CS231A Course Project: GPU Accelerated Image Super-Resolution

Fei Yue
Stanford University
450 Serra Mall Stanford, CA
jessyue@stanford.edu

Abstract

This paper proposes a GPU based real-time approach for image super-resolution. This approach is based on the fact that human observers are insensitive to single pixel shifts as long as it is globally coherent. The approach performs dilation or erosion on the pixel level depending on the local context. We do this by shifting the center pixel in the same the direction of the gradient if the color is bright, and in the opposite direction if the color is dark. Then, a series of contour aware high-pass and low-pass filtering operations are performed to retain higher sharpness and details. This approach attempts to eliminate common artifacts produced in traditional super-resolution algorithms, while maintaining a high performance for real-time processing. The result is very promising: it performs on par with state-of-art super-resolution algorithms, in the absence of the complicated learning stage. As a result, this is an excellent direction to take for super-resolution in real-time processing.

1. Introduction

Super-resolution (SR) is an operation that estimates a high-resolution output given a low-resolution input image. SR is useful for many purposes. In desktop applications, image resizing is essential in image editing software and desktop publishing. In mobile devices, it could be used to enhance image capture quality. In today's high-definition television era, SR is prominent in the enhancement of standard-definition video quality. SR of still images can be broadly applied to many areas of graphics. Specifically, it could be used to increase texture and image quality in applications or video games with low resolution. This paper seeks to find a solution of SR fitted in this context.

A challenging problem, SR has received much attention from both the image processing as well as the graphics research community, generating many different types of solutions. However, none of these yield a good solution for the GPU to be used in a real-time graphical context, as they would have excessive run time. This effort seeks a good approach of super-resolution to be used for real-time video game.

More specifically, this effort seeks to find a solution for real-time GPU enlargement of digital photographs to 2x width and 2x height. The solution would be a single-image super resolution of 4x the area, which could be used in video game textures after they are loaded into the video memory. In practice, textures are usually stored in small resolution to save storage space and transfer time for online-games. The enlargement factor of 4x is chosen because anything beyond 4x the area would consume too much video memory in this application.

Leveraging the fact that human observers are relatively insensitive to single pixel shifts as long as the adjustment is coherent with the global picture, the general approach here is to dilate (increase contrast on) large bright areas, and erode dark areas to obtain thin and sharp features. Since the GPU is highly parallel, these operations could be accomplished quickly on a per pixel basis.

2. Related Work

The most rudimentary class of approach to solve the SR problem is via data-invariant filters such as nearest-neighbor, bilinear, and cubic spline [1]. These methods are implemented in a lot of commercial software due to their simplicity, but they tend to produce visual artifacts such as blurring, ringing and aliasing.

Another family of approach in solving SR is the classical multi-image methods [2, 3, 4, 5]. A high-resolution image is calculated from multiple low-resolution images of the same scene, with sub-pixel shifts. The pixels of the low-resolution images form a linear combination. The system of linear equations can solve the pixel value of the high-resolution image. However, in many cases, low-resolution images of the same scene are not always available in multiple copies with slight shifts.

Example-based SR is another class of methods [6, 7, 8, 9, 10], which is a learning algorithm that draws

correspondence from a database of low-resolution and high-resolution image pairs. The learned model is then applied to new low-resolution images to estimate plausible high-frequency details in the high-resolution version. Example-based SR algorithms preserve fine details, but the result high-resolution image is not globally consistent. Also, sometimes the generated textures are different from the source.

Kim et al. [11] produces a single-image SR algorithm by extending the example-based method. The algorithm learns the mapping from low-resolution images to high-resolution images based on suitable pairs of example images. The problem is posed as a kernel ridge regression, which produces candidate images that reflect different local information. An output is then produced as a convex combination of the candidate images. Lastly, a prior model of a generic image class is considered to correct the errors caused at the learning stage.

Glasner et al. [12] accomplishes super-resolution from a single image by combining the classical multi-image super-resolution with example-based super-resolution to obtain SR from a single low-resolution image. Glasner claims that patches in a natural image tend to repeat itself many times in the same image. The repetition could be of the same scale as the original patch, or of different scales. In the case of the repetition of the same scale, the classical SR approach could be applied to combine images obtained at sub-pixel misalignment. The repetition of different scales provides enough information of low- and high-resolution pairs, which can be used to form a learning model for the example-based super-resolution approach. This algorithm inherits some of the drawbacks of traditional example-based algorithms, which is the inconsistency in quality, incorrect hallucination, and adding features that do not exist in source images.

An alternative direction for super-resolution is to up-scale an image while maintaining edge details and sharpness. Fattal [13] proposes such an approach based on a statistical edge dependency, which relates edge features of different resolutions. The weakness of this algorithm is that it often produces mosaic-like images, shown in Figure 1. The details become lost and the texture becomes unbelievable.

