Summary and Implementation of "A Novel Image Interpolation Technique Based on Fractal Theory"

Joshua Scoggins

November 17, 2011

Contents

1 Introduction

Fractal Interpolation is an image scaling technique that takes advantage of the concept of fractals to perform image zooming while ensuring that smoothing doesn't occur as a side effect of the scaling operation. Once the technique is understood it is quite simple but the process of understanding how this technique works is the challenge. The paper I selected was called "A Novel Image Interpolation Technique Based on Fractal Theory". It describes a hardware implementation of fractal interpolation, the algorithms used, and the theory behind fractal interpolation.

2 Fractal Theory

Fractals are an expression of what is called an iterable function system (IFS). An IFS consists of a series of functions that are applied to a set of values to change it in a consistent way. Normally these functions take the form of an affine transformation which is a way of representing the different aspects of a transformation operation in a consistent manner. These affine transformations are defined by the function

$$k\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

or

$$k(x, y) = (ax + by + e, cx + dy + f)$$

where a, b, c, d control rotation and scaling, while e and f control linear translation [4].

Using a set of affine transformations, defined using the set of functions $k_1, k_2, \ldots, k_N$, it is possible to define a wide variety of different fractal operations. This set is applied to the original image, denoted as the set of pixels C, through what is called the "Hutchinson Operator" which is defined as

$$k(C) = \bigcup_{n=1}^{N} k_n(C)$$

A fractal geometry is obtained by applying the Hutchinson operator to the previous geometry repeatedly. This makes fractals ideal for both interpolation and compression because the techniques required can be easily used in a discrete environment such as a computer. Fractals also allow us to describe a natural image in terms that are easier for a computer to use due to the large amount of self-similarity and random distribution within nature.

This information combined with "fractal Brownian motion theory" allows for image interpolation to occur using the following functions [4]:

$$x_{\text{added}} = \frac{(x_i + x_{i+1})}{2} + s * w * rand(g_0)$$

$$y_{\text{added}} = \frac{(y_i + y_{i+1})}{2} + s * w * rand(g_1)$$

Where x_{added} and y_{added} are the pixels interpolated. x_i and y_i are the known pixels. s defines direction the pixel will move and w controls the distance the pixel will move. g_0

and g_1 are GUID[1] based hash values used to seed the random number generator. This is the basis of fractal interpolation and requires that a single function replace $s * w * rand()$. This function is known as the fractal dimension and is responsible for ensuring random distribution essential to the process of fractal interpolation.

3 Fractal Dimension

As stated above the fractal dimension is a way of expressing $s * w * rand()$ without resorting to complex vector mathematics. This dimension defines how smooth a given curve will be within the resultant image. In short, the fractal dimension is the amount of space that a curve occupies within a given space the more accurate this value can be the better the interpolation will be.

There are many different fractal dimensions used to describe this smoothness. This includes the Hausdorff dimension and Minkowski-Bouligand dimensions. The Hausdorff dimension is defined in any subset of metric space where the dimension is the generalization of the fractal dimensions which includes the similarity dimension[4]. This dimension is defined as the following function:

$$H^s(F) = \lim_{\sigma \to 0} H^S_\sigma(F)$$

where

$$H^S_\sigma(F) = \inf\{\sum_{i=1}^{\infty} |U_i|^s : \{U_i\}\text{is } \sigma - \text{cover of } F\}$$

Where $F \in R^n$ where F is a non-empty set, σ is a real number above zero, and U_i is a limited subset of R_n.

This formula works by figuring out the exact shape of the fractal edge and computes it's area. The problem is that while this formula generates a very accurate real number it is very difficult to compute[4]. To solve this problem another dimension called the Box Counting Dimension was proposed which is far simpler[4]. It works off of the concept of self-similarity which is defined as a bounded set, A, that is the union of a number, N_r of non-overlapping scaled copies of itself, where r is the scaling factor. The dimension is defined as:

$$D = \frac{\log(N_r)}{\log(\frac{1}{r})}$$

While there are many other fractal dimensions, the box counting method is probably the most efficient computationally and is one of the most widely used dimensions for fractal interpolation. There is one problem with this dimension, it can't be efficiently implemented in hardware due to the use of logarithms[4].

However, the problem still remains to find a new dimension that will approximate the textural features of a natural image. Thus this paper introduces the concept of something called the *k-dimension*. It is defined as:

[1] A GUID is a 128-bit number represented in hexadecimal. It is used in GPT which is the partition table format for EFI based system.

$$k = \frac{\sum_{j=0}^{n-1}\sum_{i=0}^{n-2}(|G_{(i+1,j)} - G_{(i,j)}|) + \sum_{i=0}^{n-1}\sum_{j=0}^{n-2}(|G_{(i,j+1)} - G_{(i,j)}|)}{2 * 255^n}$$

Where k describes the roughness of the image, G is the set of intensity values within the source image, and n is the square size of the pixel blocks. This dimension is meant to be easy to compute on dedicated hardware. In fact, this dimension was designed with operating with the express purpose of operating on MxM pixel blocks efficiently. However, before actually talking about using this dimension it's important to understand how the extra pixels will be generated.

3.1 Random Midpoint Displacement Method

The random midpoint displacement method is also known as a "plasma fractal". It works by taking a set of colors and arranging as the corners of a rectangle. The center point of the rectangle is determined by taking the values of these four corners, averaging them, and then adding a "displacement factor". This adds a degree of "noise" to the center intensity. The midpoint between each edge is computed by taking the average of the two points. Once theses new points have been acquired the rectangle is divided into four smaller rectangles. The process repeats itself until the width or height of each subdivided section is less than a single pixel. This process is used to generate realistic looking terrain in movies and video games. This method is used in image zooming to distribute the pixels with an unknown intensity across the image. With respect to fractal interpolation the "displacement factor" is the fractal dimension and each midpoint is a new pixel in the resultant image.

4 Fractal Interpolation for Natural Images

As stated above, fractal interpolation is a technique where an iterable function system (IFS) is used to zoom an image from it's original resolution to a larger resolution while maximizing image coherency and minimizing smoothing artifacts introduced through the interpolation process.

Fractal Interpolation is the application of random midpoint displacement where the displacement value is computed from the fractal dimension. If the image is being upscaled by a large factor then the required computational overhead is quite massive due to the number of required subdivisions for each pixel quad. To mitigate this the image is divided into MxM pixels subsets and the K-dimension is computed[4]. Figure 1 on the following page details the computation order of each MxM pixel subset where the grey pixels are the original pixels and the white pixels are the added pixels[4].

Using figure 1 on the next page it's important to note that pixel e must be computed before pixel f or the value will not be correct. Pixel E is computed with the following formula:

$$G(x,y) = \frac{1}{4}\{G(x-1,y-1)+G(x+1,y)+G(x-1,y+1)+G(x+1,y+1)\}+\sqrt{1-2^{2k-2}}*||\Delta X||*\text{Gauss}*\sigma$$

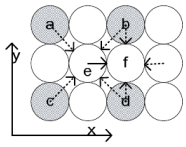

Figure 1: Order of Computation for an MxM pixel subset

Once pixel e is computed it's possible to compute pixel f

$$G(x,y) = \frac{1}{4}\{G(x,y-1)+G(x-1,y)+G(x+1,y)+G(x,y+1)\}+2^{-\frac{k}{2}}*\sqrt{1-2^{2k-2}}*||\Delta X||*\text{Gauss}*\sigma$$

Where Gauss is a gaussian random number, σ is the average pixel variance, k is the fractal dimension, and $||\Delta X||$ is the distance between pixels.

