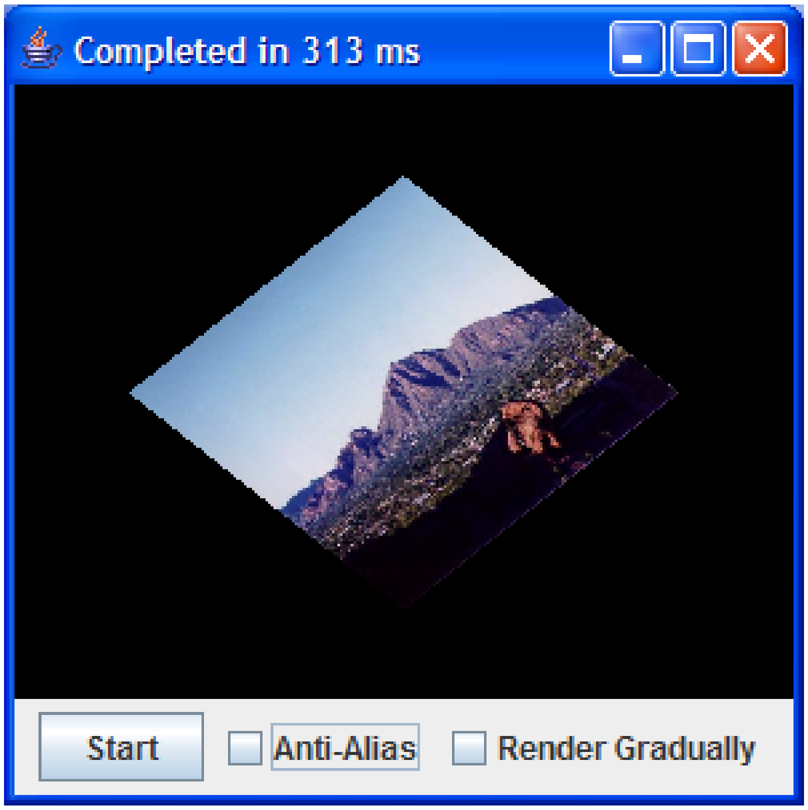


Translated

These transformations can be combined to produce more sophisticated transformations.For example, the texture is rotated around the `(0,0)` point, where you may want to transform around the center of the texture. To do this, we first translate the texture so the center is at `(0,0)` , then we rotate the texture and then translate the texture back to the center. Like all transformations, the ordering is important.

```
Texture texture = new ImageTexture("c://camelback.jpg");

texture = new UVTranslate(texture,-0.5,-0.5);
texture = new UVRotate(texture, 45);
texture = new UVScale(texture,2,2);
texture = new UVTranslate(texture,0.5,0.5);

TextureViewer.show(texture);
```

In this example, we also applied a scale to the rotated texture so we can see the whole image rotated in the middle of the texture.



Caution

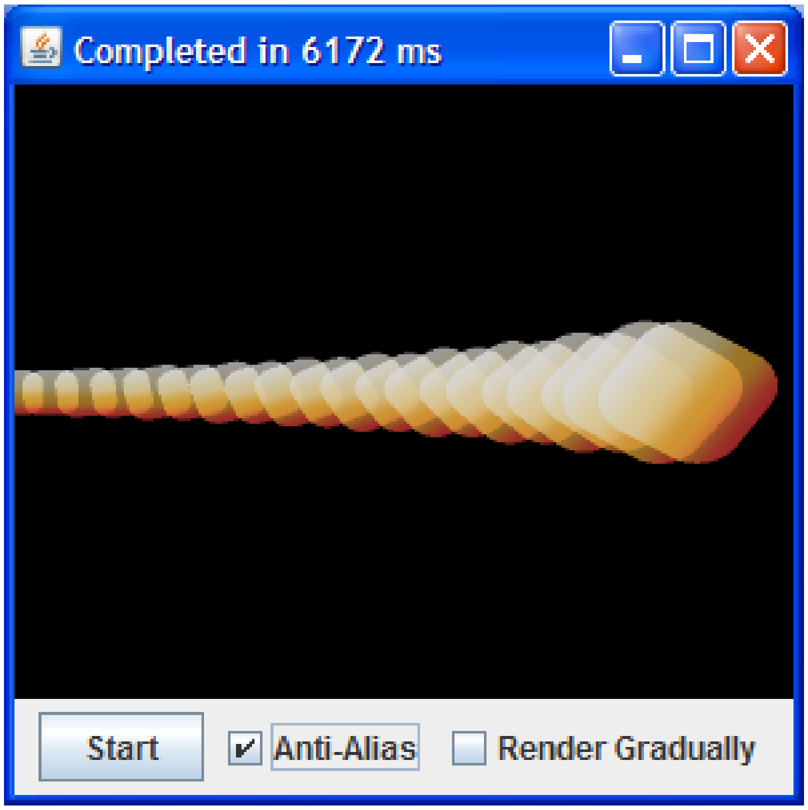
Note that transformations and rotations could lead to `(u,v)` parameters that are outside of the range 0 to 1. For this reason, you need to use the `normalize(double value)` function in the `AbstractTexture` class. This method converts the out of range value into an in-range value. For example, `normalize(-5.3)` returns 0.7 since -5.3 is 0.7 from the next lowest integer value. This method can resolve your `(u,v)` values for textures where the size of the fractional part is an issue (i.e. the `Checker` pattern).

In order to make things easier when using transformations, we have a `UVTransformer` texture class which lets you use scale,rotation and translation all in one texture. The transformations are applied in the order of scale,rotation and then translation. The transformations are automatically applied to the texture around the center as opposed to the top left corner where (0,0) is. This is similar to the multi-transformation example shown above. Below is an example of using this texture to create a series of rounded boxes with a semi-transparent gradient fill. Each box is moved to the right a little more and scaled up in size as we go from left to right.

```
public static void main(String[] args) {
  //make the color a gradient
```

```
    Texture gradient = new HorizontalGradient(ColorGradient.buildFire
    //Put an alpha filter texture over it
    gradient = new AlphaSignal(gradient, 0.75);

    //make a rounded corner that uses our gradient as the color
    RoundedCornerTexture box = new RoundedCornerTexture(gradient, 0.3

    //create the final merging texture
    MultiMergeTexture mixer = new MultiMergeTexture();

    for (int i = 0; i < 20; i++) {
      //calcualte the scale and offset for this box
      double offset = ((double) i / 22) - 0.5;
      double scale = 14 - (i / 2);
      //create the new transformer texture wrapped around the box
      UVTransformer transformer = new UVTransformer(box, offset, 0, s
      //add it to the final merged texture
      mixer.getTextures().add(transformer);
    }
    //display it on a black background.
    TextureViewer.show(new Background(mixer, RGBAColor.black()));
}
```

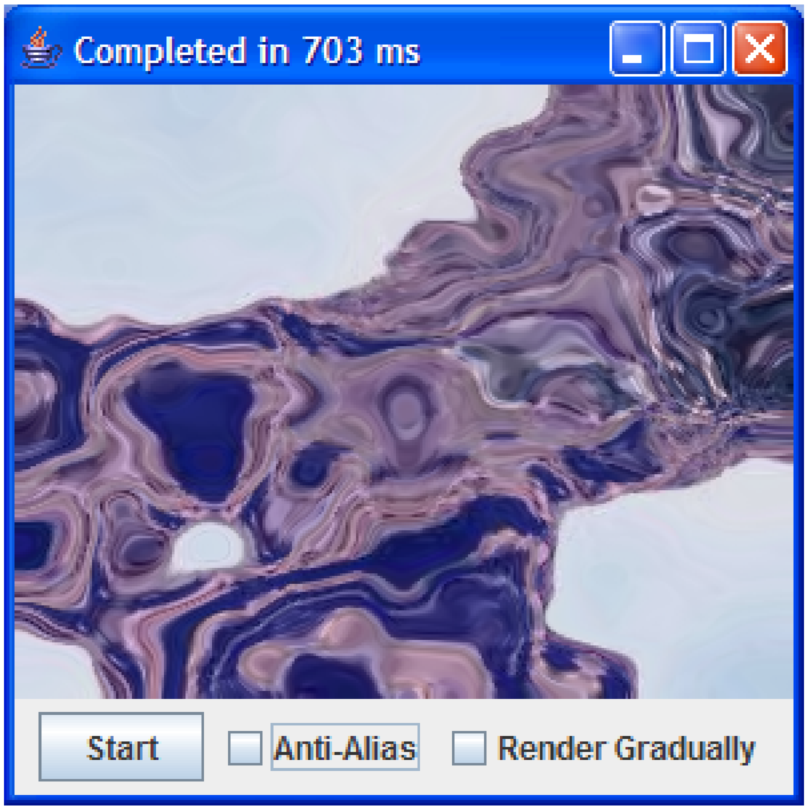
We use a lot of textures here to create an interesting effect. We also see how we can use other simple textures to modify existing ones. For example, we apply an `AlphaFilter` to the gradient so we don't have to build a new gradient with a different alpha value. Also note that we re-use the rounded corner box instance that we created to re-display the box. We just apply a different transformation to each instance. We can re-use the box, and most textures, because there is no data contained in the texture instances. They in essence act like singletons or stateless beans.



2.6. Other transformations

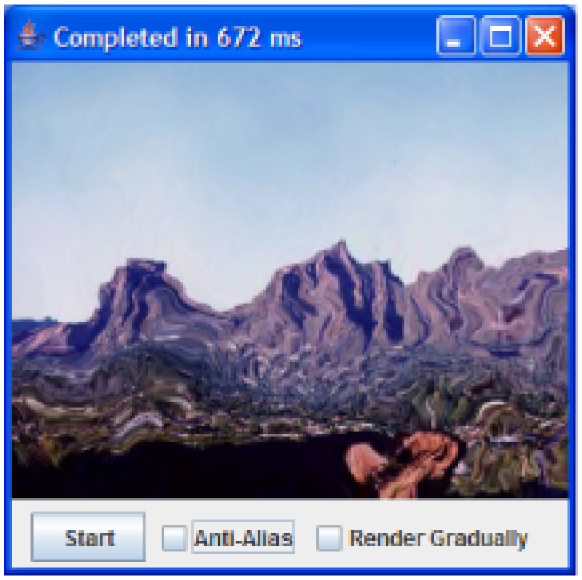
We can transform the `(u,v)` points in other non-constant ways. One is to wrap the texture in a `UVNoise` texture which takes the `(u,v)` input and generates noise from them to create 2 new `(u,v)` values.

```
Texture texture = new ImageTexture("c://camelback.jpg");

texture = new UVNoise(texture,1,5);
TextureViewer.show(texture);
```

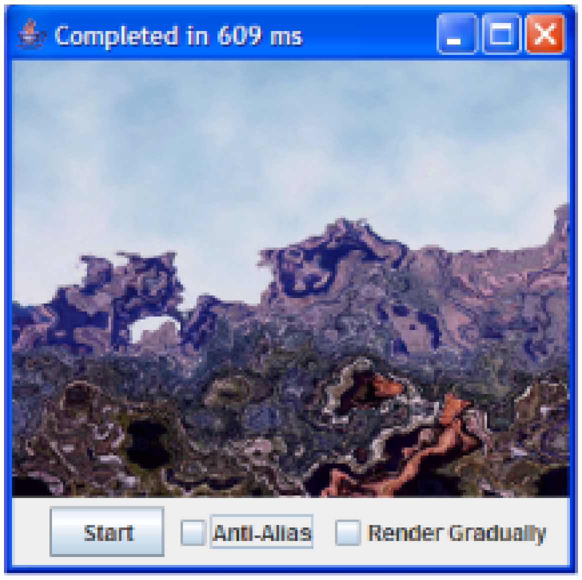


Another way is to use the `UVNoiseTranslate` texture which takes the input `(u,v)` parameters and adds a little bit of noise to them so the resultng `(u,v)` values are somewhere near where they were originally and not some pseudo random values like the previous example.

```
// Less Noise

  Texture texture = new ImageTexture("c://camelback.jpg");
  texture = new UVNoiseTranslate(texture,4,4,0.3);
  TextureViewer.show(texture);


// More Noise

  Texture texture = new ImageTexture("c://camelback.jpg");
  texture = new UVNoiseTranslate(texture,4,4,0.3);
  TextureViewer.show(texture);
```



Less Noise



More Noise

These two examples demonstrate the use of different quantities of noise to offset the `(u,v)` values. One gives a slighly perturbed image while the other is larger and looks like raindrops in a reflective puddle type of image.