Figure 1: Example of a Mosaic-like SR Image [13]

Sun et al. [14] proposes another algorithm that emphasizes on preserving edge details and picture sharpness using a gradient profile prior. The gradient profile prior is learned from a dataset, and then applied to new test images. This paper also uses a gradient approach to preserve edge details. However, we accomplish similar results as Sun et al. without the complexity of the learning algorithm.

Overall, the existing algorithms have the following problems:

- Inconsistency in quality
- Adding features that do not exist
- Incorrect hallucination
- Transforming texture into something not similar to the source
- Blurriness

This effort attempts to address the above issues under the real-time performance constraint.

3. Approach

The algorithm takes a two-pass approach. In the first pass, the color is converted into non-perspective weighted intensity values first using (1).

$$I = \frac{(red + green + blue)}{3}, \quad (1)$$

Then, the algorithm generates the gradient and high pass textures. The gradient components are generated via the x-direction and y-direction kernels described in (2) and (3) respectively. The tangent to the gradient direction is the contour line of the pixel.

$$\frac{1}{2}\begin{bmatrix} -0.5 & 0 & 0.5 \\ -1 & 0 & 1 \\ -0.5 & 0 & 0.5 \end{bmatrix} \quad (2)$$

$$\frac{1}{2}\begin{bmatrix} -0.5 & -1 & -0.5 \\ 0 & 0 & 0 \\ 0.5 & 1 & 0.5 \end{bmatrix} \quad (3)$$

The high-pass texture is obtained by first getting a low-pass texture via kernel (4), then with equation (5), where I is the pixel intensity value. The high-pass texture will be used in the next pass to recover details lost by edge-enhancement operations in the perpendicular direction.

Figure 2: A progression of the algorithm. a) Step 1: Nearest-neighbor sampling b) Step 2: Applied gradient high-pass filter, removes checkerboard pattern c) Step 3: Applied gradient shift, more sharpness and details in hair and eyes d) Step 4: Applied high-pass texture, more overall sharpness

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix} \quad (4)$$

$$highPass = I - lowPass \quad (5)$$

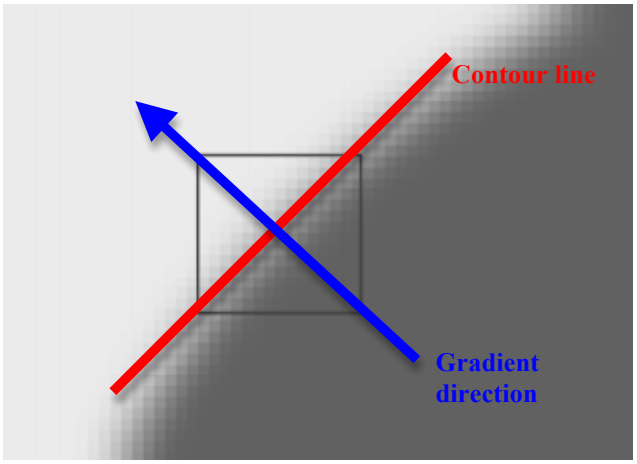

Figure 3: Contour Line and Gradient Direction

Figure 2 entails the process of the second pass. In this pass, the following steps are taken. Along the contour line (figure 3), three intensity values are obtained which are 0.5 source pixels apart. The high-pass filter is taken with kernel (6) on the three intensity values.

$$\begin{bmatrix}0.625 & -0.25 & 0.625\end{bmatrix} \quad (6)$$

This removes the staircase effect along edges and increases sharpness along the contour. We use it as our base result. However, details along the direction perpendicular to the contour line are reduced by this operation. To counteract this effect, we source the details from the high-passed texture obtained in step one. We take three samples, 0.125 to 0.5 source pixels apart, tangent to the contour from the high-passed texture. A low-pass operation (7) is applied to these values, which filters out the high-frequency edges in the direction of the contour, since the base result already increases sharpness in that direction. Filtering the high-pass samples in the direction of the gradient also helps to remove bilinear filtering artifact.

$$\begin{bmatrix}0.25 & 0.5 & 0.25\end{bmatrix} \quad (7)$$

We add the filtered high-pass to the base color to get the final result.

We modify the above technique to increase the perception of fine details by adjusting the contours in the image. We apply an erosion feature to dark features to thin them and a dilation feature to light features to expand them [15]. This is accomplished by adjusting the base sampled position and then applying the above algorithm to the new shifted base sample position. Dark colors are shifted towards the negative direction of the gradient. Light colors are shifted towards the positive direction of the gradient, and medium colors remain un-shifted.

4. Experiments

A set of test images, described in section 4.1, are used as input images for the algorithm. The algorithm outputs up-sampled images that are two times the height and two times the width of the original. We hand-tuned the high-pass filters, (6, 7) for individual images to obtain best results. We then pass on the resulting image set, along

with two benchmarks, bilinear and Glasner, to five human test subjects to study independently of each other. These test subjects' background in image processing and graphics range from none to expert. Each test subject is given a test survey that contains evaluation criteria described in section 4.2. The test subject ranks all three different algorithms from low to high on each criterion.