Since this is supposed to a hardware implementation of fractal interpolation the image is divided into 4x4 pixel blocks that the dedicated hardware operates on. This image division is important because it reduces the amount of storage required by the hardware and also ensures faster completion time because the hardware is really just operating on 4x4 images that make up a far larger image.

4.1 Paper Results

Compared to bilinear and nearest neighbor interpolation, fractal interpolation surpassed each in a different area. Against nearest neighbor interpolation fractal interpolation looked better and got rid of the "sawtooth" effect on edges. With bilinear interpolation the preservation of texture was inferior to fractal interpolation[4]. In the end fractal interpolation preserved the high-frequency components from the original image the best. It is because of this that fractal interpolation is probably the superior choice to both bilinear and nearest neighbor interpolation.

This paper was an interesting read but what was even more interesting was implementing it.

5 Implementation Details and Results

This paper was quite a challenge to pull an implementation from because the grammar was so bad that my mind seemed to skip sentences continually. However, with that being said there were tons of cryptic implementation details complete with a hardware block diagram. My final solution is a combination of the paper's implementation combined with my own ideas to get a working solution. I divided the work of implementing this paper into multiple sections:

1. Converting intensities from bytes to floats
2. The k-dimension function
3. Random midpoint displacement

4. The actual fractal dimension
    (a) Implementing a Gaussian Random Number Generator
    (b) Computing Pixel Variance
5. Detecting F type pixels

5.1 Converting Intensities from Bytes to Floats

Probably one of the biggest things I had to do was to use floats instead of bytes. The reason behind this is that a lot of information is lost by continually forcing the continuous results of the fractal dimension into discrete values via type casting. At the start of the transformation operation I converted the image over to being a two dimensional array of floats acquired through the following formula

$$g = \sum_{x=0}^{W} \sum_{y=0}^{H} \frac{f(x,y)}{\text{float}(2^n - 1)}$$

Which in the case of my current implementation translated to

$$g = \sum_{x=0}^{W} \sum_{y=0}^{H} \frac{f(x,y)}{255.0}$$

This was done ahead of time because if it was done when iterating over each pixel then four multiplies would have to occur with many of them being redundant due to the same value being computed over and over again.

Converting the intensities to floating point also ensured that I wouldn't have to be performing tons of conversions which get expensive and error-prone as the complexity of the equation increases.

When I was done I just converted the image back into a two dimensional array of bytes and passed back across the AppDomain to the main program to be displayed.

5.2 The K-Dimension Function

The first task I had was to implement the k-dimension function described by the following formula:

$$k = \frac{\sum_{j=0}^{n-1} \sum_{i=0}^{n-2} (|G_{(i+1,j)} - G_{(i,j)}|) + \sum_{i=0}^{n-1} \sum_{j=0}^{n-2} (|G_{(i,j+1)} - G_{(i,j)}|)}{2 * 255^n}$$

Where n is the size of the pixel mask to be used and G is the set of intensity values that are mapped to each pixel in the source image[2]. This was the easiest part of the paper to implement. There were no curve balls and the only problem I found was that the image needed to be padded out so that the image mask size wouldn't accidentally cause the image to go out of bounds. In my original implementation this was achieved by padding the image

[2]The paper calls this set Y but I found that extremely confusing

so that it was a multiple of the mask size. The padding caused the image to be scaled using replication scaling which turned out to be another issue all together. In my implementation the method is called `ComputeK`.

5.3 Random Midpoint Displacement

Figuring out how random midpoint displacement worked was probably the biggest and most crucial part of the entire implementation as it generates the extra pixels necessary for the fractal dimension to be useful.

As stated above this displacement method is why this interpolation technique can be called "fractal interpolation". While it took me a while to understand what was going on the idea is really simple. Keep subdividing a 2x2 block into smaller and smaller sub rectangles until the width or height between the corners of each sub-rectangle is less than a pixel. At that point store the average of the four corners plus the fractal dimension to get the intensity of the value at a specified point within the new image.

At first I assumed that this method required pixels from the source image to already exist in the result image equally displaced across the image. This worked fine when the image was square but when the scaling factor was not a whole number then problems would occur where the values were not valid. I realized that the problem was that the algorithm was "randomly" displacing pixels throughout the image instead of uniformly.

Once I realized this I figured out that instead of displacing the source pixels across the result image I should just use the source pixels as the corners of each midpoint rectangle to be subdivided. I chose 2x2 blocks because they were the easiest to work with and didn't require the source image to be padded to be a multiple of the mask size to work properly. In essence I was taking the source image and creating terrain based off it. It just turns out that resultant terrain represents the source image just upscaled to the desired resolution. It also meant that I never had to worry about the resolution of the source or result image because the algorithm was handling the placement of pixels instead of some half-cocked notion of where I thought those pixels should go.

The actual function, `DivideGrid`, uses a recursion instead of a iteration because it was far easier to implement and think about [3]. What was even more interesting about implementing this method was that I found a bug in how Mono performs the floor operation on 32-bit floats and caused an StackOverflow to occur when it should have worked just fine. When computing the new width and height of the subdivided rectangle I took the floor of half the original rectangle's width and height. My logic behind this being that a whole number would converge far faster and be less error prone. It seems that the floor function would return a number that was interpreted as being larger than a single pixel and thus required more subdivision. Normally one would think this would just add an extra iteration to the mix but instead the runtime would enter a strange state where it thought that any number was greater than a single pixel regardless of its real value. Thus the code would cause a stack overflow as it went deeper and deeper trying to get down to less than or equal to a single pixel. I am still exploring if I did something to cause it or if its a bug in the Mono runtime. To fix this problem I do not take the floor of the new width and height which completely

[3]Not to mention that I couldn't really figure out how to cleanly implement it using an iterative approach

fixes the issue and prevents a stack overflow.

This algorithm is actually pretty fast, especially when the displacement factor was zero and once it working I shifted my attention over to computing the fractal dimension.

5.4 Computing the Actual Fractal Dimension

The fractal dimension, as stated earlier in the paper, is the amount of space taken up by the edges of the fractal itself. Computing this value exactly is quite hard and because of that there are many different approximation algorithms that give almost the same result. The paper talked about a new one that was easy to implement in hardware. However, because it was designed for hardware there were a few aspects of it that were difficult to comprehend and then implement. For starters, there are actually two different functions depending on where the pixel is in the image. The pixels that have valid diagonal neighbors I denoted E type pixels where the rest of the unknown pixels are denoted as type F. These names came from a diagram in the paper that explained what each type was defined by and the associated function to compute it's intensity and I use this term so that I could easily identify what kind of pixel I was looking for.