2.7. Thread Safety

By default, when a texture is rendered, it is done so by spawning one or more threads. The number of threads is defaulted to the the number of processors available. This means that any signals or textures that you use to render a texture need to be thread safe because chances are that they are running in multiple threads on multiple processors. On the plus size, this will pretty much speed up the time taken to render the texture by the number of processors on the system.

As a general guide to creating thread safe textures and signals, make sure that they are immutable so that once you pass parameter values in to the signal/texture in the constructor, they cannot be changed by either the object itself or the calling code. The texture and signal objects should be stateless such that they carry no state from one call to the next. Any color objects passed to a texture are defensively copied so you can modify the object in calling code as it won't affect the texture holding a copy of it. All the textures and signals provided with the library to the best of my knowledge are thread safe with the caveat that they may use non-thread safe user defined textures or channels.

Chapter 3. More Textures

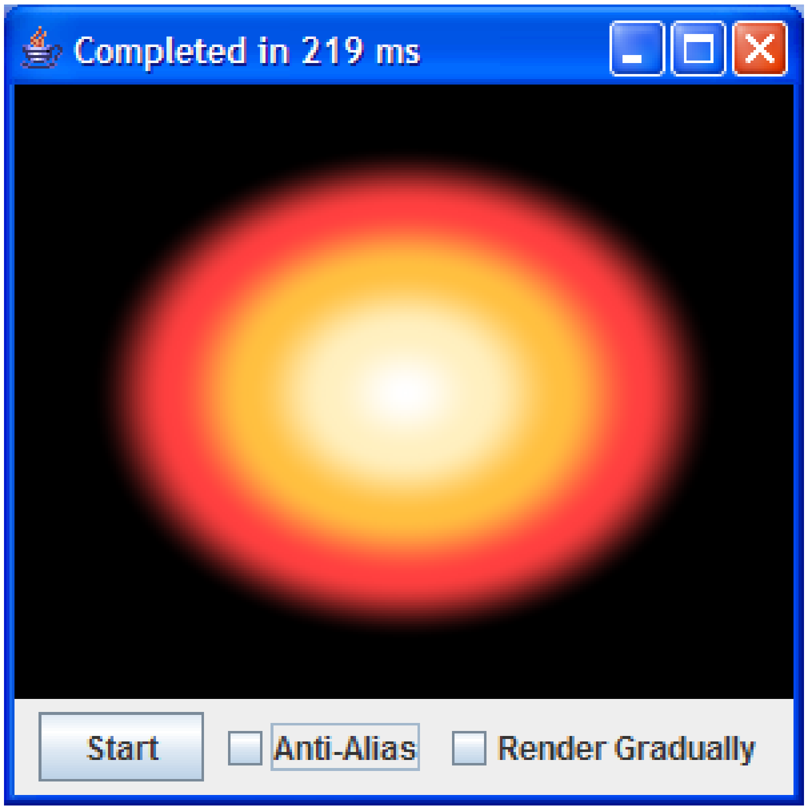
This section deals with making some more textures based on the principles illustrated in the previous chapters

3.1. Eclipse of the Sun

Let's start with a fireball texture first. First, we'll create a texture which consists of a radial gradient which goes from the center of the texture towards the edges.

```
public class Fireball extends AbstractTexture {

    private Texture texture;


    public Fireball() {
        this(buildFireGradient(), 14, 6, 0.2);
    }

    protected static ColorGradient buildFireGradient() {
        ColorGradient gradient = ColorGradient.buildFire();

        gradient.add(RGBAColor.black());
        gradient.add(RGBAColor.black());
        gradient.add(RGBAColor.black());
        gradient.add(RGBAColor.black());

        return gradient;

    }

    public Fireball(double scale, int octaves, double size) {
        this(buildFireGradient(), scale, octaves, size);
    }

    public Fireball(ColorGradient gradient, double scale, int octave
        texture = new RadialGradient(gradient);
    }

    public void getColor(double u, double v, RGBAColor value) {
        texture.getColor(u, v,value);
    }
}
```

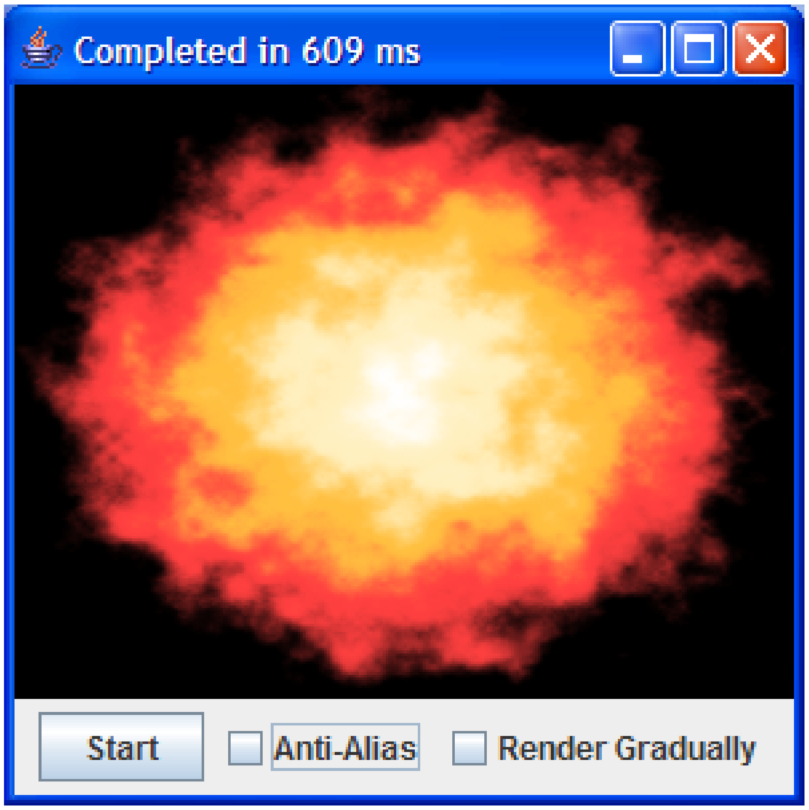
We added addition black colors on the end of the gradient since we don't want the fire ball to cover the entire texture, we want it positioned more in the center. This creates a fairly predictable texture.



Now we want to perturb the `(u,v)` points using a noise function so we create a flame effect. We do this using the `UVTranslate` texture which takes a source texture and uses it to generate the texture colors by calling it with `(u,v)` values that have been adjusted by noise. We make this change in the constructor where we decorate the radial gradient texture with a UV noise translation texture

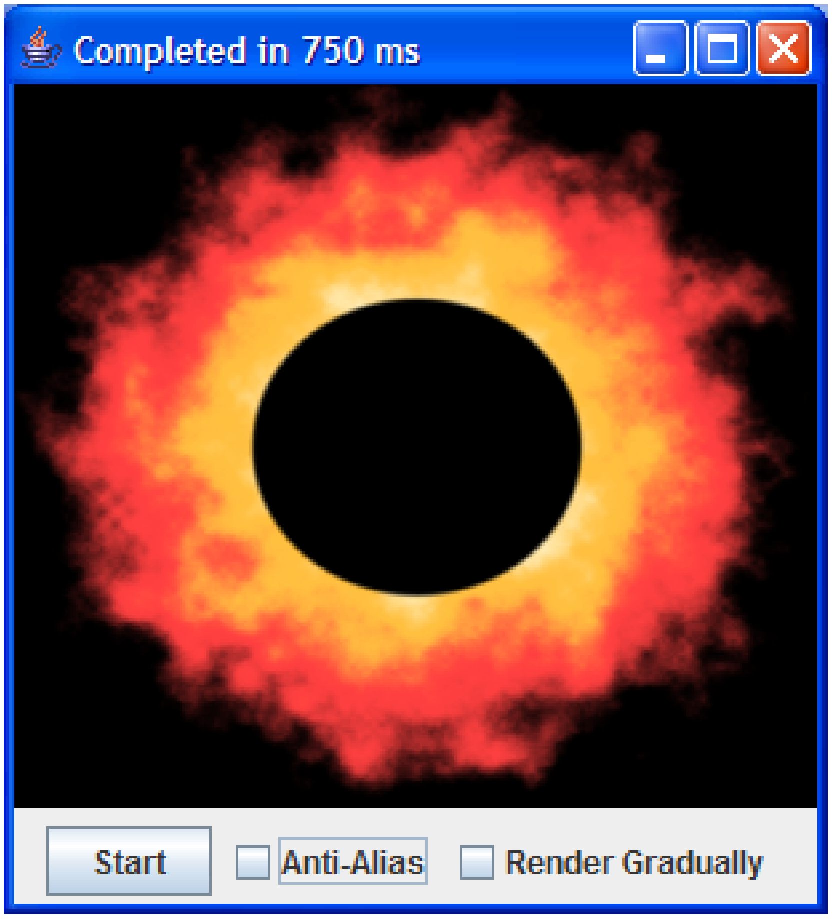
```
    public Fireball(ColorGradient gradient, double scale, int octaves
        texture = new RadialGradient(gradient);
        texture = new UVNoiseTranslate(texture,14,5,0.2);
    }
```

This gives us a rather pleasing result with a nice glowing fireball in the middle of our texture



Now let's extend the example to make the fireball look like an eclipsed sun by inserting a big dark circle in the middle. We'll make this a composite texture and create the black circle as an anonymous class. We'll then overlay the black circle texture over the fireball and use transparency to show the fire around the planet.

```
public class SunEclipse extends AbstractTexture {

    Texture texture;

    public SunEclipse() {
        Texture planet = buildPlanet();
        Texture fireball = new Fireball();
        texture = new MergedTexture(fireball, planet);
    }

    public void getColor(double u, double v, RGBAColor value) {
        texture.getColor(u, v, value);
    }

    private Texture buildPlanet() {
        //get the distance from the center of the texture,
        //if < 10, it is opaque black
        //if > 10.5 it is transparent
        //else we interpolate the transparency between 10-10.5 to giv
        //it a nice transition
        return new AbstractTexture() {

            public void getColor(double u, double v, RGBAColor value)
                u = u - 0.5;
                v = v - 0.5;
                double dist = Math.sqrt((u * u) + (v * v));
                dist = dist * 50;
                if (dist < 10) {
                    value.setColor(0, 0, 0, 1);
                } else {
                    if (dist > 10.5) {
                        value.setColor(0, 0, 0, 0);
                    } else {
                        //in between
                        double fd = dist - (int) dist;
                        fd = fd * 2;

                        value.setColor(0, 0, 0, 1 - fd);
                    }
                }
            }
        };

    }
}
```