The reason for the subjective qualitative study is because the objective of this effort is not to "correctly" reproduce a source image that is down-sampled and then up-sampled via a super-resolution algorithm. Rather, we want to produce an image that is visually pleasing to the human eye.

Another objective of the effort is to produce an algorithm that has a high quality to performance trade-off ratio. As such, section 4.3 details our performance analysis.

4.1. Dataset

Since the algorithm does not require any training, only a set of low-resolution source images are used as input. We take 26 test images from Glasner and 10 images from photo.net with the following features:

- Natural images
- Human faces
- Real-world man-made objects
- Computer-generated/synthetic objects
- Plants
- Animals with interesting fur/texture/pattern
- Images with repeating natural patterns
- Images with repeating man-made patterns
- Images with high frequencies
- Images with aliasing

The above list comprehensively covers all cases one would encounter in real-time video games as well as most other general-purpose scenarios. This dataset allows direct comparison with the results from our algorithm and Glasner. Specifically, Glasner performs well in images with dominant natural and man-made patterns. It is interesting to see how our algorithm matches their results.

4.2. Evaluation Criteria

Qualitatively, visual inspection is used to evaluate the algorithm. The following guidelines are used for visual evaluation. First, the up-sampled image must contain believable textures that are similar to the source, and limit hallucination artifacts – any added detail cannot stand out from the detail in the photograph or look out of place to a typical observer. For example, the technique should not introduce waviness on straight edges. Second, match perceptual sharpness of source image. Third, be devoid of linear or cubic up-sampling artifacts. Fourth, the quality of the image must be consistent throughout the entire picture. Last, no loss in details.

Since the algorithm is constrained to run in real-time, performance is an important factor. The performance of the algorithm is gauged by the number of pixel reads from the source image, since this is the slowest part out of all the GPU operations. A discussion and comparison against Glasner's algorithm is also shown in section 4.4.

4.3. Qualitative Evaluation

The result of the experimental survey are processed and tabulated in Table 1.

Metrics	Bilinear	Glasner	Yue
1.Believability (high is desirable)	High	Medium	High
2.Sharpness (high is desirable)	Low	High	Medium
3.Up-sampling artifacts (low is desirable)	High	Low	Low
4.Consistency of quality (high is desirable)	High	Low	High
5.Loss of details (low is desirable)	High	Low	Low

Table 1: Texture Fetches Per Pixel

Overall, our method is superior to bilinear in all the criteria, and also out-performs Glasner in the criteria 1 and 4.

In addition, five sets of images are included in this paper. Figure 4 is a good representation of a face example. Glasner's algorithm adds additional texture in the nose which clearly does not belong. Fattal's algorithm generates a pastel-like image which is unnatural. Our algorithm, in comparison, gives a sharp image that is very believable in human perception. Figure 5 is a representative image in the animal class with natural patterns. In this picture, the Glasner's patch-based algorithm is defective because not all the pixels of the image are up-sampled with the same quality. The patch with dominating zebra patterns (marked i) is very sharp. However, the zebra's head (marked ii) and the grass area (marked iii) are of very low quality. In comparison, our algorithm is consistent throughout. Figure 6 shows a defect in our algorithm. For input images that have aliasing, the output image also has aliasing. Glasner's algorithm handles aliased images better. Figure 7 shows an over-sharpening case of our algorithm. Although we produce a very crisp result, the plant background (marked in i) is over-sharpened. Lastly, figure 8 shows a failed case of our algorithm compared to bilinear. The image has much high frequency, and our algorithm seems to handle the texture in the lower right corner (marked in i) very poorly. It also does not perform much better than bilinear in overall sharpness.