The formula to compute E pixel intensity is:

$$G(x,y) = \frac{1}{4}\{G(x-1,y-1)+G(x+1,y)+G(x-1,y+1)+G(x+1,y+1)\}+\sqrt{1-2^{2k-2}}*||\Delta X||*\text{Gauss}*\sigma$$

The formula to compute F pixel intensity is:

$$G(x,y) = \frac{1}{4}\{G(x,y-1)+G(x-1,y)+G(x+1,y)+G(x,y+1)\}+2^{-\frac{k}{2}}*\sqrt{1-2^{2k-2}}*||\Delta X||*\text{Gauss}*\sigma$$

While I understood what the formula consisted of and what each component did I was stuck trying to figure out how to implement different aspects. The first was $||\Delta X||$

5.4.1 Figuring out how to compute $||\Delta X||$

The paper states that $||\Delta X|| = |\sqrt{(x_1 - x_2)^2 + (y_1 - y2)^2}|$ which is supposed to represent the distance between pixels. I had the hardest time figuring out what that meant because there were so many different pixels that the algorithm was operating on at any given point in time. When I finally figured out how to properly implement random midpoint displacement the answer to this problem fell into place. The values x_1, x_2, y_1, y_2 represent the starting and stopping coordinates of the current pixel block that is being looked at. Thus it means that a 2x2 pixel block starting at $(0, 0)$ and ending at $(1, 1)$ will have a pixel distance of one because those two points become the corresponding points in the formula.

With that mystery solved I shifted my attention over to generating gaussian random number generation.

5.4.2 Implementing a Gaussian Random Number Generator

This was a bit of a stretch to figure out and implement because the paper uses a hardware gauss generator to pull gaussian random numbers from. Since my implementation wasn't

going to be in hardware I needed to figure out another way of generating gaussian random numbers. At first I was preparing myself to implement a complete gaussian random number generator but then I found that the built-in `Random` type in C# made a very good substitute using it's `NextDouble()` method. To ensure an adequate amount of randomness in the generator I seeded the generator with the hashcode of the string representation of a random 128-bit GUID. This is a really effective way to seed random because there are 2^{128} different GUID's that can be generated so the probability of hitting a collision is quite low.

5.4.3 Implementing Variance

Variance was the last piece of the puzzle that I had to figure out how to implement. At first I was going to just omit it but I found that the resulting image became way too bright. Implementing variance actually turned out to be a piece of cake as there is a linear version of it that can be computed at runtime. As stated above the average variance is the amount of difference between the values in a collection. In my code the method is called `Variance`.

5.5 Detecting F Type Pixels

Once I got the fractal dimension implemented and working correctly I saw that the image looked pretty good but the edges were the only issue. Sometimes the color value would be slightly off and create a distracting stair step effect. I knew that this was why the paper gave a separate formula for F type pixels. The only problem I had was how to detect these F type pixels.

I tried tons of different solutions from alternating between E and F type pixels to combining the two fractal dimension formulas together. The method that I found worked the best was to bind the intensity of N_4 to be beyond full bright and compute them in a second pass after all the e-pixels had been computed. That way there wouldn't be any missing pixels to have to work around. This technique fixed the edge problems but introduced what can only be described as "sawteeth". These "sawteeth" are pixels that have very different values compared to the rest of the edge. What's even more interesting is that these sawteeth will always be one pixel in size regardless the scaling factor.

While I'm pretty sure the sawteeth are a side effect of something I don't understand in this paper I still feel that my implementation is a complete success. The image scales well and doesn't suffer from the smoothing effect common with bilinear interpolation.

5.6 Results and Future Improvements

I have to say that the results of the upscale are comparable to the results shown in the paper itself. I used several different pictures to test this algorithm and upscaled them to different resolutions. I tried to find many different "natural" images and acquired images from movies such as Star Trek: The Wrath of Khan, Star Trek (2009), and Star Wars: The Empire Strikes Back. I also used a hand-drawn picture of a pig by mangaka Harumi Chihiro. There is a lot of detail in these images that are perfect for fractal interpolation. See figure 4 on page 13, 9 on page 18, 14 on page 23, and 2 on page 13 respectively.

With all of my changes I have to say that I'm quite impressed with my results. Initially, I thought that the objective of fractal interpolation was to be generally superior to bilinear and nearest neighbor interpolation in every way but I know see that fractal interpolation really shines with images that have a lot of textural detail. To fully see the differences one must upscale an to a very high resolution which can take a while with my implementation but the results are quite interesting. Figure 20 on page 29 shows the difference between bilinear and fractal interpolation when a 1920x1080 image is upscaled to 4096x3072. While the bilinear image is smoother it is missing a lot of the texture present in the image upscaled using fractal interpolation.

It's important to note that all of the example images do not include any f-pixels of any kind. This is because of the sawtooth effect mentioned earlier. However, with that being said the images still look really sharp and detailed without them so I'm not really sure what the point of the f-pixels are in the long run. Taking them out of the equation actually makes the operation run far faster[4]

Even with the results I got I'm not impressed with my implementation. The biggest problem with it being that it can take a really long time for tiny images to be upscaled to a very large resolution. In fact upscaling the pig in figure 2 on page 13 to the pig in 19 on page 28 took nearly 30 minutes to complete. Most of my ideas to solve this problem have to do with using 4x4 pixel blocks. This should decrease the amount of time required to traverse across the image. I would have done this in my current implementation but I hit so many edge cases that I felt it was better to use a 2x2 mask instead because it was less error prone and seemed to generate a fairly high quality image.

If increasing the pixel block size doesn't work then I'll try to reduce the number of subdivides required by introducing the concept of partial upscales that double the image size continually until the requested resolution is reached. The only down side I can see is that the amount of memory consumed may increase greatly. At this point I'm not really sure what else I could improve but I'm pretty sure that ways to optimize this code will appear as time goes on.

It seems that fractal interpolation is perfectly suited for upscaling videos in realtime. Not only because the authors of the paper included a hardware block diagram but that the amount of time to upscale an image from 720x480 to 1920x1080 was really fast even though my implementation could have been better.

6 Conclusion

Fractal Interpolation is a very different method of image interpolation that I believe is far more accurate at the cost of being more computationally complex. However, as time goes on I believe that this complexity will fall by the wayside as computers become more and more powerful.

The paper I read was a nice introduction to the field of Fractal Interpolation mainly because it dove right into the implementation. The only problem with it was the fact that it had many spelling and grammatical mistakes that would have been caught had it been proofread by a native English speaker. The other problem was that certain critical aspects

[4]This was unintended.

of the paper were either vague or missing all together. This included an implementation of random midpoint displacement and the vague description F type pixels that seemed to be plagued with errors.

With all of that said, I am glad that I was able to implement this interpolation algorithm because it was fascinating to learn about something I really had no prior experience in. While the mathematical concept behind fractal interpolation didn't directly come into play, it was quite helpful in understanding why the technique is valid. This area of research fascinates me and I would seriously consider it for my thesis if I wasn't such a "nut" for static code scheduling on EPIC architectures like the Intel Itanium.