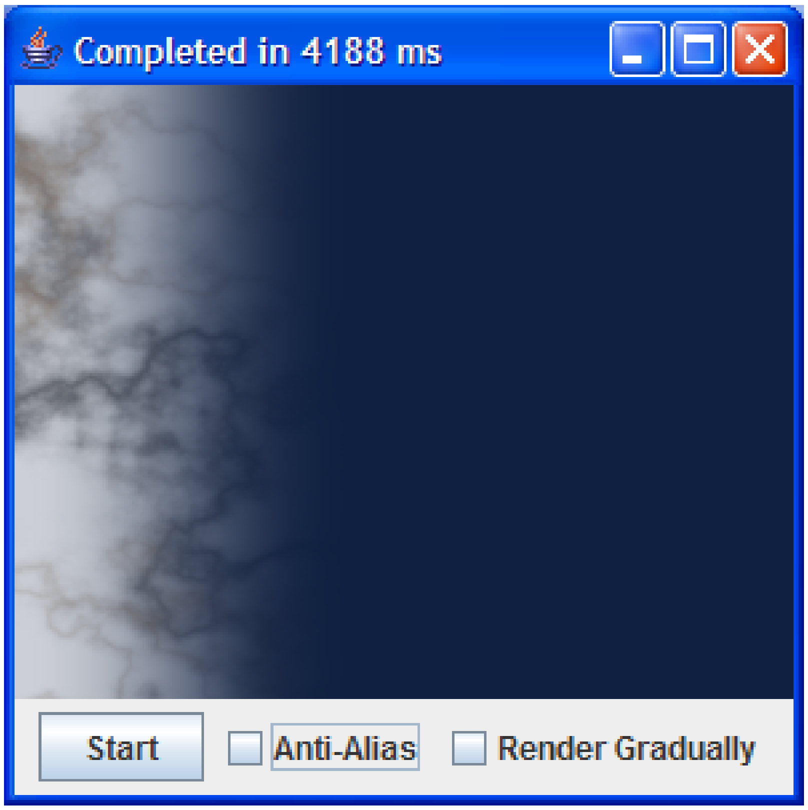


3.2. Web Background

We can use color gradients with some transparent colors to generated faded background textures. We start with a marble texture and overlay it with a color gradient that has the same color across the gradient, but different alpha values.

```
Texture marble = new ComplexMarble();

ColorGradient gradient = new ColorGradient();
gradient.add(new RGBAColor(16,32,64,0.3));
gradient.add(new RGBAColor(16,32,64,1));
gradient.add(new RGBAColor(16,32,64,1));

Texture overlay = new VerticalGradient(gradient);

Texture texture  = new MergedTexture(marble, overlay);
```

```
TextureViewer.show(texture);
```

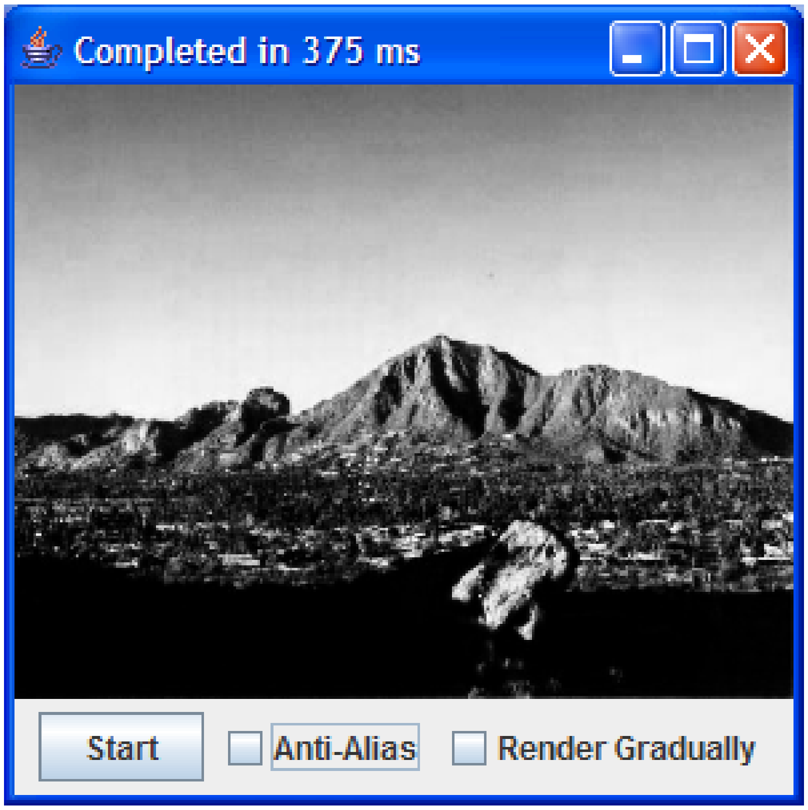


3.3. Filtering based on a source texture

We can use one texture to drive the channel signal for another texture. In this example, we use our image texture to set the alpha value for another texture which is a solid color. Because the alpha value of this texture is driven from the input signal, we get the shape of the image poking through on the final texture. This is overlaid onto a background image which shows through depending on the alpha value of the overlay.

```
public static void main(String[] args) {

    //get out image texture
    Texture image = new ImageTexture("c://camelback.jpg");

    //build a channel signal to return the red component of the i
    ChannelSignal signal = new TextureSignal(image,Channel.RED);
```

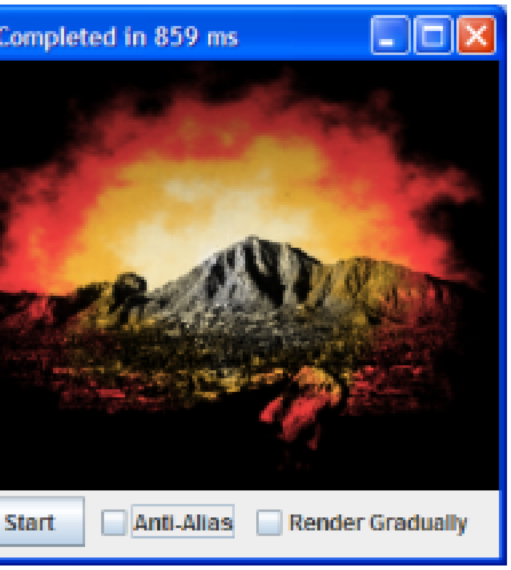
```
        //create an alpha filter based on a black background and the
        Texture overlay = new AlphaSignal(SolidTexture.blackBackgrou

        //select the background texture
        Texture background = SolidTexture.whiteBackground();

        //merge the overlay and the background
        Texture merged = new MergedTexture( background,overlay);

        //show the texture
        TextureViewer.show( merged);
    }
```



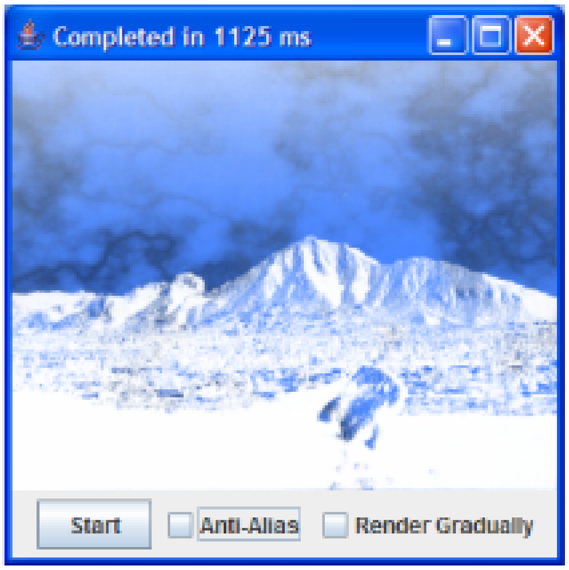
We can alter the makeup of this texture by using different textures with the alpha filter and also for the background texture.

```
// Image A
Texture background = new Fireball();
```

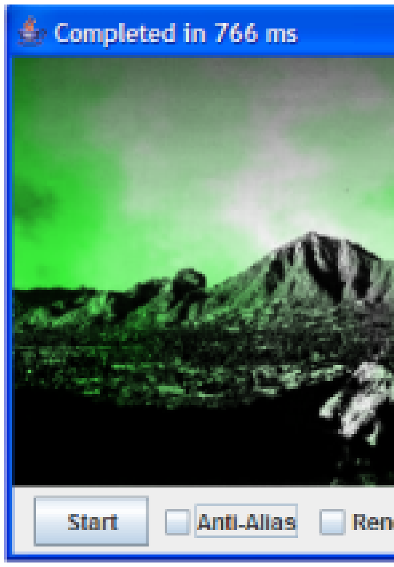
```
// Image B
Texture overlay = new AlphaSignal(SolidTexture.whiteBackground(), n
Texture background = new ColorFilter(new ComplexMarble(), new RGBAC

//Image C
Texture background = new UVTranslate(new Flame(Flame.greenFlame),1
```



A



B



C

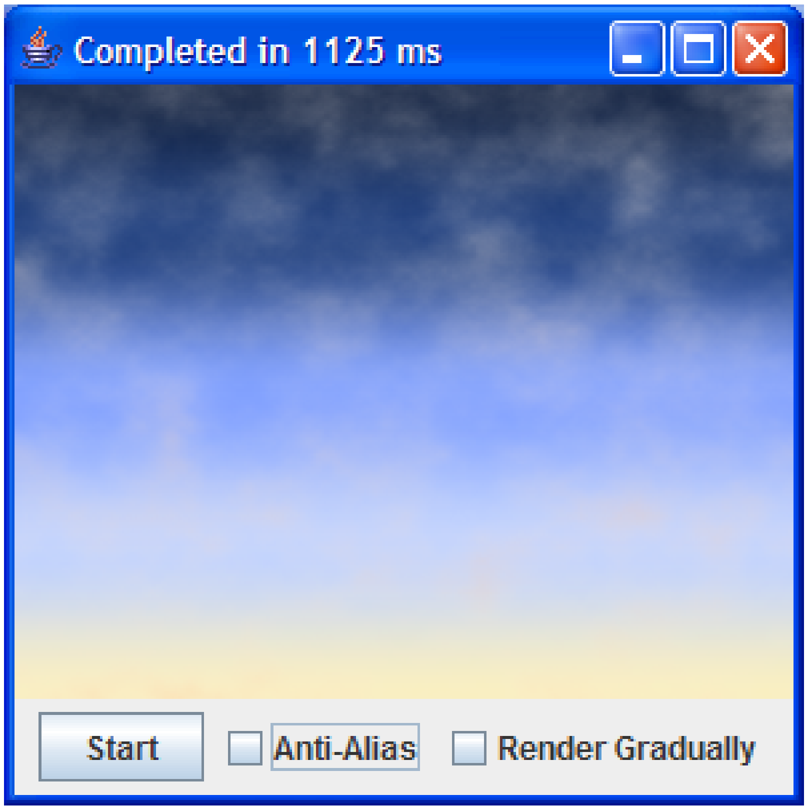
3.4. Setting Suns

Let's use our noise functions to create some cloud textures and apply it to our sunset texture we created earlier. We'll use two different sets of clouds and let them grow smaller as they appear further in the distance.

We start with two gradients that are used to color the cloud textures. These two gradients are used to create two new `HorizonalGradient` textures which are then run through a `NoisyTexture` texture. The `NoisyTexture` texture takes an input and sets the alpha value to a the result of a noise function. We take our two cloud textures and merge them into a single `clouds` texture. This texture is then laid over the top of our `SunsetTexture` we created earlier.

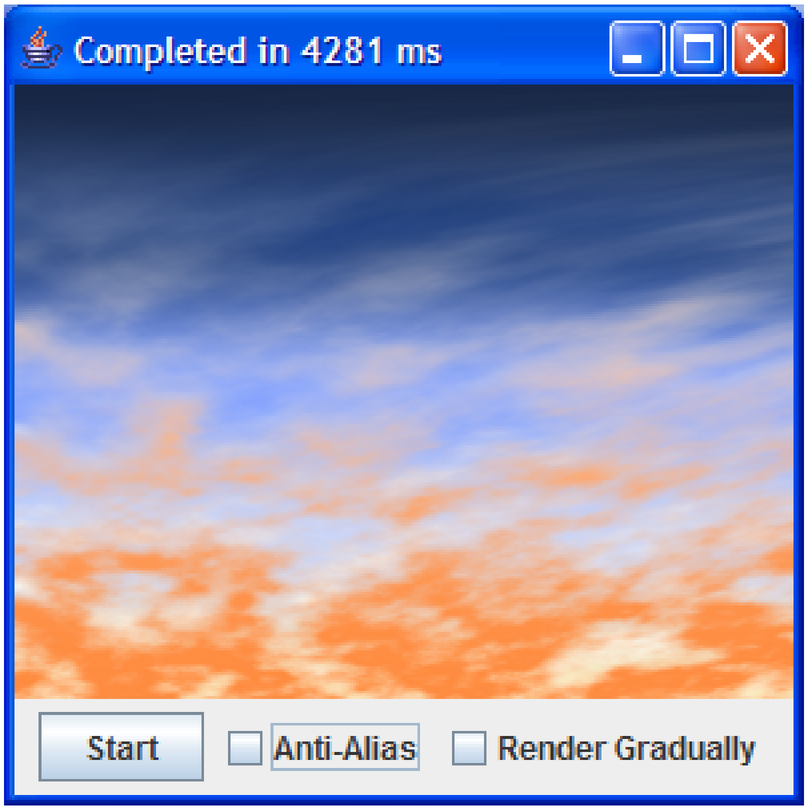
```
public class CloudTexture extends AbstractTexture {

    ColorGradient redCloudGradient;
    ColorGradient lightCloudGradient;
    MergedTexture clouds;
    NoisyTexture redCloud;
    NoisyTexture lightCloud;
    MergedTexture mixer;

    public CloudTexture() {

        redCloudGradient = new ColorGradient()
            .add(new RGBAColor(255, 240, 230))
            .add(new RGBAColor(255, 140, 64));

        lightCloudGradient = new ColorGradient()
            .add(new RGBAColor(255, 250, 240))
            .add(new RGBAColor(255, 255, 245));

        redCloud = new NoisyTexture(
            new HorizontalGradient(redCloudGradient));
        lightCloud = new NoisyTexture(
            new HorizontalGradient(lightCloudGradient));

        //merge the clouds into one texture
        clouds = new MergedTexture(lightCloud, redCloud);

        //lay the clouds over the sunset background
        mixer = new MergedTexture(new SunsetTexture(), clouds);
    }

    public void getColor(double u, double v, RGBAColor value) {
        mixer.getColor(u, v, value);
    }
```

If we look at the texture now, we get some flat clouds over the top of a distant sunset.