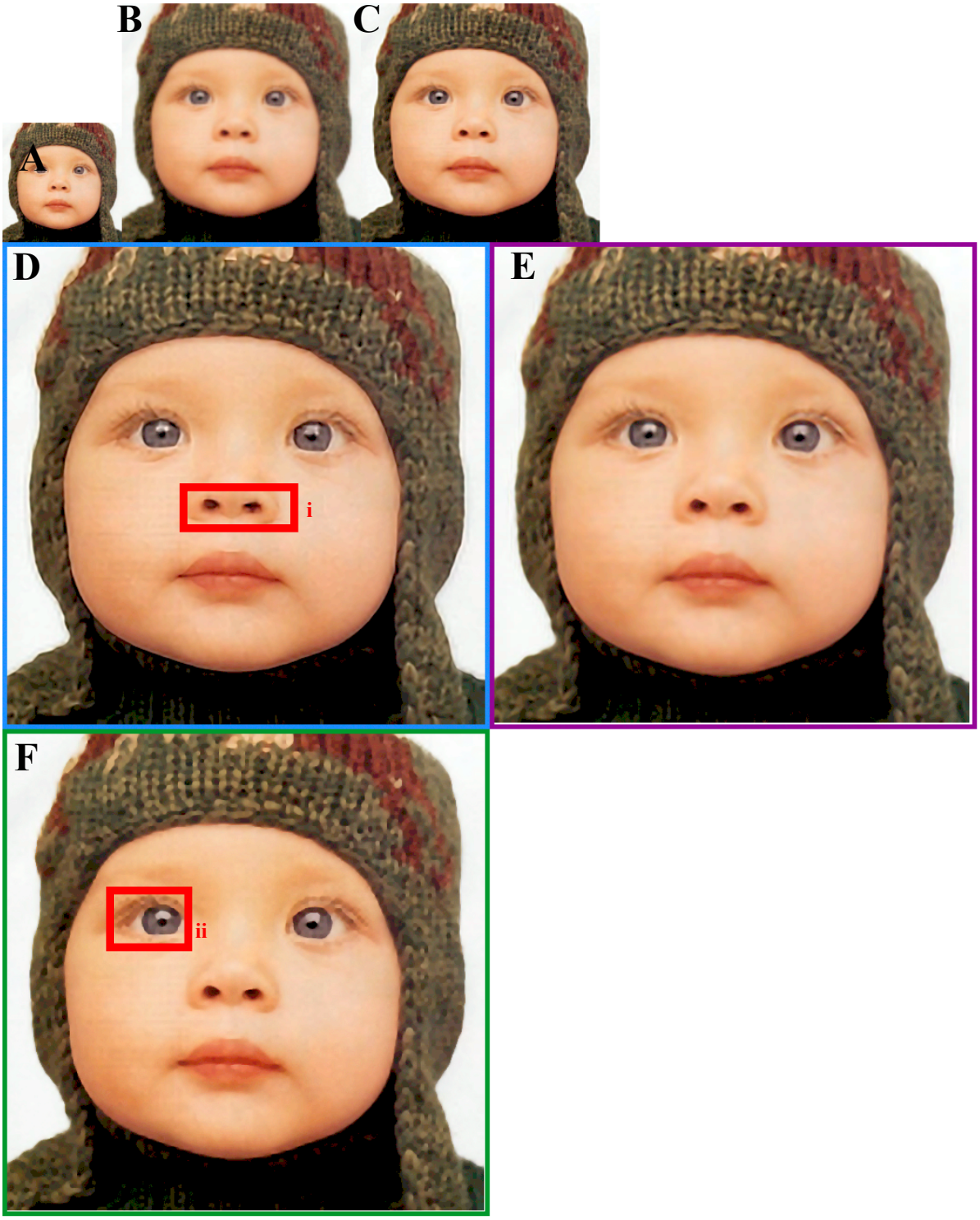

Figure 4: Baby (face) a) Original Image (1x) b) Bilinear (2x): lots of blurriness c) Yue (2x): sharp, realistic image d) Glasner (3x): adding features inside the nose, shown by (i) which should not be there. e) Kim (3x): sharp, realistic image f)Fattal (3x): pastel-like mosaic patterns shown by (ii)