A Images

Figure 2: 160x160 Image of a Pig drawn by Harumi Chihiro

Figure 3: Original Image of Khan From *Star Trek 2: The Wrath of Khan*

Figure 4: Upscaled Image of Khan From *Star Trek 2: The Wrath of Khan* using Fractal Interpolation

Figure 5: Another image of Khan from *Star Trek 2: The Wrath of Khan*. Resolution: 720x480

Figure 6: Another image of Khan from *Star Trek 2: The Wrath of Khan*. Upscaled to 1280x720 using Fractal Interpolation

Figure 7: Another image of Khan from *Star Trek 2: The Wrath of Khan.* Upscaled to 1280x720 using Bilinear Interpolation

Figure 8: Another image of Khan from *Star Trek 2: The Wrath of Khan.* Upscaled to 1280x720 using Nearest Neighbor Interpolation

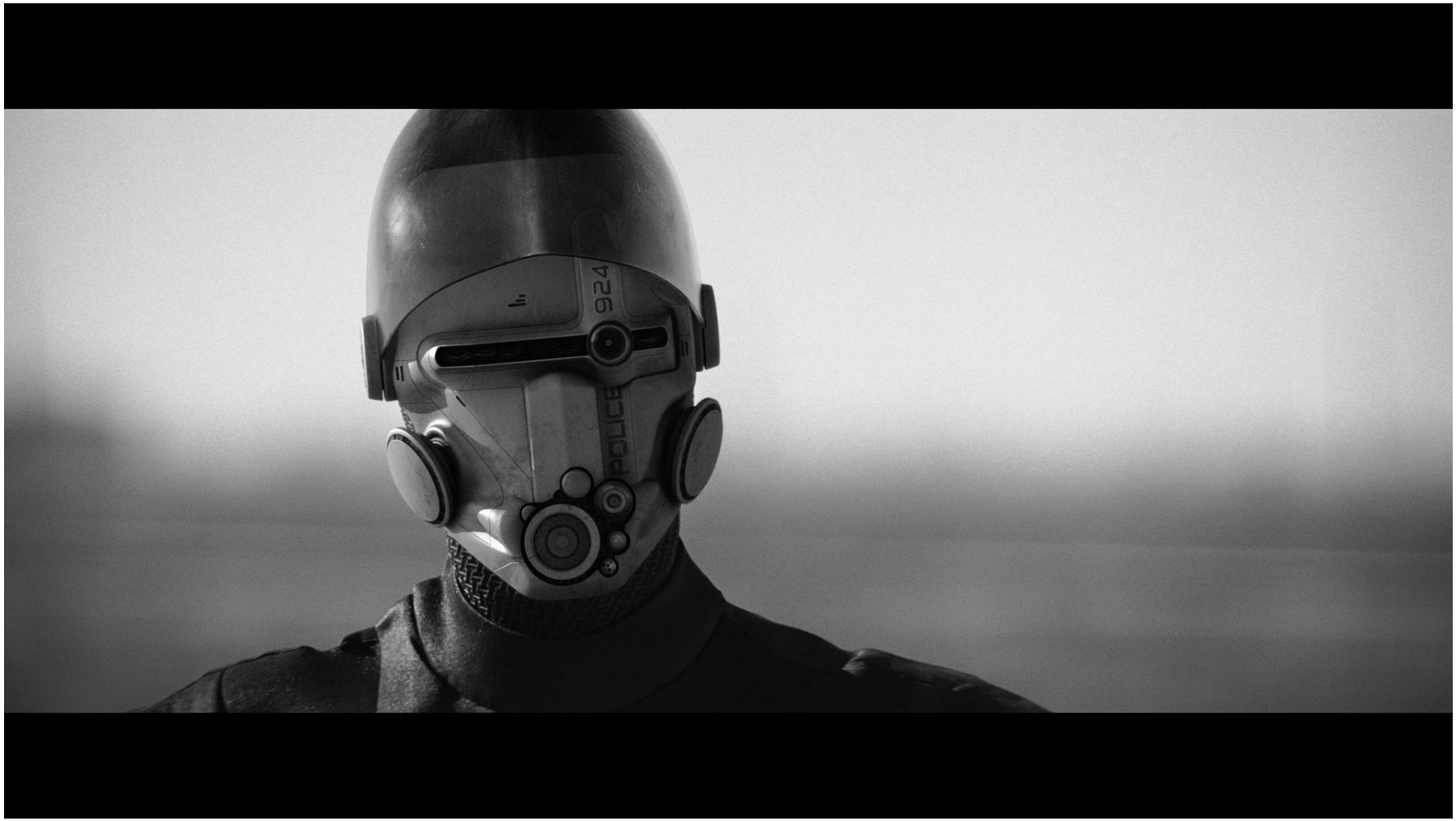

Figure 9: Original 1920x1080 Image of Police Officer from *Star Trek (2009)*

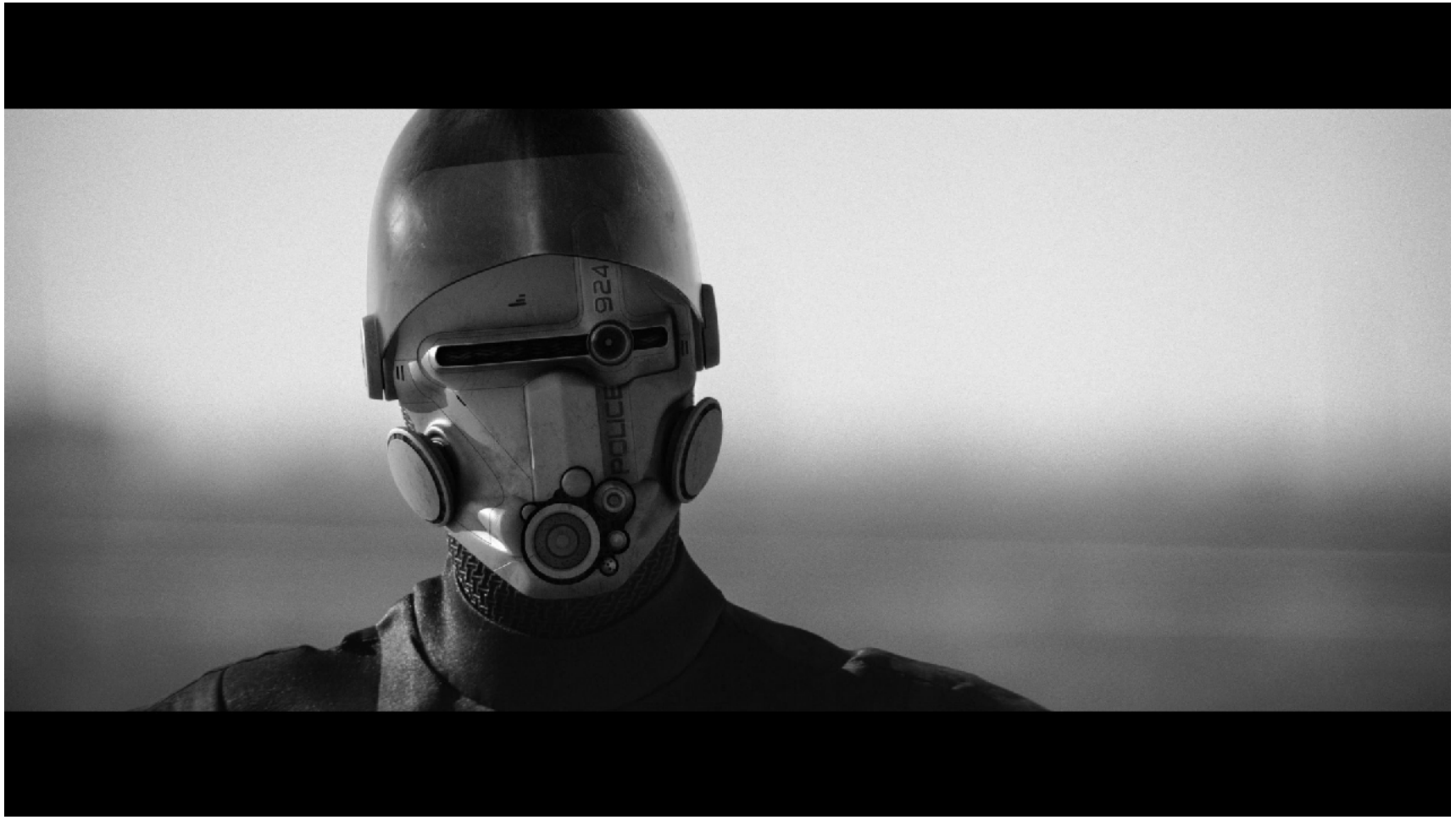

Figure 10: Image of Police Officer from *Star Trek (2009)* Downscaled from 1920x1080 to 1280x720

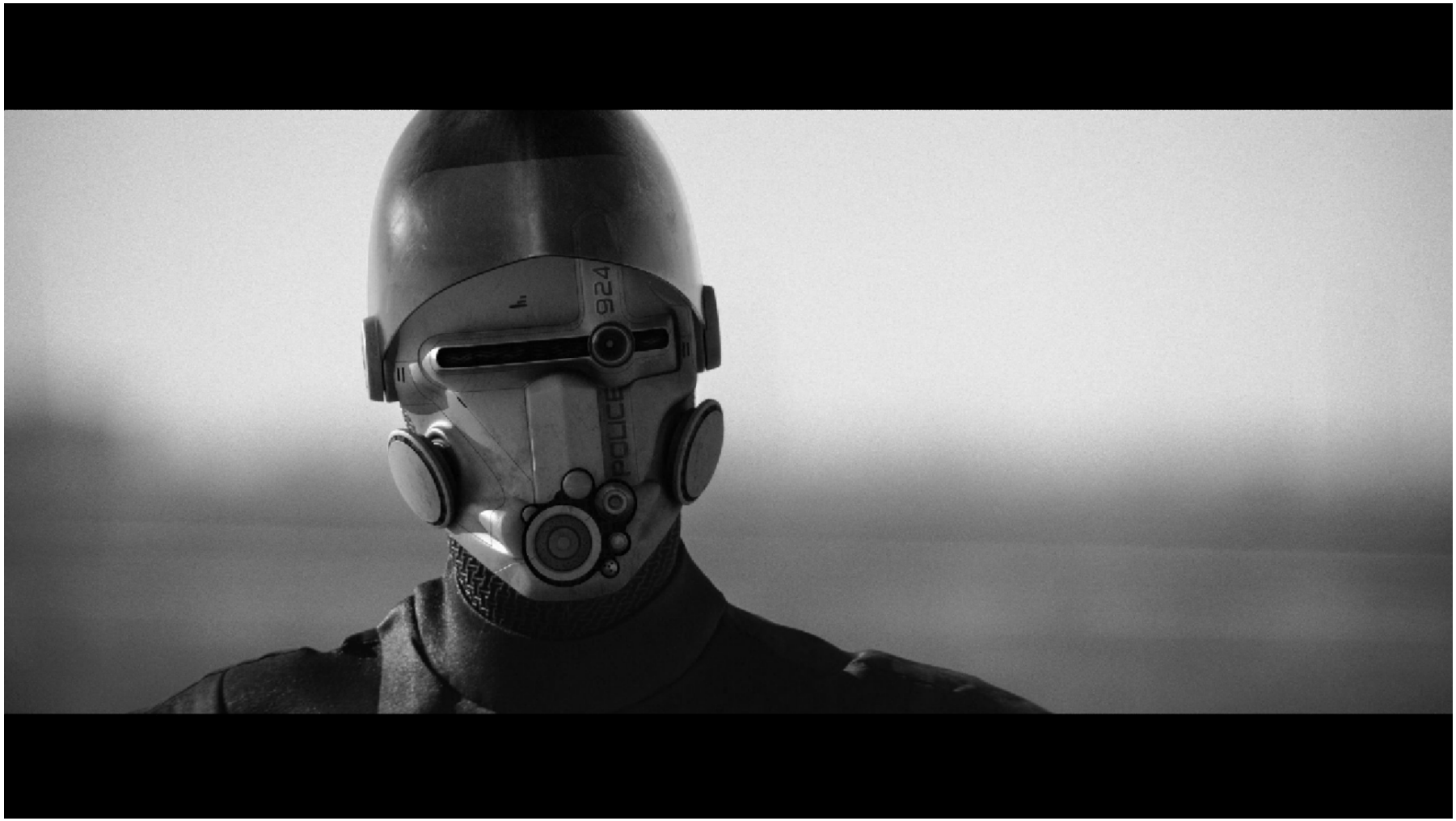

Figure 11: Image of Police Officer from *Star Trek (2009)* upscaled from 1280x720 to 1920x1080 using Fractal Interpolation