What we want to do is try and speed up the rate of the noise the 'further away' the cloud texture is. In this case, the clouds are further away the closer we get to the bottom of the texture (as v approaches 1). Also, the further away the clouds are, the more dense they appear. This is because when we look at clouds above us, we are looking through a thing sliver of cloud. Further away, our line of sight cuts through the cloud layer at an angle making it appear denser. To resolve this, we change the scale and size values of the noise functions the nearer to the bottom of the texture we get.

```
    public void getColor(double u, double v, RGBAColor value) {
        redCloud.setScale(2 + (v * 14));
        lightCloud.setScale(0.4 + (v * 18));

        redCloud.setSize(0.3 + (v * 1.2));
        lightCloud.setSize(0.3 + v);

        mixer.getColor(u, v, value);
    }
```



Not a thoroughly realistic sunset, but it might suffice for an image where you need some kind of background which will be out of focus in the background, especially if you are using a 3D renderer which will handle depth of field for you. This image was composed in a paint package and the background was manually blurred.

3.5. Polka, Polka, Polka

If we are looking to create a polkadot type of texture, we have to consider that we want an almost random pattern in how we position the dots, yet we need to be able to reliably get the same position back for subsequent calls regarding dot positions.

We do this in the `Polkadot` texture by scaling the `(u,v)` params by a value (in this case 8) and extracting the integer and floating point parts. This has the effect of splitting the texture into an 8 X 8 grid, and we have the integer part indicating which square it is by `(iu,iv)` and the floating point part which is `(fu,fv)` to indicate the position of the texture in that square.

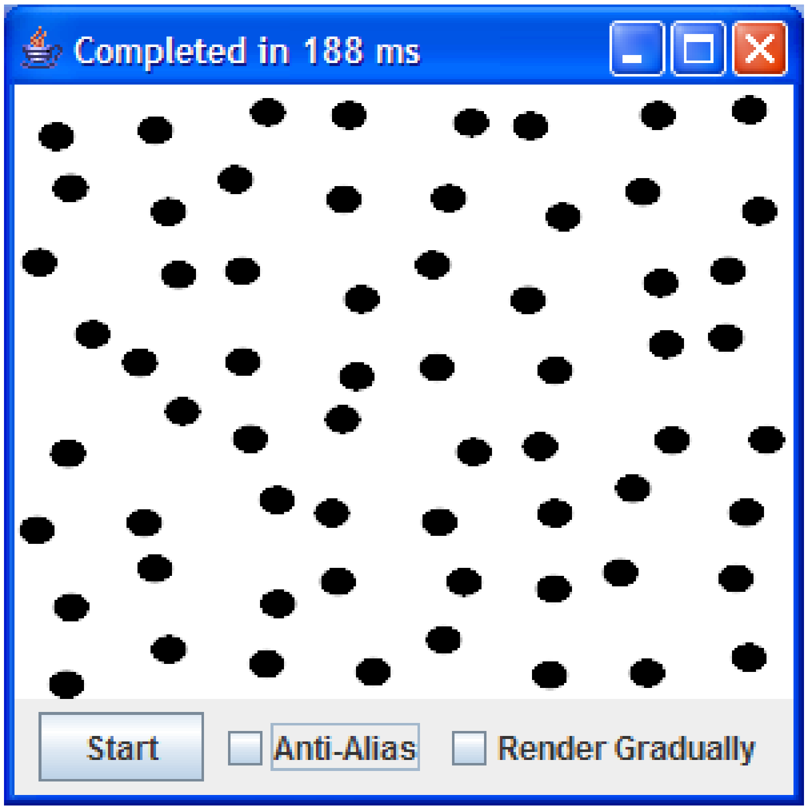
Once we have this, we can call the noise function to get the position of the center to the circle by passing in the integer components of our scaled `(u,v)` . Since `(iu,iv)` is the same for all points in the square, we can reliably get the same noise value back so we will always know the center of our dot in each square.

When we know the position of the center of the dot, we can determine how far our `(fu,fv)` point is and whether it is inside or outside of the dot. Once we know that, we can determine the final color. In this case, we provide two textures to the `Polkadot` texture one for the background and one for the dots. We also include some smoothing so that rather than switch from dot to background, we interpolate between the two at the dot edges.

```
public class Polkadot extends AbstractTexture {

    private Texture backgroundTexture;
    private Texture dotTexture;
    private double scale;
    private double dotSize;

    public Polkadot() {
        this(SolidTexture.whiteBackground(), SolidTexture.blackBackgr
    }

    public Polkadot(Texture backgroundTexture, Texture dotTexture) {
        this(backgroundTexture, dotTexture, 8, 0.4);
    }

    public Polkadot(Texture backgroundTexture, Texture dotTexture, do
        this.backgroundTexture = backgroundTexture;
        this.dotTexture = dotTexture;
        this.scale = scale;
        this.dotSize = dotSize;
    }

    public void getColor(double u, double v, RGBAColor value) {

        RGBAColor dotColor = new RGBAColor();
```

```
calculateColorFromTexture(u, v, backgroundTexture, value);
calculateColorFromTexture(u, v, dotTexture, dotColor);

//make it an 8 X 8 grid
u = u * 8;
v = v * 8;


//get the integer and fractional parts
int iu = (int) u;
int iv = (int) v;
double fu = u - iu;
double fv = v - iv;

//iu,iv determines which square we are on
//noise(iu,iv) determines the center of the
//dot for that square.
double du = noise.noise2(iu, iv);
double dv = noise.noise2(iu + 565, iv + 273);

//center the dot in the center are of the square
//we have to place the dot at least dotSize/2 away
//from the edge of the square.
du = du * (1 - dotSize);
dv = dv * (1 - dotSize);
du = du + (dotSize * 0.5);
dv = dv + (dotSize * 0.5);

//du,dv = location of dot center in this square
du = du - fu;
dv = dv - fv;

//calculate the distance of fu,fv from the center
//of the dot
double dist = Math.sqrt((du * du) + (dv * dv));

//if the distance is less than dotsize/2 then
//it is in the dot
if (dist < (dotSize * 0.5)) {
    //check if it is on the dot edge
    if (dist < (dotSize * 0.45)) {
        //if not, just merge the dot color
        value.merge(dotColor);
    } else {
```

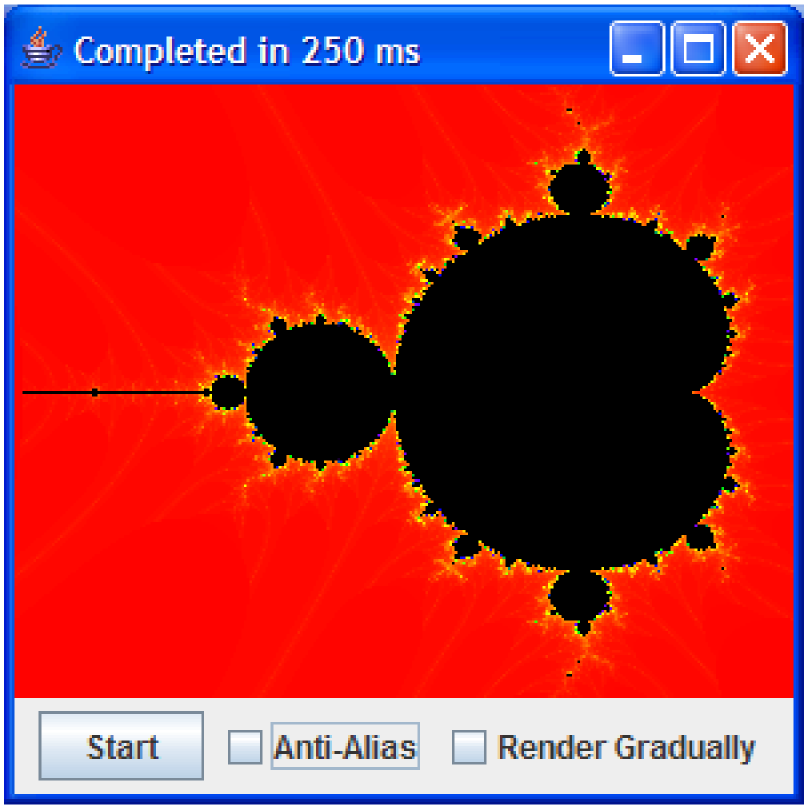
```
                        //this point is on the dot edge so we need
                        //to smooth it.

                        //dist is in the range dotSize * (0.45 to 0.5)
                        dist = dist - 0.45;
                        double alpha = dotColor.getAlpha();

                        alpha = alpha * (dist * 20);
                        dotColor.setAlpha(1 - alpha);
                        value.merge(dotColor);
                    }

                }
            }
        }
```



3.6. Fractal Fun

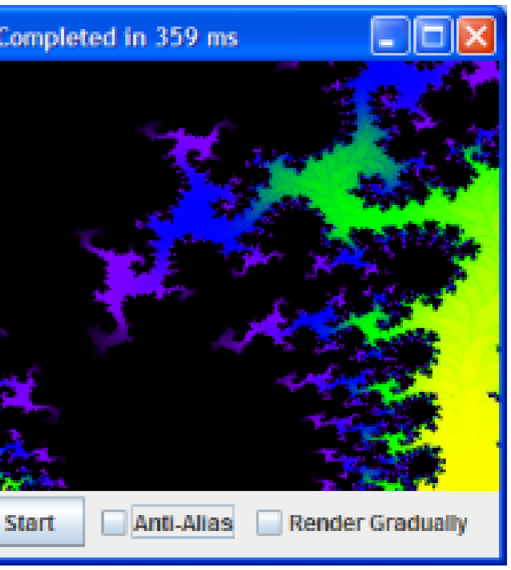
We have a number of textures which take a `ChannelSignal` and use it to interpolate a `ColorGradient` . Typically, these are noise based values, but we can also use other sources. In this example, we will use a Mandelbrot generator in a `ChannelSignal` to give us fractal values.