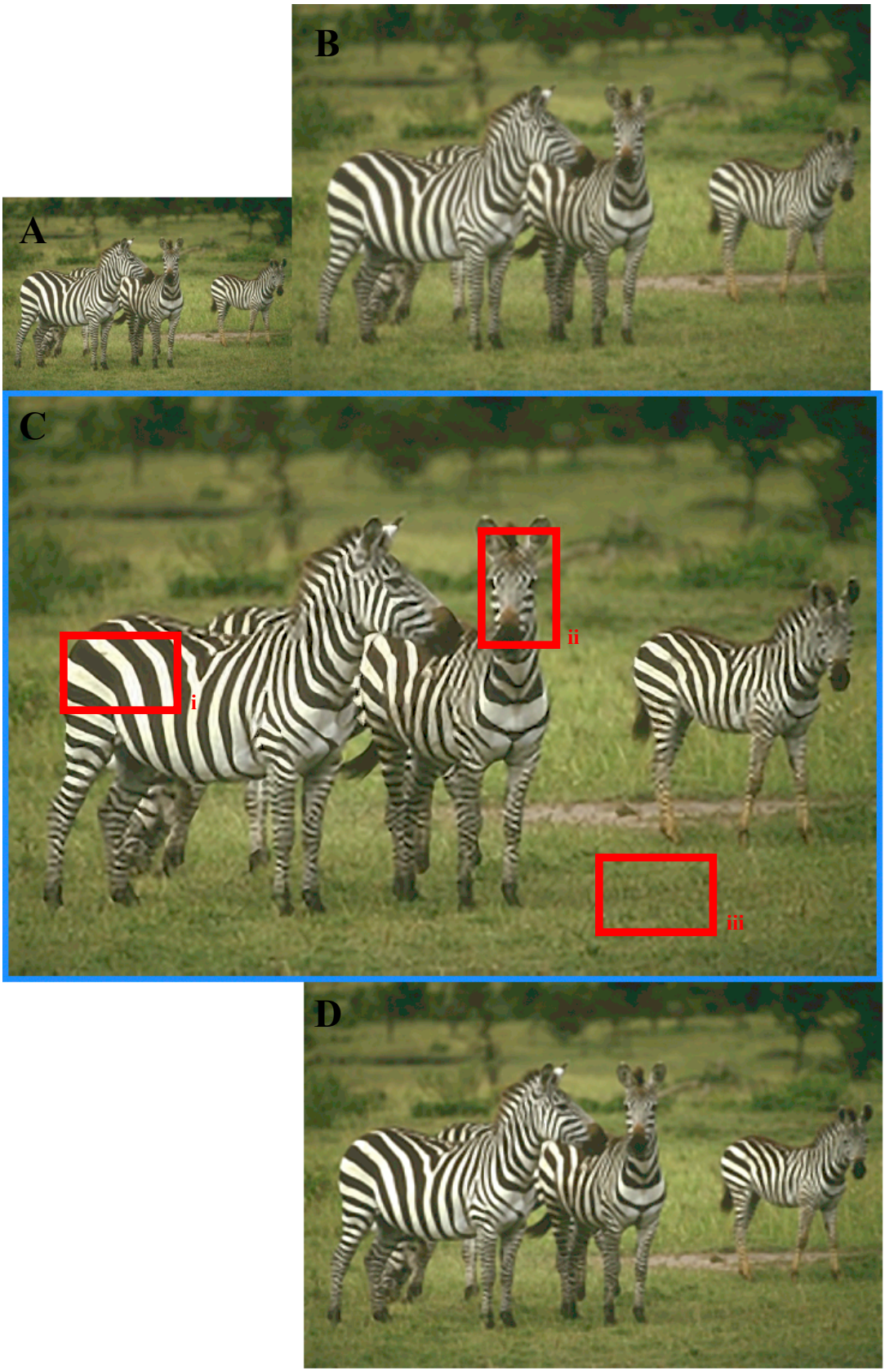

Figure 5: Zebra (animal, natural pattern) a) Original image (1x) b) Bilinear (2x): lots of blurriness c) Glasner (3x): inconsistent quality shown by (i) very clear, and (ii), (iii) very blurry d) Yue (2x): clear, consistent image

Figure 6: Kitchen (man-made pattern) a) Original image (1x) b) Bilinear (2x): lots of blurriness c) Yue (2x): clear, but floor pattern contains aliasing shown by (i) d) Glasner (3x): clear, less aliasing