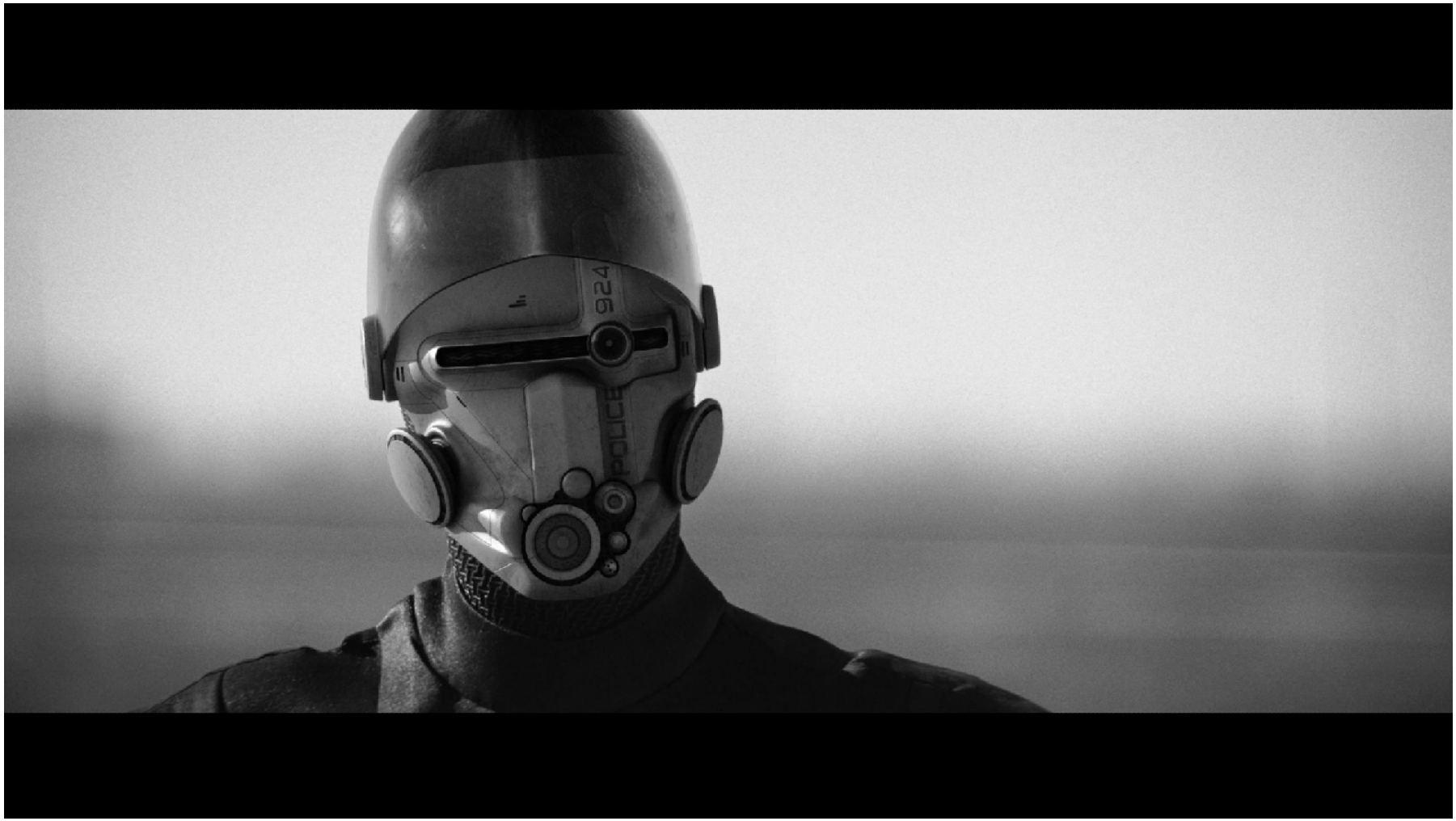

Figure 12: Image of Police Officer from *Star Trek (2009)* upscaled from 1280x720 to 1920x1080 using Bilinear Interpolation

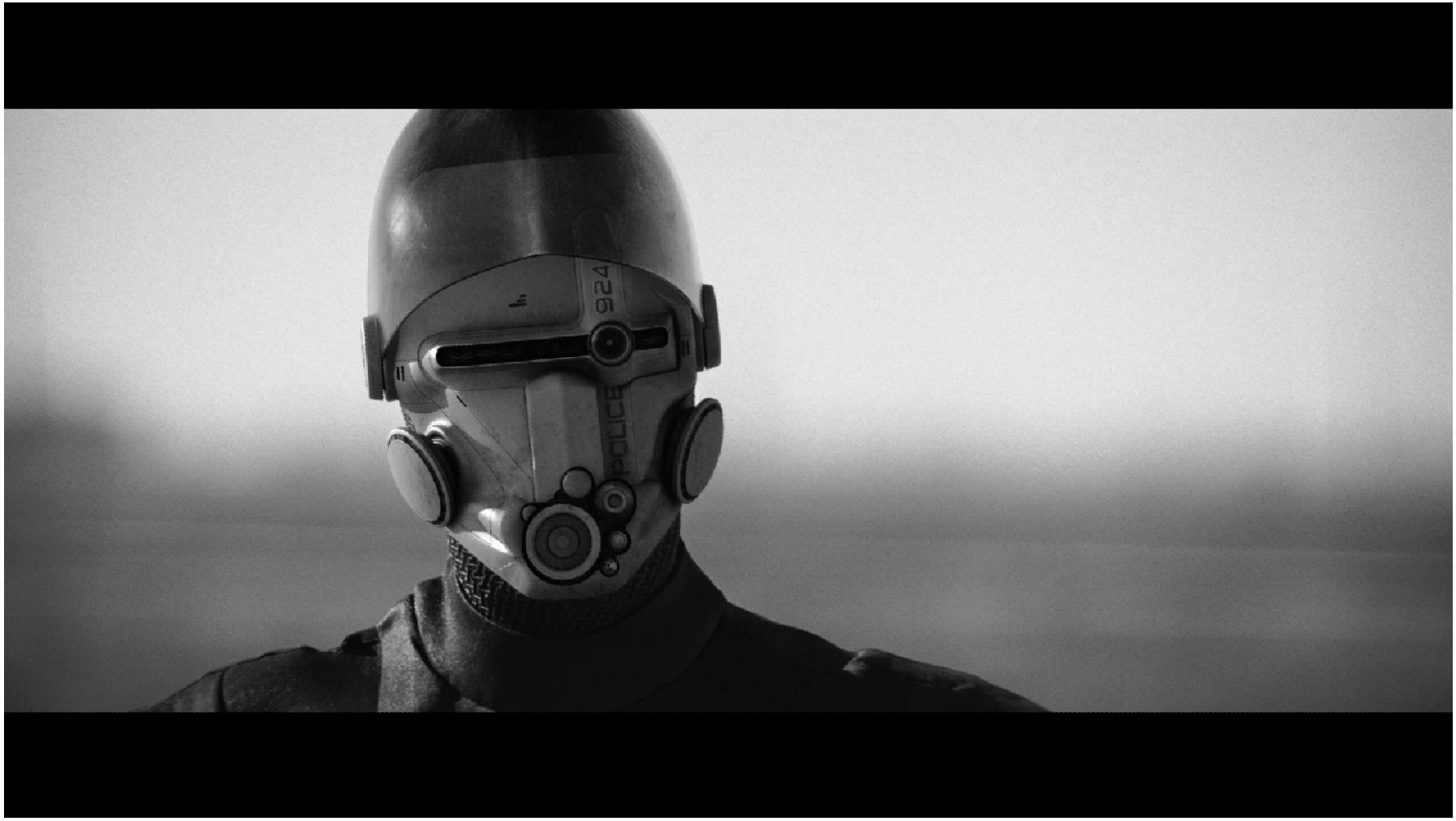

Figure 13: Image of Police Officer from *Star Trek (2009)* upscaled from 1280x720 to 1920x1080 using Nearest Neighbor Interpolation

Figure 14: Original 1920x1080 Image from *Star Wars: The Empire Strikes Back*

Figure 15: Image from *Star Wars: The Empire Strikes Back* Downscaled to 720x480

Figure 16: Image from *Star Wars: The Empire Strikes Back* Upscaled from 720x480 to 1280x720 using Fractal Interpolation

Figure 17: Image from *Star Wars: The Empire Strikes Back* Upscaled from 720x480 to 1280x720 using Bilinear Interpolation

Figure 18: Image from *Star Wars: The Empire Strikes Back* Upscaled from 720x480 to 1280x720 to 1920x1080 using Fractal Interpolation