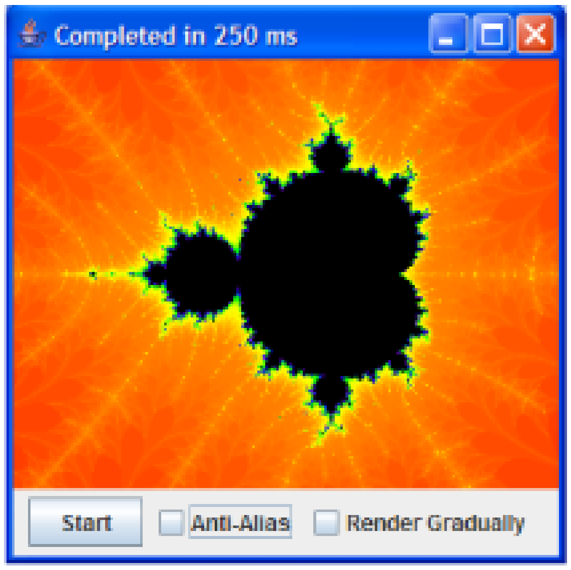
```
public class Mandelbrot extends AbstractTexture {

    private ColorGradient gradient;
    private MandelbrotSignal signal;

    public Mandelbrot(ColorGradient gradient, double startX, double s
        signal = new MandelbrotSignal(startX,startY,endX,endY);
        this.gradient = gradient;
    }

    public void getColor(double u, double v, RGBAColor value) {
        double val = calculateSignalFromFilter(u, v, signal);
        gradient.interpolate(val,value);

    }
}
```

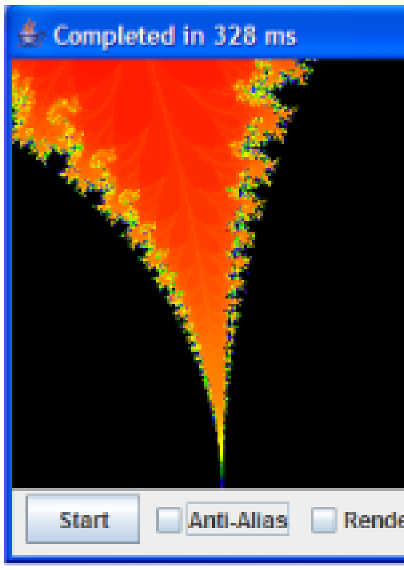


As part of the constructor, the `Mandelbrot` texture will take start and end parameters for the range to generate. Unless you have a mandelbrot viewer, picking values can be hit and miss so here are some examples.

```
Texture texture = new Mandelbrot(0.35, 0.35, 0.3507, 0.3507);
Texture texture = new Mandelbrot(-1.487,-0.006950,-1.46944,0.007
Texture texture = new Mandelbrot(-0.86429,-0.2578125,-0.5671,-0.
Texture texture = new Mandelbrot(-0.75401,-0.23191,-0.68671,-0.1
Texture texture = new Mandelbrot(-0.745725,-0.215732,-0.728179,-
```

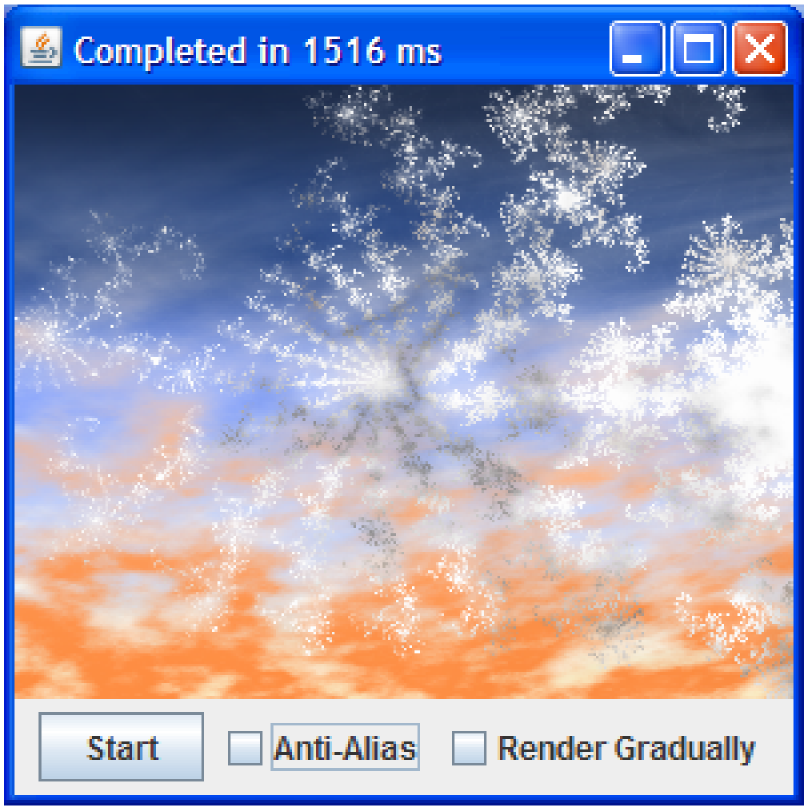






This is another example of wrapping the texture logic in a Channel Signal that can be re-used in other textures. We can use the channel signal to drive the alpha value for merging two textures.

```
public static void main(String[] args) {
    Texture texture = new AlphaSignal(new ComplexMarble(),new Ma
    MergedTexture text = new MergedTexture(new CloudTexture(), t
    TextureViewer.show(text);
}
```



3.7. Where did my texture go?

This texture involves using `NoiseSignal` textures to toggle between different textures to create a camouflage effect. Let's start with 3 simple colors to use in our texture and two `SmoothThreshold` textures switching between two solid color textures. The `SmoothThreshold` texture is the same as the `Threshold` texture except that when the texture switches, it does so smoothly instead of toggling instantly between the two.

```
SolidTexture green = new SolidTexture(new RGBAColor(119,139,111));
SolidTexture brown = new SolidTexture(new RGBAColor(119,110,100));
SolidTexture beige = new SolidTexture(new RGBAColor(166,150,116));

Texture cam1 = new SmoothThreshold(green,beige, new NoiseSignal(4,
Texture cam2 = new SmoothThreshold(beige,brown, new NoiseSignal(4,
```

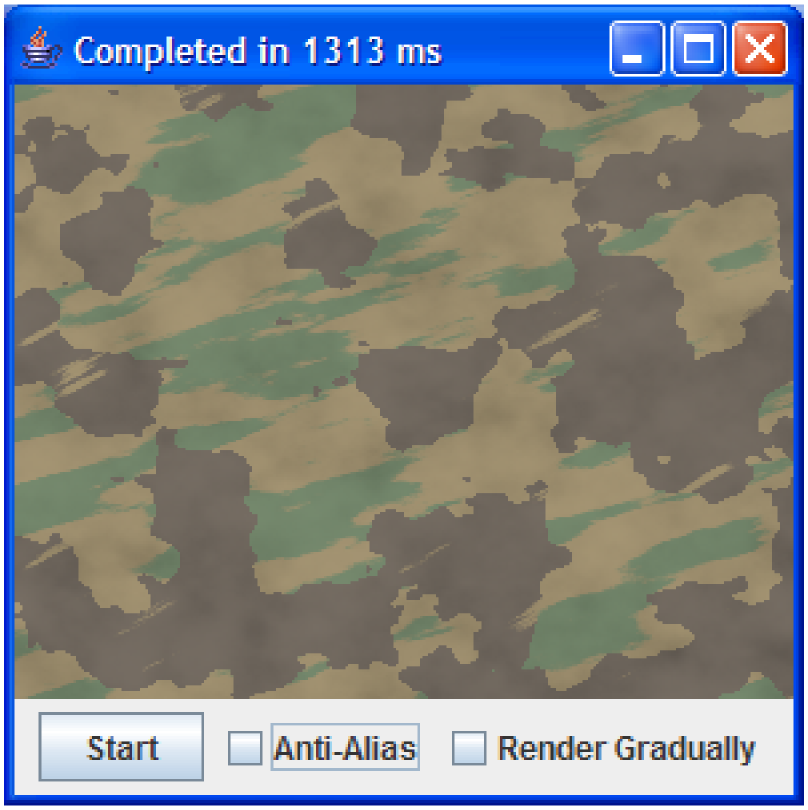
Now let's take these two textures cam1 and cam2 and toggle between the two. Before we do that let's apply a rotation and scale to the UV values before calculating the texture. This lets us scale and rotate the UV value per texture so we get skewed and rotated results.

```
        Texture rotatedCam1 = new UVRotate(new UVScale(cam1,1,4),20)
        Texture rotatedCam2 = new UVRotate(new UVScale(cam2,1,6),34)
        texture = new SmoothThreshold(rotatedCam1, rotatedCam2, new N

        texture = new Dirty(texture,RGBAColor.black(),0.32);
```

We take our rotated and scaled textures and create a new Threshold texture which will merge them based on another noise signal.

Finally we add a Dirty texture to the final texture. This creates colored noise (black in this case) and applies it to the source texture with a given strength. In this case, we are using it to just dirty up the final texture.

```
public class Camouflage extends AbstractTexture {

    private Texture texture;

    public Camouflage() {
        SolidTexture green = new SolidTexture(new RGBAColor(119,139,1
        SolidTexture brown = new SolidTexture(new RGBAColor(119,110,1
        SolidTexture beige = new SolidTexture(new RGBAColor(166,150,1

        Texture cam1 = new SmoothThreshold(green,beige, new NoiseSign
        Texture cam2 = new SmoothThreshold(beige,brown, new NoiseSign


        Texture rotatedCam1 = new UVRotate(new UVScale(cam1,1,4),20)
        Texture rotatedCam2 = new UVRotate(new UVScale(cam2,1,6),34)
        texture = new SmoothThreshold(rotatedCam1, rotatedCam2, new N

        texture = new Dirty(texture,RGBAColor.black(),0.32);

    }

    public void getColor(double u, double v, RGBAColor value) {
        texture. getColor(u*1.3, v*1.3,value);
    }
}
```



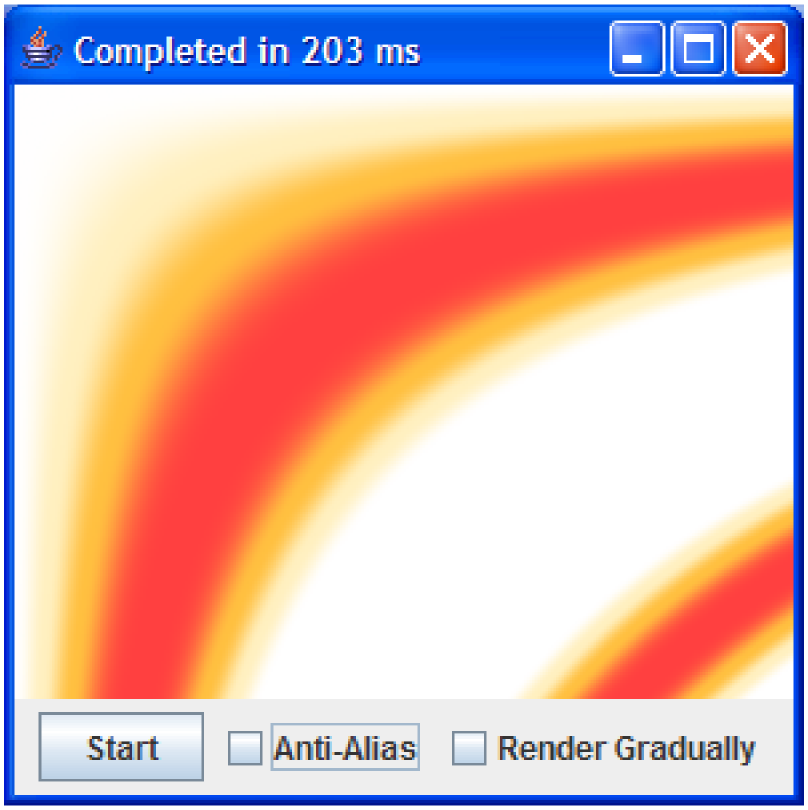
You could create a huge image of this texture and use it in 3D modelling applications to texture map tanks and so on.