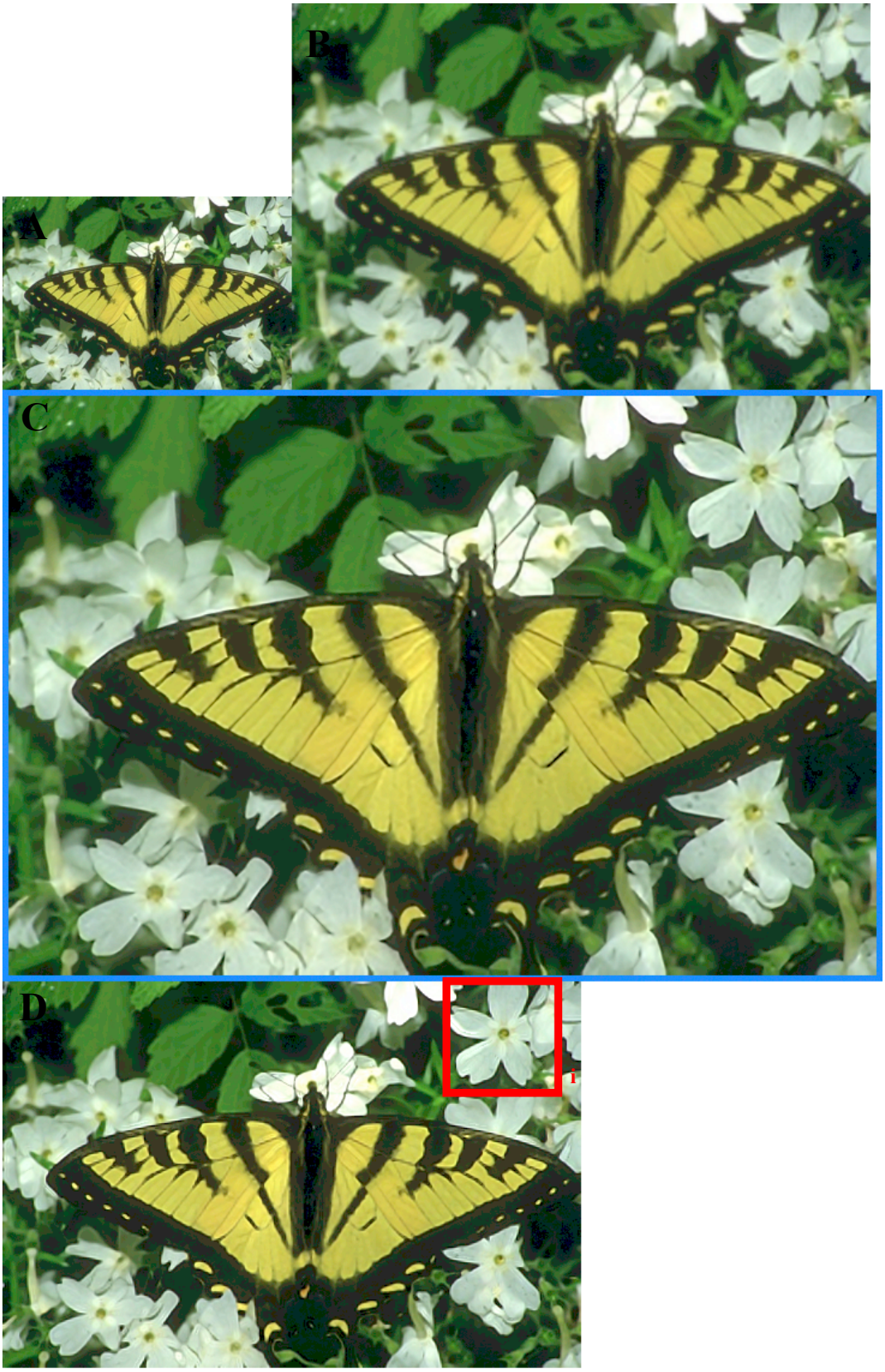

Figure 7: Butterfly (plant, animal) a) Original image (1x) b) Bilinear (2x): lots of blurriness c) Glasner (3x): clear image d) Yue (2x): clear image, but (i) is over-sharpened with ringing