Figure 19: Image of a Pig drawn by Harumi Chihiro upscaled to 4096x4096 using Fractal Interpolation

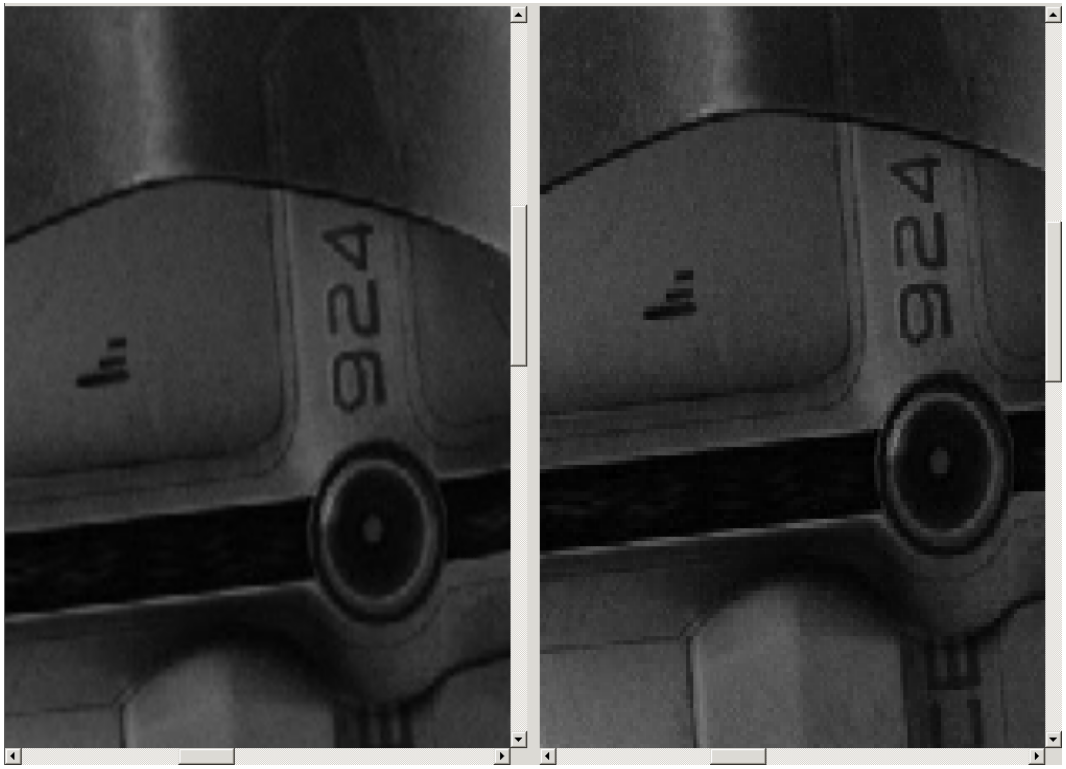

Figure 20: Comparison between bilinear and fractal interpolation of the same 1280x720 zoomed to 4K resolution. The right image was generated using fractal interpolation and the left was generated using nearest neighbor interpolation