3.8. Using Anonymous Classes

There are times when you might not need to create a brand new Signal or Texture class where you can simple use anonymous classes to implement the functionality. Doing so is easy with our single method interfaces.

```
//create anonymous signal
ChannelSignal signal = new ChannelSignal() {

    public double getValue(double u, double v) {
       return Math.sin(u * v * 10);
    }
};
```

```
Texture texture = new GradientSignalTexture(signal,ColorGradient.bu

TextureViewer.show( texture);
```



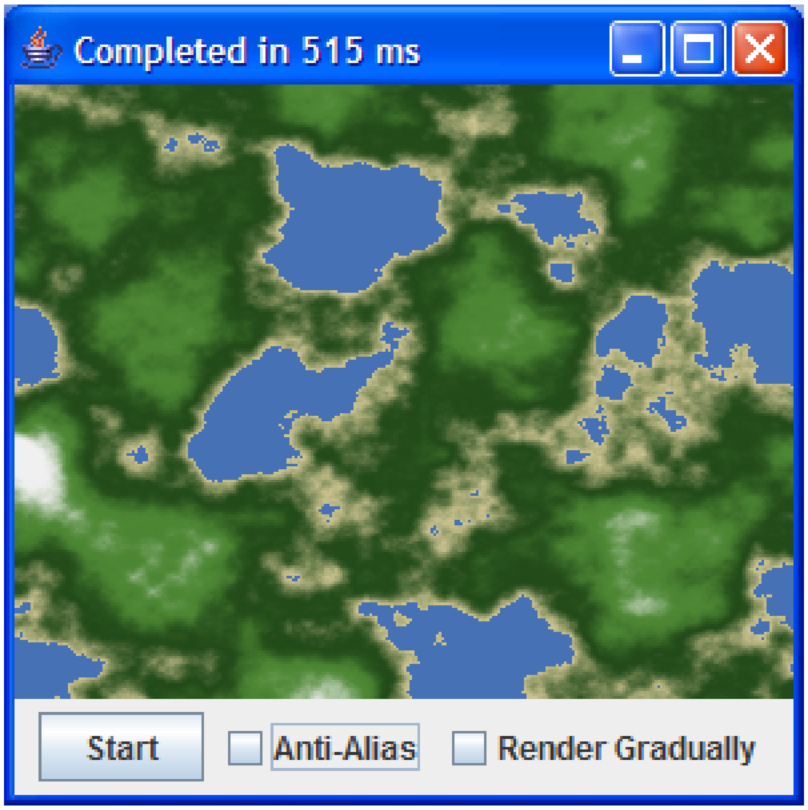
3.9. Map Texture Map

Let's try making a map out of our noise function and shade the landscape according to the height and add in a sea level at which point we just shade the texture blue.

We'll start with a `Map` texture class that takes a gradient, a sea color, and the usual noise parameters for scale and octaves, and a value indicating where the sea level is. To calculate the map, we'll calculate a noise value based on the `(u,v)` values, and adjust it so the values fill the range 0..1 a little better. Since the value could go outside the 0..1 range, we will clamp it. Finally, we check the height value and if it is above sea level, we use a map color. If it is below sea level, we use the sea color. We don't want to fade into the sea color since the edge of the water on land is a fairly hard edge.

```
    double height = noise.fbmNoise2(u * scale, v * scale, octaves);
```

```
height = height * 1.65;
height = height - 0.25;
height = GraphUtils.clamp(height);

if (height < seaLevel) {
    value.setColor(seaColor);
} else {
    value.setColor(gradient.interpolate(height));
}
```



3.10. Saving High Resolution Images

If you want to generate a high resolution image outside of the viewer, and save it to disk for printing or using as a background, you can do so by going directly to the `TextureImage` class which is the foundation class for rendering textures that the `TextureViewer` class uses. We simply create an instance and pass a resolution into the constructor and call the `render` method. Once the image has rendered to its internal canvas, we can then save the image, or in the case of the viewer, render it in the window. Note that since the renderer is multi-threaded by default, the `renderAndWait` method must be used so it will not execute asynchonously. If we use the `render` method, then we will probably end up saving the image when it is only 10% of the way through.

```
public class SaveFileDemo {

    public static void main(String[] args) {

        TextureImage textureImage = new TextureImage(2400, 2400);
        Texture mand = new Mandelbrot(-0.86429,-0.2578125,-0.5671,-0

        //we want 4X4 anti-aliasing
        textureImage.setAntiAliased(true);

        textureImage.renderAndWait(mand);

        textureImage.saveAsPNG("c:\\myImage.png");
    }
}
```

3.11. Signals, Signals Everywhere

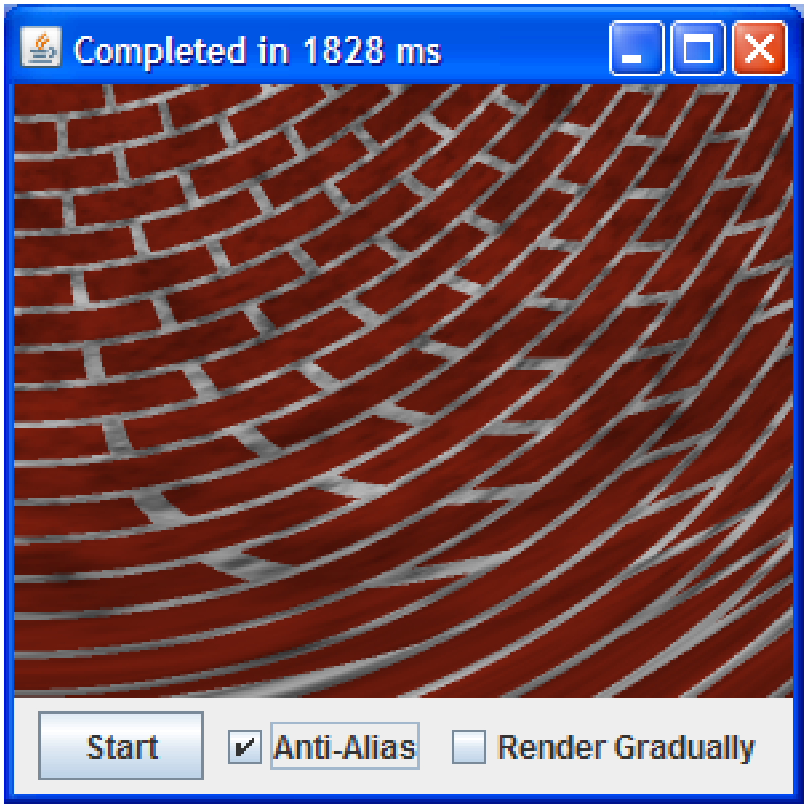
It could be quite easy to go overboard with using signals in textures and use them for every single constant numerical value. We have a `ConstantSignal` signal class that just returns a single value that could be used for default constant values. For example, we could make the speed or octaves of a noise texture dependent on the `u,v` values. As an example, we will create a `UVRotation` texture where the angle of rotation is dependent on a `ChannelSignal` used.

```
public class UVSignalRotateTexture extends AbstractTexture {

    private ChannelSignal signal;
    private Texture source;

    public UVSignalRotateTexture(Texture source, ChannelSignal signa
        this.signal = signal;
        this.source = source;
    }

    public void getColor(double u, double v, RGBAColor value) {
        //get the angle from the signal
        double radAngle = calculateSignalFromFilter(u, v, signal);
        //do the rotation
```

```
            double ru = Math.cos(radAngle) * u - Math.sin(radAngle) * v;
            double rv = Math.sin(radAngle) * u + Math.cos(radAngle) * v;
            //get the final color with the newly rotated u,v values
            calculateColorFromTexture(ru, rv, source,value);
        }
    }
```

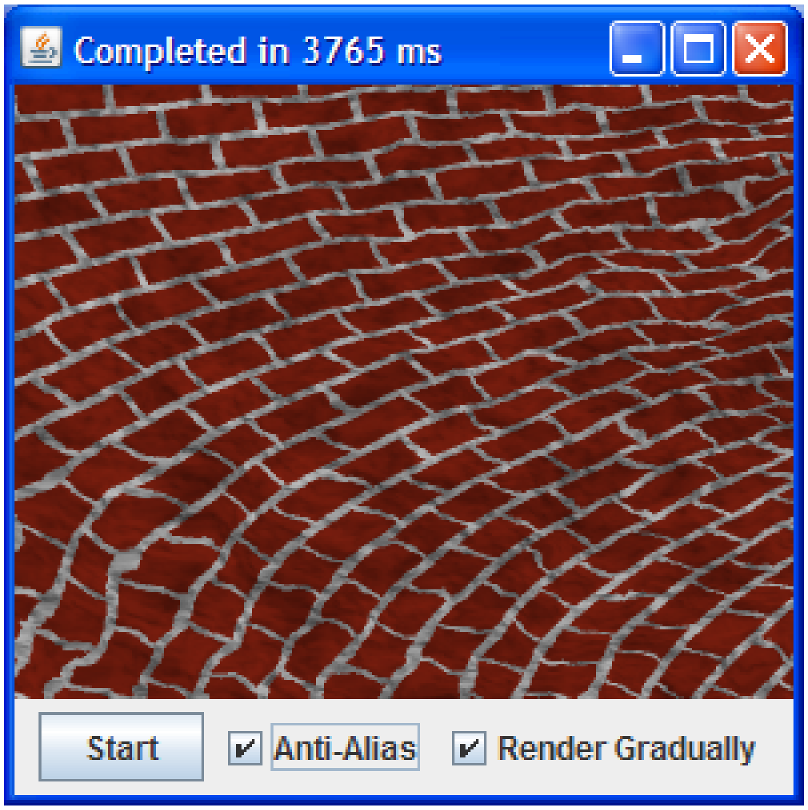
We get a value from the signal and then use this to drive the angle of rotation. Let's try it out using a `USignal` signal that returns the value 0 to 1 as it moves across the texture.

```
public static void main(String[] args) {

    Texture texture = new DirtyBrick();
    ChannelSignal signal = new USignal();
    texture = new UVSignalRotateTexture(texture, signal);

    TextureViewer.show(texture);
}
```


Completed in 1828 ms

Again, we gain the benefits of a pluggable system where we can plug on signal or texture into another and use that new value. We can plug this signal into a `NoisySignal` signal which will add random elements into it. We also changed it from a `USignal` to a `VSignal` .

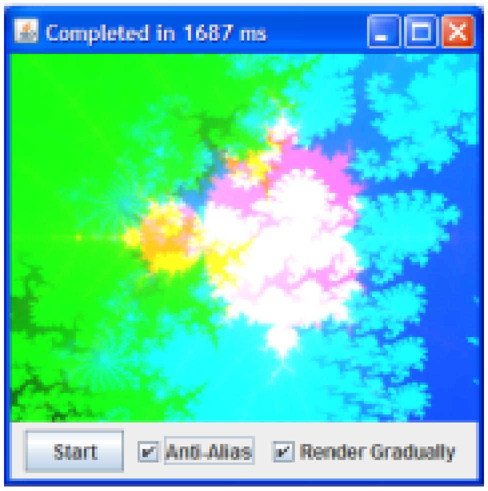
```
public static void main(String[] args) {

  Texture texture = new DirtyBrick();
  ChannelSignal signal = new  VSignal();
  signal = new NoisySignal(signal,0.2,8,12);
  texture = new UVSignalRotateTexture(texture, signal);
  TextureViewer.show(texture);
}
```



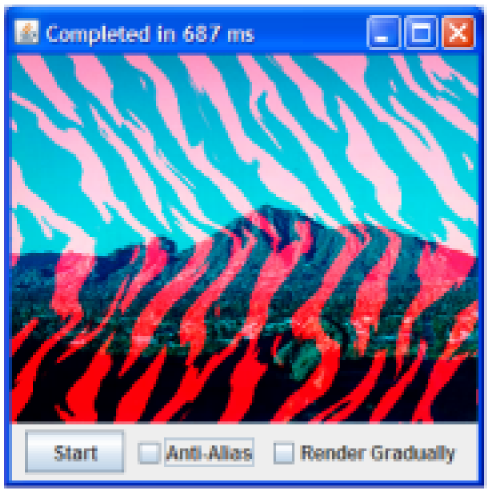
We could make every single numerical parameter in a texture a signal parameter, but it would be over the top. However, it does indicate how useful the `ChannelSignal` architecture can be. We also have a class which lets you provide `ChannelSignals` to make up the red,green, blue, and alpha channels. You can pass in either 1 signal to be used for all channels, or one for each channel.

```
Listing A:

public static void main(String[] args) {

  ChannelSignal signal1 = new MandelbrotSignal(-1.487, -0.006950, -1
  ChannelSignal signal2 = new MandelbrotSignal(0.35, 0.35, 0.3507, 0
  ChannelSignal signal3 = new MandelbrotSignal(-0.75401, -0.23191, -0

  ChannelSignal signal4 = new SineWave(100, 10, 2);// new ConstantSig

  Texture texture = new ChannelTexture(signal1, signal2, signal3, sig
  TextureViewer.show(new Background(texture, RGBAColor.black()));
}




Listing B:

  public static void main(String[] args) {

    Texture image = new ImageTexture("c://camelback.jpg");

    ChannelSignal signalRed = new AnimalStripe();
    ChannelSignal signalGreen = new TextureSignal(image,ChannelSignal
    ChannelSignal signalBlue = new TextureSignal(image,ChannelSignal

    ChannelSignal signal4 = new ConstantSignal();

    signal4 = new InvertSignal(new RadialSignal(10));


    Texture texture = new ChannelTexture(signalRed, signalGreen, sign
    TextureViewer.show(new Background(texture, RGBAColor.black()));

    }
}




Listing C (variation on B):

  public static void main(String[] args) {
```