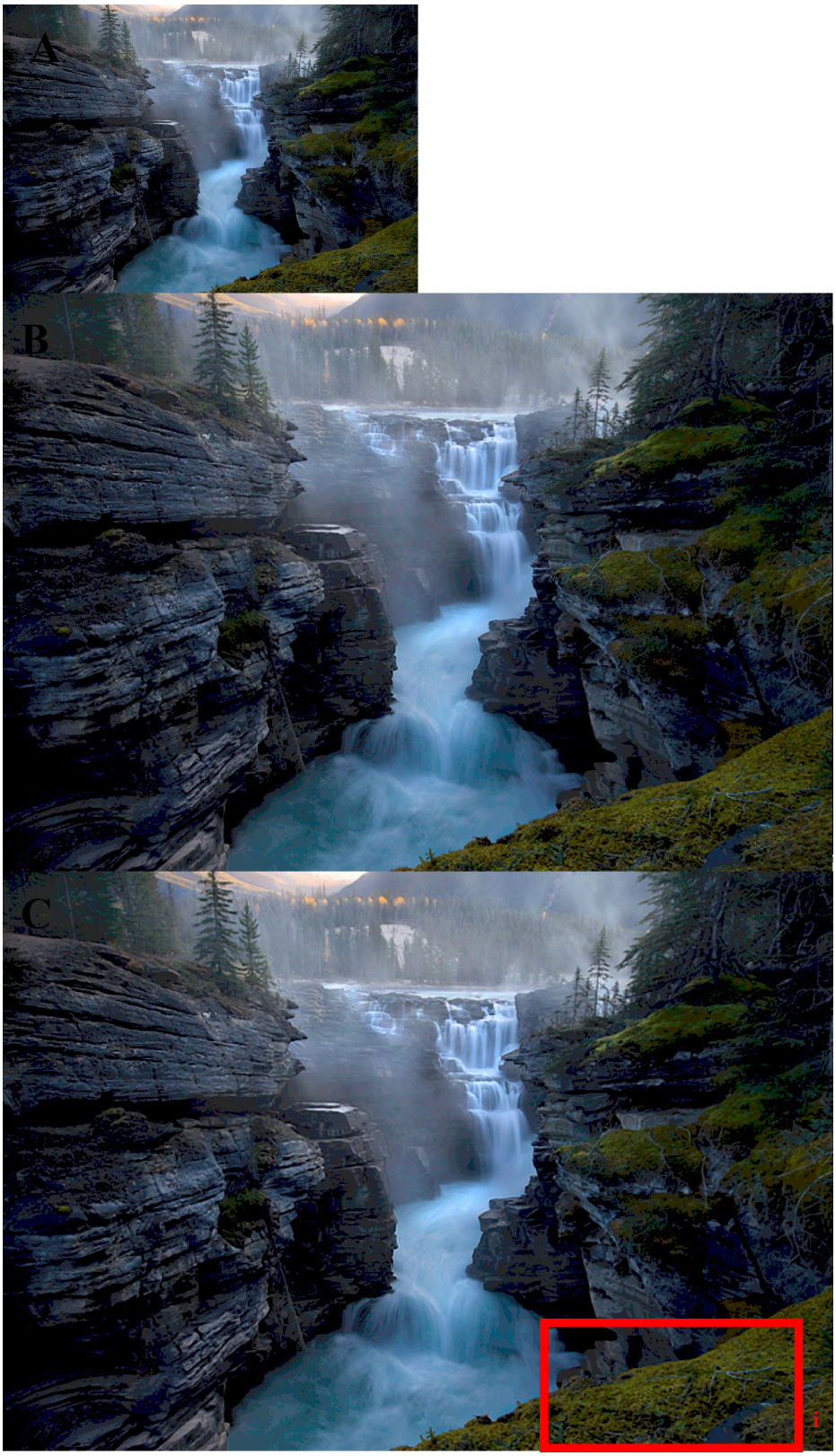

Figure 8: Waterfall (high-frequency image, courtesy of photo.net) a) Original image (1x) b) Bilinear (2x) c) Yue (2x): not significantly better than bilinear other than slightly higher contrast. Does not handle high frequency area (i) well.

4.4. Performance Evaluation

Our algorithm's run-time cost is linear (O(n)), with practical running time bounded by a fixed number of texture fetches per source image pixel and output image pixel. The pixel fetches are tabulated in Table 2.

In comparison to most learning-based algorithms, our algorithm is much faster. Taking Glasner et al. for example, the algorithm's performance is bottlenecked by the nearest neighbor search [16], which uses a 5x5 patch per pixel, and find k = 9 nearest neighbor patches. Disregarding the multi-resolutions aspect of the paper, the cost of building a single search structure is:

$$n = width \times height$$

Therefore, the run time is proportional to:

$$O(n) = width^2 \times height^2$$

This makes the algorithm not usable in real-time, and not comparable to our GPU based method.

	Number of Fetches
Source pixels	9
High-pass texture	3
Color	3
Gradient texture	1
Total	**16**

Table 2: Texture Fetches Per Pixel

5. Conclusion

Overall, the approach described in this paper generates promising results that could be used in real-time video game up-sampling. The quality to performance ratio of our algorithm is superior to all other existing SU algorithms. The logical next step is to replace the hand-tuned parameters with an adaptive algorithm that is suitable for each image. We also need to seek better ways for controlling dilation and erosion based on local relative intensity instead of global intensity. We also learned that the algorithm currently does not perform well for images with a lot of high frequencies and aliasing. However, this can probably be corrected by adapting the filter width along the gradient and contour. As well, we need to improve the algorithm to better preserve details by limiting filtering in areas which exhibit noise.

6. References

[1] H. S. Hou and H. C. Andrews. Cubic splines for image interpolation nand digital filtering. IEEE Trans. on SP, 26(6):508–517, 1978.

[2] M. Irani and S. Peleg. Super Resolution From Image Sequences, ICPR, 2:115—120, June 1990.

[3] M. Irani and S. Peleg. Improving resolution by image registration. CVGIP, (3), 1991.

[4] D. Capel. Image Mosaicing and Super-Resolution. Springer–Verlag, 2004

[5] S. Farsiu, M. Robinson, M. Elad, and P. Milanfar. Fast and robust multi frame super resolution. T-IP, (10), 2004.

[6] S. Baker and T. Kanade. Hallucinating faces. In Automatic Face and Gesture Recognition, 2000.

[7] W. T. Freeman, T. R. Jones, and E. C. Pasztor. Example based super-resolution. Comp. Graph. Appl., (2), 2002.

[8] W. Freeman, E. Pasztor, and O. Carmichael. Learning low level vision. IJCV, (1), 2000.

[9] J. Sun and M. F. Tappen. Context-Constrained Hallucination for Image Super-Resolution. IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2010).

[10] Y. Ha Cohen, R. Fattal, D. Lischinski. Image Upsampling via Texture Hallucination. IEEE International Conference on Computational Photography (ICCP 2010).

[11] K. Kim and Y. Kwon. Example-based learning for single image SR and JPEG artifact removal. MPI-TR, (173), 08.

[12] D. Glasner, S. Bagon, M Irani. Super-Resolution From a Single Image, ICCV 2009.

[13] R. Fattal. Image upsampling via imposed edge statistics. In SIGGRAPH, 2007.

[14] J. Sun, Z. Xu, and H. Shum. Image super-resolution using gradient profile prior. In CVPR, 2008

[15] T. Lottes, Personal Communication. November, 2011.