B Source Code

```
public class MidpointInterpolationFilter : Filter {
  private bool allowSawtooth, allowGauss, allowVariance;
  private float k;
  private Random rnd;
////////////////////////////////////////////////////////////////////
  public MidpointInterpolationFilter(string name) : base(name) {
    rnd = new Random(Guid.NewGuid().ToString().GetHashCode());
  }
////////////////////////////////////////////////////////////////////
  public override byte[][] Transform(Hashtable input) {
    //reseed it
    if(input == null)
      return null;
    rnd = new Random(Guid.NewGuid().ToString().GetHashCode());
    byte[][] b = (byte[][])input["image"];
    int width = (int)input["width"];
    int height = (int)input["height"];
    allowSawtooth = (bool)input["sawtooth"];
    allowVariance = (bool)input["variance"];
    allowGauss = (bool)input["gauss"];
    ScaleInfo si = new ScaleInfo(b.Length, b[0].Length, width, height);
    if(si.IsZooming) {
      //compute k across the entire image
      int nWidth = si.ResultWidth;
      int nHeight = si.ResultHeight;
      //apply the original image values to the given points
      float w = si.WidthScalingFactor;
      float h = si.HeightScalingFactor;
      float?[][] newImage = new float?[nWidth][];
      float[][] displacementTable = new float[nWidth][];
```

```
float[][] imageF = ToFloatTable(b);
byte[][] resultant = new byte[si.ResultWidth][];
for(int i = 0; i < nWidth; i++) {
  newImage[i] = new float?[nHeight];
  displacementTable[i] = new float[nHeight];
  resultant[i] = new byte[nHeight];
}
for(int i = 0; i < nWidth - 1; i++) {
  int x0 = (int)((float)i / w);
  int x1 = (int)((float)(i + 1) / w);
  float p0 = (float)Math.Pow(x0 - x1, 2.0f);
  float[] line0 = imageF[x0];
  float[] line1 = imageF[x1];
  for(int j = 0; j < nHeight - 1; j++) {

    int y0 = (int)((float)j / h);
    int y1 = (int)((float)(j + 1) / h);
    //order has been messed up for sometime
    //should go (x0,y0), (x0,y1), (x1,y1), (x1,y0)
    float f0 = line0[y0];
    float f1 = line0[y1];
    float f2 = line1[y1];
    float f3 = line1[y0];
    k = ComputeK(2,i,j,b);
    displacementTable[i][j] = k; //save the base value
    float gauss = !allowGauss ? (float)rnd.NextDouble() : 1.0f;
    float var = !allowVariance ? Variance(f0,f1,f2,f3) : 1.0f;
    float tmp = (float)Math.Sqrt(1.0f - (float)Math.Pow(2.0f,
          2.0f * k - 2f)) * DeltaX(p0,y0,y1) * gauss * var;
    k = tmp;
    DivideGrid(resultant, newImage, i, j, w, h, f0, f1, f2, f3);
  }
```

```
}
if(allowSawtooth) {
  List<float> temp = new List<float>();
  for(int x = 0; x < si.ResultWidth; x++) {
    int outCount = 0;
    int pX = x - 1;
    bool pXValid = (pX >=0);
    if(pXValid) outCount++;
    int fX = x + 1;
    bool fXValid = (fX < si.ResultWidth);
    if(fXValid) outCount++;
    for(int y = 0; y < si.ResultHeight; y++) {
      if(newImage[x][y] == null || newImage[x][y] > 1.0f) {
        temp.Clear();
        int pY = y - 1;
        int fY = y + 1;
        int count = outCount;
        float v0 = 0f, v1 = 0f, v2 = 0f, v3 = 0f;
        //f-type computation
        if(pXValid) {
          v0 = ((float)newImage[pX][y]);
          temp.Add(v0);
        }
        if(pY >= 0) {
          v1 = ((float)newImage[x][pY]);
          temp.Add(v1);
          count++;
        }
        if(fXValid) {
          v2 = ((float)newImage[fX][y]);
          temp.Add(v2);
        }
```

```
            if(fY < si.ResultHeight) {
              v3 = ((float)newImage[x][fY]);
              temp.Add(v3);
              count++;
            }
            float total = (1.0f / (float)count);
            float p0 = total * (v0 + v1 + v2 + v3);

            float gauss = !allowGauss ? (float)rnd.NextDouble() : 1.0f;
            //get the k value
            float dX = DeltaX(pX,fX,pY,fY);
            float cK = displacementTable[x][y];
            float var = !allowVariance ? Variance(temp.ToArray()) : 1.0f;
            float p1 = (float)Math.Pow(2.0f, -(k / 2.0f));
            float p2 = (float)Math.Sqrt(1.0f - (float)Math.Pow(2.0, 2.0f * cK - 2.0f));
            float result = 255.0f * Normalize(p0 + p1 * p2 * dX * gauss * var);
            resultant[x][y] = (byte)result;
          }
        }
      }
      temp = null;
    }
    newImage = null;
    displacementTable = null;
    si = null;
    rnd = null;
    imageF = null;
    return resultant;
  }
  else {
    ByteImage image = new ByteImage(b);
    if(si.IsShrinking) {
```

```
        var _i = image.ReplicationScale(si.ResultWidth, si.ResultHeight);
        byte[][] target = new byte[_i.Width][];
        for(int i = 0; i < _i.Width; i++) {
          byte[] q = new byte[_i.Height];
          for(int j = 0; j < _i.Height; j++)
            target[i][j] = _i[i,j];
          target[i] = q;
        }
        image = null;
        return target;
      }
      else return b;
    }
  }
////////////////////////////////////////////////////////////////////
  public void DivideGrid(
      byte[][] result,
      float?[][] points,
      float x, float y,
      float width, float height,
      float c1, float c2, float c3, float c4) {
    if(x >= points.Length || y >= points[0].Length) return;
    float edge1, edge2, edge3, edge4, middle;
    float newWidth = width / 2.0f;
    float newHeight = height / 2.0f;
    if (width > 1.0f || height > 1.0f) {
      middle = Normalize(((c1 + c2 + c3 + c4) / 4.0f) + k);
      edge1 = Normalize((c1 + c2 ) / 2.0f);
      edge2 = Normalize((c2 + c3 ) / 2.0f);
      edge3 = Normalize((c3 + c4 ) / 2.0f);
      edge4 = Normalize((c4 + c1 ) / 2.0f);
```

```
    DivideGrid(result, points, x, y, newWidth,
        newHeight, c1, edge1, middle, edge4);
    DivideGrid(result, points, x + newWidth, y,
        width - newWidth, newHeight, edge1, c2,
        edge2, middle);
    DivideGrid(result, points, x + newWidth,
        y + newHeight, width - newWidth,
        height - newHeight, middle, edge2,
        c3, edge3);
    DivideGrid(result, points, x, y + newHeight,
        newWidth, height - newHeight, edge4,
        middle, edge3, c4);
  }
  else {
    int rX = (int)Math.Round(x);
    int rY = (int)Math.Round(y);
    if(rX >= points.Length || rY >= points[0].Length) return;
    if(points[rX][rY] == null) {

      var v = Normalize(((c1 + c2 + c3 + c4) / 4.0f) + k);
      points[rX][rY] = v;
      result[rX][rY] = (byte)(255.0f * v);
      if(allowSawtooth) {
        int q0 = rX + 1;
        int q1 = rX - 1;
        int q2 = rY + 1;
        int q3 = rY - 1;
        if(q0 < points.Length) points[q0][rY] = 2f;
        if(q2 < points[0].Length) points[rX][q2] = 2f;
        if(q1 >= 0) points[q1][rY] = 2f;
        if(q3 >= 0) points[rX][q3] = 2f;
      }
```

```
      }
    }
  }
////////////////////////////////////////////////////////////////////
  public override Hashtable TranslateData(Hashtable input) {
    try {
      input["width"] = int.Parse((string)input["width"]);
      input["height"] = int.Parse((string)input["height"]);
    }
    catch(Exception) { return null; }
    return input;
  }
////////////////////////////////////////////////////////////////////
  public float ComputeK(int n, int startX, int startY, byte[][] image) {
    int total0 = 0;
    for(int j = 0; j < (n - 1); j++) {
      for(int i = 0; i < (n - 2); i++) {
        total0 += Math.Abs(image[startX + (i + 1)][startY + j]
            - image[startX + i][startY + j]);
      }
    }
    int total1 = 0;
    for(int i = 0; i < (n - 2); i++) {
      for(int j = 0; j < (n - 1); j++) {
        total1 += Math.Abs(image[startX + i][startY + (j + 1)]
            - image[startX + i][startY + j]);
      }
    }
    float numerator = total0 + total1;
    float denominator = 2.0f * (float)(Math.Pow(255, (float)n));
    return (numerator / denominator);
  }
```

```
////////////////////////////////////////////////////////////////////
  public float Variance(params float[] f) {
    float n = 0f;
    float mean = 0f;
    float m2 = 0f;
    foreach(var x in f) {
      n = n + 1f;
      float delta = x - mean;
      mean = mean + delta/n;
      m2 = m2 + delta*(x - mean);
    }
    return m2 / (n - 1);
  }
////////////////////////////////////////////////////////////////////
  public float DeltaX(float x0, float x1, float y0, float y1) {
    return DeltaX((float)Math.Pow(x0 - x1, 2.0f), y0, y1);
  }
////////////////////////////////////////////////////////////////////
  public float DeltaX(float p0, float y0, float y1) {
    return (float)Math.Abs((float)Math.Sqrt(p0 +
          (float)Math.Pow(y0 - y1, 2.0f)));
  }
////////////////////////////////////////////////////////////////////
  public static float[][] ToFloatTable(byte[][] items) {
    float[][] resultTable = new float[items.Length][];
    for(int i = 0; i < resultTable.Length; i++) {
      resultTable[i] = new float[items[i].Length];
      for(int j = 0; j < resultTable[i].Length; j++)
        resultTable[i][j] = ((float)items[i][j]) / 255.0f;
    }
    return resultTable;
  }
```

```
//////////////////////////////////////////////////////////////////////
  private float Normalize(float iNum)  {
    if(iNum < 0) return 0.0f;
    else if(iNum > 1.0f) return 1.0f;
    return iNum;
  }
}
```

References

[1] H. Honda, M. Haseyama, and H. Kitajima. Fractal interpolation for natural images. In *Image Processing, 1999. ICIP 99. Proceedings. 1999 International Conference on*, volume 3, pages 657 –661 vol.3, 1999.

[2] Jeffery R. Price and Monson H. Hayes III. Resampling and reconstruction with fractal interpolation functions. 5(9):228 –230, September 1998.

[3] Justin Seyster. Plasma fractal. Technical report, 2002. URL is http://www.ic.sunysb.edu/Stu/jseyster/plasma/.

[4] Zaifeng Shi, Suying Yao, Bin Li, and Qingjie Cao. A novel image interpolation technique based on fractal theory. In *Computer Science and Information Technology, 2008. ICCSIT '08. International Conference on*, pages 472 –475, 29 2008-sept. 2 2008.