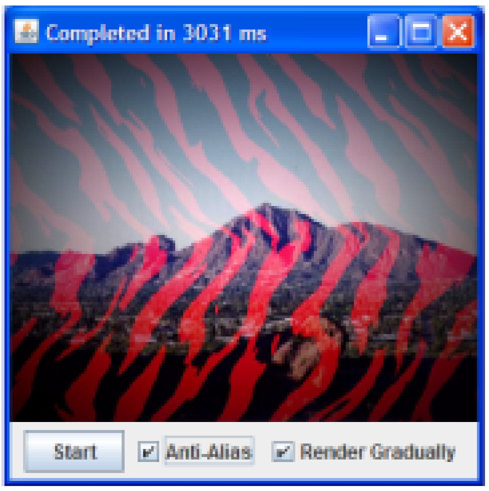
```
            Texture image = new ImageTexture("c://camelback.jpg");

            ChannelSignal stripe = new AnimalStripe();
            ChannelSignal imageRed = new TextureSignal(image,ChannelSignal.Ch
            ChannelSignal defaultValue = new ConstantSignal(1);

            ChannelSignal signalRed = new SignalThreshold(defaultValue,imageR

            ChannelSignal signalGreen = new TextureSignal(image,ChannelSigna
            ChannelSignal signalBlue = new TextureSignal(image,ChannelSignal

            ChannelSignal signal4 = new ConstantSignal();

            signal4 = new InvertSignal(new RadialSignal(0.9));


            Texture texture = new ChannelTexture(signalRed, signalGreen, sign
            TextureViewer.show(new Background(texture, RGBAColor.black()));

        }
    }
```



Listing A



Listing B



Listing C

3.12. Springing Along

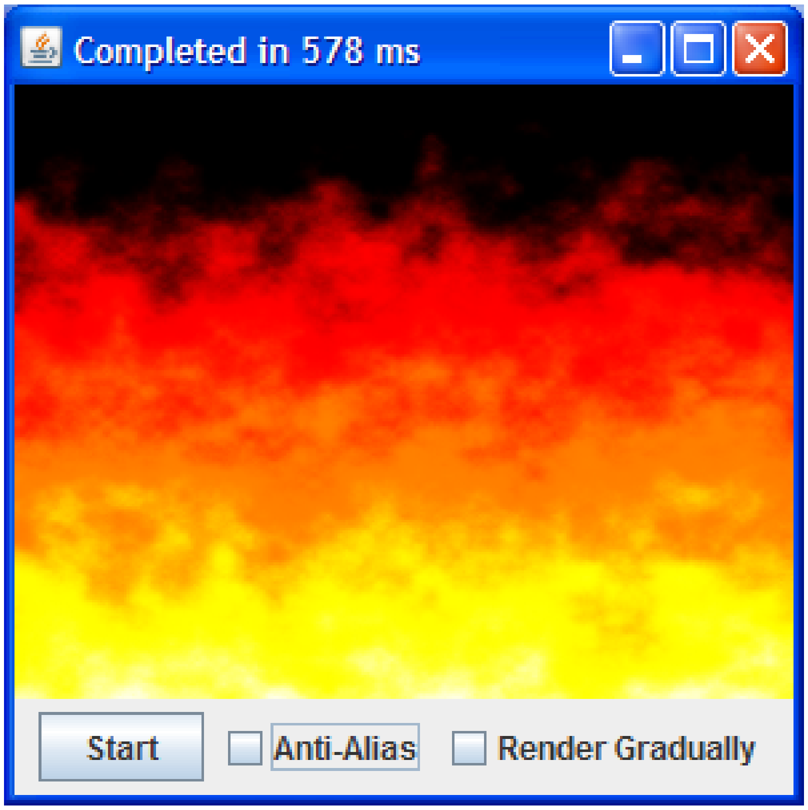
This section describes how to hook up textures and signals using Spring. This way, we could modify our textures without recompiling, and we can re-use some of the items much easier. This demonstration describes the content of the `application.xml` configuration part as well as the code required to fetch the final texture bean.

To start with, we'll create a fire gradient using a `ColorGradient` . Not only do we have to create a bean for the gradient, but also for the colors it contains.

```
    <bean name="fireGradient" class="org.texturemaker.geom.ColorGradi
        <property name="colors">
            <list>
                <ref bean="blackColor"/>
                <ref bean="blackColor"/>
                <ref bean="redColor"/>
                <ref bean="orangeColor"/>
                <ref bean="yellowColor"/>
                <ref bean="whiteColor"/>
            </list>
        </property>
    </bean>

    <bean class="org.texturemaker.geom.RGBAColor" name="whiteColor">
        <property name="red" value="255"/>
        <property name="green" value="255"/>
        <property name="blue" value="255"/>
    </bean>

    <bean class="org.texturemaker.geom.RGBAColor" name="orangeColor"
        <property name="red" value="255"/>
        <property name="green" value="128"/>
        <property name="blue" value="0"/>
    </bean>

    <bean class="org.texturemaker.geom.RGBAColor" name="yellowColor"
        <property name="red" value="255"/>
        <property name="green" value="255"/>
        <property name="blue" value="0"/>
    </bean>

    <bean class="org.texturemaker.geom.RGBAColor" name="redColor">
        <property name="red" value="255"/>
        <property name="green" value="0"/>
        <property name="blue" value="0"/>
    </bean>
    <bean class="org.texturemaker.geom.RGBAColor" name="blackColor">
        <property name="red" value="0"/>
        <property name="green" value="0"/>
        <property name="blue" value="0"/>
    </bean>
```

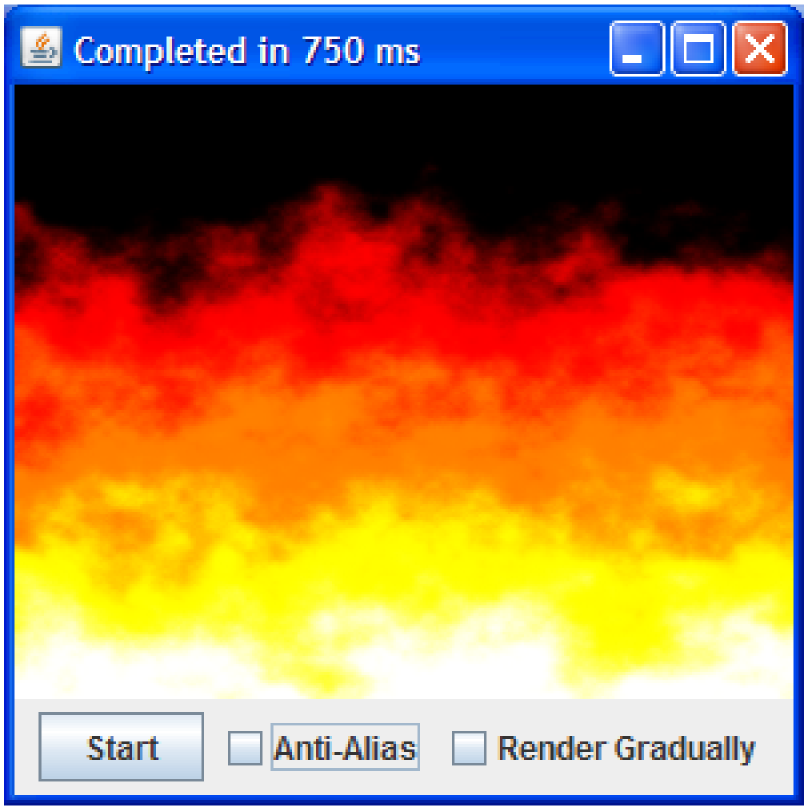
This creates the fire gradient that we need. We will construct our fire using a chain of textures to modify the basic horizontal fire gradient. We start with a `NoisySignal` that uses a `VSignal` signal as its source signal. This is used to disturb the `(u,v)` values used to calculate the fire color in the `ColorGradient`

```
    <bean name="vSignal" class="org.texturemaker.signals.VSignal"/>

    <bean name="noisySignal" class="org.texturemaker.signals.NoisySig
        <property name="source" ref="vSignal"/>
        <property name="size" value="0.3"/>
        <property name="octaves" value="15"/>
        <property name="scale" value="15"/>
    </bean>

    <bean name="verticalFire" class="org.texturemaker.textures.signal
        <property name="gradient" ref="fireGradient"/>
        <property name="signal" ref="noisySignal"/>
    </bean>
```



This looks ok, but we don't have much intensity (white parts) of the fire at the bottom, and it looks fairly flat with no flames leaping upwards. We can use a translation to move the fire upwards so we can see more of the bottom, but while we are at it, we could introduce some more noise to the fire. To do this, we add another `NoisySignal` that is fed into our original noisy signal. This time, we give it a negative size value so the fire shifts upwards as the noise is introduced.

```
<bean name="noisySignal" class="org.texturemaker.signals.NoisySig
    <property name="source" ref="offsetNoisySignal"/>
    <property name="size" value="0.3"/>
    <property name="octaves" value="15"/>
    <property name="scale" value="15"/>
</bean>

<bean name="offsetNoisySignal" class="org.texturemaker.signals.No
    <property name="source" ref="vSignal"/>
    <property name="size" value="-0.3"/>
    <property name="octaves" value="5"/>
    <property name="scale" value="5"/>

</bean>
```



Now we have some more of a white heat at the bottom, but it still looks rather flat. To remedy this, we can scale the image so the `v` value changes much faster.

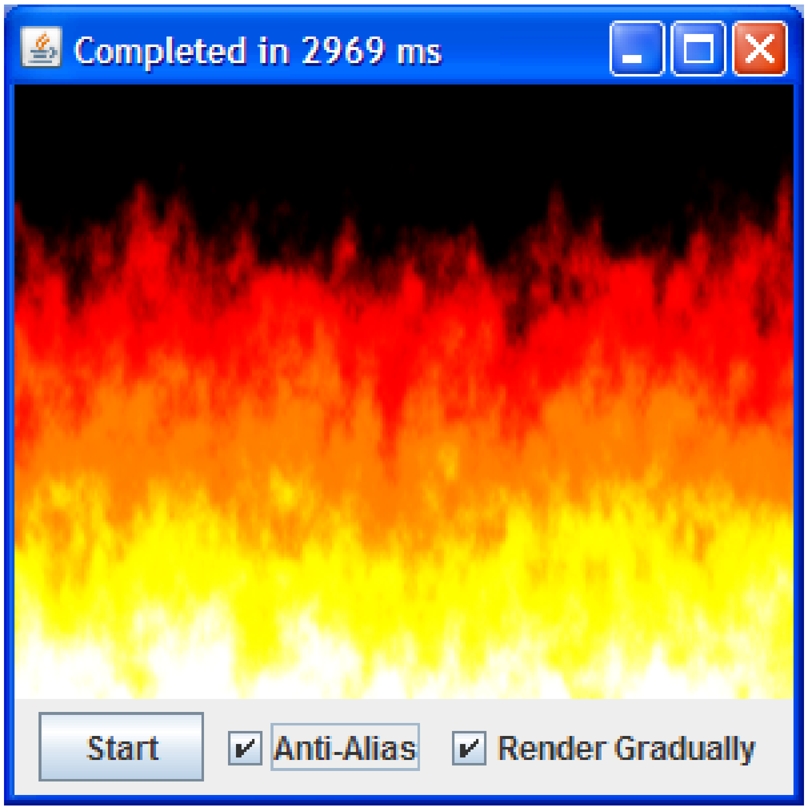
```
<bean name="scaleUV" class="org.texturemaker.textures.uv.UVScale">
    <constructor-arg index="0" ref="verticalFire"/>
    <constructor-arg index="1" value="2.5"/>
    <constructor-arg index="2" value="1"/>
</bean>
```

This time, we also used the constructor arguments to set the bean up. In order to view our texture, we need to initialize the Spring environment, pass it the `application.xml` file and request and instance of the texture we want to display.

```
public static void main(String[] args) {
    Resource res = new ClassPathResource("applicationContext.xml");
```

```
        XmlBeanFactory factory = new XmlBeanFactory(res);
        Texture bean = (Texture) factory.getBean("scaleUV");
        TextureViewer.show(bean);
    }
```

It is really that simple. Using Spring to set up the textures could be very useful for defining a library of often used textures, that can just be imported into your spring configuration. It can even let you define commonly re-used textures such as `NoisySignal` instances to re-use for putting a subtle amount of noise on a signal for example.



Chapter 4. Textured Tiger Burning Bright

This chapter walks through the process of building a complex texture from start to finish. In this case, we will build a tigerskin texture. We start by thinking about the different components that go into the texture, and then the best way to build the texture such that we can re-use the different elements of it. We start by dividing the texture into different elements.

4.1. Elements in the texture

1) The main fur color, in this case a gold to white/grey gradient.
2) The actual stripes themselves, sort of noisy black lines running across the texture in one direction.
3) The fur texture itself which gives the impression that the color is painted on individual hairs. This will be tricky, and will mostly consist of a noise that is scaled across the fur in one direction.

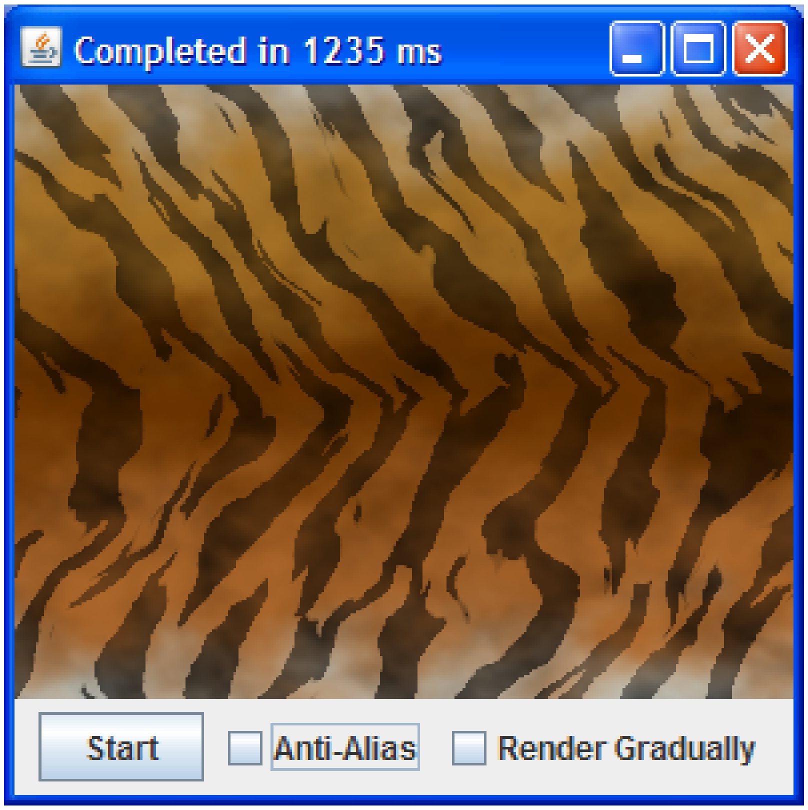
4.2. Implementing Tiger Colors

We can split this into several different textures. The first will be called `TigerColor` and will consist of the main fur colors with the stripes on top of them. The stripes themselves will be generated from a `ChannelSignal` called `AnimalStripes` . Once we have our tiger colors, we can see how we like it and then pass that texture into our `Fur` texture which will apply a directional noise to the texture.

```
public class TigerColors extends AbstractTexture {

    private ColorGradient baseFurGradient = new ColorGradient();
    private Texture  stripe;
    private Texture mixer;

    public TigerColors() {

        baseFurGradient.add(new RGBAColor(201,203,202));
        baseFurGradient.add(new RGBAColor(171,118,40));
        baseFurGradient.add(new RGBAColor(108,53,0));
        baseFurGradient.add(new RGBAColor(173,103,43));
        baseFurGradient.add(new RGBAColor(200,205,198));
        //based our stripe on a black texture with the alpha channel
        stripe = new AlphaSignal(new RGBAColor(0,0,0,0.7),new NoisySi
        //add some noise in here so it's not so sharp on the edges.
        stripe = new UVNoiseTranslate(stripe,8,6,0.05);

        //make the background based on the horizonal gradient using o
        Texture background = new HorizontalGradient(baseFurGradient)
        //again, apply a translation to the colors so it is not such
        background = new UVNoiseTranslate(background,10,4,0.2);

        //create a mixer that applies our black stripes to our gradie
        mixer = new MergedTexture(background, stripe);
    }

    public void getColor(double u, double v, RGBAColor value) {
        mixer.getColor(u, v,value);
    }
}
```

The only thing left here is to define our `AnimalStripe` signal. We calculate this by using multiple sine values, some of them rotated and then we scale, add and subtract one value from another. We also apply noise to the values to give us more random edges.

```
public class AnimalStripe extends AbstractChannelSignal {

    private static PerlinNoise noise = new PerlinNoise();
    private ChannelSignal sine1;
    private ChannelSignal sine2;
    private ChannelSignal sine3;

    private double centralDisplacementSize;

    public AnimalStripe(double centralDisplacementSize) {
        this.centralDisplacementSize = centralDisplacementSize;
        sine1 = new SineWave(10, 3, 1);
        sine2 = new SignalUVRotate(new SineWave(12, 4, 1), 30);
        sine3 = new SignalUVRotate(new SineWave(12, 4, 1), -30);

    }

    public double getValue(double u, double v) {
        //calculate the distance from the vertical center
        //this doesn't handle out of range u,v values
        double dist = Math.abs(0.5-v);
        u = u + (dist * centralDisplacementSize);

        //scale our u,v
        u = u * 3.5;
        v = v * 3.5;

        //calculate a couple of re-usable noise values
        double n1 = noise.fbmNoise2(u, v, 3) * 1;
        double n2 = noise.fbmNoise2(u + 37, v + 29, 3) * 1;

        //rescale u,v so we get an elongated set of stripes
        u = u * 1.5;
        v = v * 0.5;

        //calculate the sines and add/substract
        double s1 = sine1.getValue(u, v) * 0.5;
        double s2 = sine2.getValue(u + n2, v) * 1.4;
        double s3 = sine3.getValue(u + n2, v + n1);
        double value = s1 + s2 - s3;

        //clip the value so only 0.5 and above is a stripe
        return clip(value);

    }

    //if value > 0.55 return 1
    // if 0.55 > value > 0.5, return 0..1
    // else return 0
    private double clip(double value) {
        if (value >= 0.5) {
            if (value > 0.55) {
                return 1;
            } else {
                return (value - 0.5) * 10;
            }
        } else {
            return 0;
        }
    }

    public double getCentralDisplacementSize() {
        return centralDisplacementSize;
    }

    public void setCentralDisplacementSize(double centralDisplacemen
        this.centralDisplacementSize = centralDisplacementSize;
    }
}
```

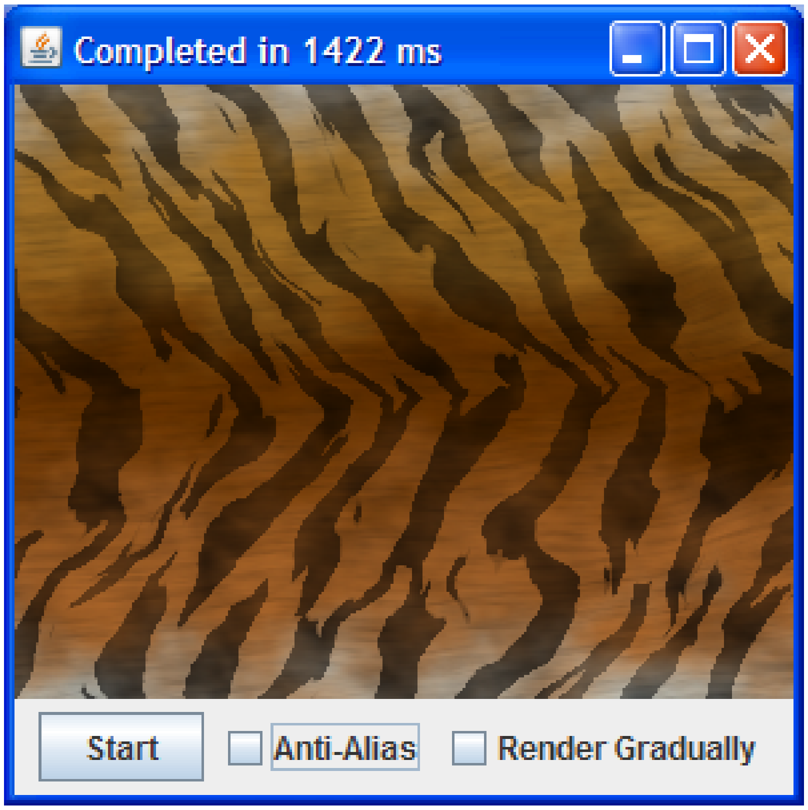


4.3. Adding a fur texture

Now we have our colors, we could technically just use this texture if we were interested in texture mapping a 3D model. However, to complex the exercise, we will apply the fur coloring to a `Fur` texture which will give it a fur-like grain.

Our fur texture simply takes the source texture (our fur color) , and disturbs it in the direction of the fur. It also applies a directional noise texture over the top of it. I also added a Channel Signal to rotate the `(u,v)` values to rotate the direction of the fur as it moved. I set the angle for each point based on a noise value based on the `(u,v)` values.

```
public class Fur extends AbstractTexture {

    private Texture furColor;
    private ChannelSignal noiseSignal;
    private SignalUVRotate rotationSignal;

    public Fur(Texture furColor) {
        this.furColor = furColor;
        noiseSignal = new NoiseSignal(1,4,0.5);
        noiseSignal = new SignalUVScale(noiseSignal, 14,128);
        rotationSignal = new SignalUVRotate(noiseSignal, 10);

    }

    public void getColor(double u, double v, RGBAColor value) {

        double angle = noise.fbmNoise2(u*3, v*3, 3);
        rotationSignal.setAngle(angle*10);
        double val = calculateSignalFromFilter(u, v, rotationSignal)
        //make it fall off quickly
        val = val * val;

        //get the fur color
        calculateColorFromTexture(u, v,getFurColor(), value);
        //invert the coefficient value
        value.scale(1-val);
        //set the alpha to 1, this shouldn't be scaled.
        value.setAlpha(1);
    }

    public Texture getFurColor() {
        return furColor;
    }

    public void setFurColor(Texture furColor) {
        this.furColor = furColor;
    }
}
```



This demo can be seen by invoking `TextureViewer.show(new Fur(new TigerColors()));` in an application. If you are rendering this at a high resolution, you might want to add a UV Noise Translator to jiggle the U,V values slightly to reduce some of the smoothness. While the textures can render at different resolutions, they don't always look their best at all resolutions. You can see the effects with the noise translator by using `TextureViewer.show(new UVNoiseTranslate(new Fur(new TigerColors()),100,3,0.01));`