[16] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In SODA, 199

7. Supplementary Materials

```
//
//
// DOUBLE IMAGE SIZE SHADER GLSL SOURCE FOR OPENGL
//
//


//
// NON-PERCEPTUAL CONVERSION OF COLOR TO INTENSITY
//
float ToIntensity(vec4 pix) { return (pix.r + pix.g + pix.b) / 3.0; }


//
// PASS A
//
// Generate gradient texture and highpass texture.
// Output textures same size as input color texture.
//
void PixelShaderPassA(
 out vec4      dir, // output gradient direction
 out vec4      hi,  // output high pass
 in  sampler2D tex, // input texture
 in  vec2      pos  // pixel position in texture
) {


 // 3x3 pixels around the pixel
 //   nw nm ne
```

```
//   mw mm me
//   sw sm se
vec4 nw = textureLodOffset(tex, pos, 0.0, ivec2(-1, -1));
vec4 nm = textureLodOffset(tex, pos, 0.0, ivec2( 0, -1));
vec4 ne = textureLodOffset(tex, pos, 0.0, ivec2( 1, -1));
vec4 mw = textureLodOffset(tex, pos, 0.0, ivec2(-1,  0));
vec4 mm = textureLodOffset(tex, pos, 0.0, ivec2( 0,  0));
vec4 me = textureLodOffset(tex, pos, 0.0, ivec2( 1,  0));
vec4 sw = textureLodOffset(tex, pos, 0.0, ivec2(-1,  1));
vec4 sm = textureLodOffset(tex, pos, 0.0, ivec2( 0,  1));
vec4 se = textureLodOffset(tex, pos, 0.0, ivec2( 1,  1));

// convert into intensity
float nwI = ToIntensity(nw);
float nmI = ToIntensity(nm);
float neI = ToIntensity(ne);
float mwI = ToIntensity(mw);
float mmI = ToIntensity(mm);
float meI = ToIntensity(me);
float swI = ToIntensity(sw);
float smI = ToIntensity(sm);
float seI = ToIntensity(se);

// filter weights
float w1 = 1.0;
float w2 = 0.5;

// output gradient direction
```

```
  dir.x = ((-w2 * nwI) + (-w1 * mwI) + (-w2 * swI) + (w2 * neI) + (w1 * meI) + (w2 * seI)) / (w2+w1+w2);
  dir.y = ((-w2 * nwI) + (-w1 * nmI) + (-w2 * neI) + (w2 * swI) + (w1 * smI) + (w2 * seI)) / (w2+w1+w2);

  // convert from {-1,1} to {0,1} for output
  dir.xy = dir.xy * 0.5 + 0.5;

  // lowpass filter weights
  float ww0 = 4.0;
  float ww1 = 2.0;
  float ww2 = 1.0;

  // lowpass
  vec4 lo = vec4(0.0);
  lo += (mm * ww0);
  lo += (nm + mw + me + sm) * ww1;
  lo += (nw + ne + sw + se) * ww2;
  lo /= (ww0*1.0 + ww1*4.0 + ww2*4.0);

  // highpass
  hi = mm - lo;

  // convert from {-1,1} to {0,1} for output
  hi = hi * 0.5 + 0.5;
}

//
// PASS B
```

```
//
// Generates enlarged output.
// Takes as input the non-enlarged texture, and output from pass B.
//
void PixelShaderPassB(
  out vec4      result, // output color in enlarged surface
  in  sampler2D texCol, // input color texture
  in  sampler2D texDir, // input gradient texture
  in  sampler2D texHi,  // input highpass texture
  in  vec2      pos,    // pixel position in output surface
  in  vec2      pix     // size of non-enlarged source pixel {1.0/imageWidthInPixels, 1.0/imageHeightInPixels}
) {

  // fetch gradient direction
  vec2 dir = textureLod(texDir, pos, 0.0).xy;
  dir = dir * 2.0 - 1.0;

  // normalize the gradient direction
  dir.xy *= 1.0/(sqrt(dot(dir,dir)) + (1.0/65536.0));

  // contour vector is 90 deg to gradient
  vec2 contour = dir.yx * vec2(1.0,-1.0);

  // fetch color from this output pixel position
  vec4 color = textureLod(texCol, pos.xy, 0.0);
  float colorI = ToIntensity(color);

  // transform intensity into amount to shift in direction of gradient
```

```
// dark colors shift towards the negative gradient
// middle colors have no shift
// light colors shift towards the positive gradient
// maximum shift is 0.75 pixels (depending on image content)
// units are in source image pixels
float shift = (colorI * 2.0 - 1.0) * (0.5);

// shifted position in image
vec2 shifted = pos + dir * shift * pix;

// fetch from three positions along shifted contour
// spacing of fetches is 0.5 source pixels
vec4 colorA = textureLod(texCol, shifted - (contour * pix * 0.5), 0.0);
vec4 colorB = textureLod(texCol, shifted, 0.0);
vec4 colorC = textureLod(texCol, shifted + (contour * pix * 0.5), 0.0);

// base of result is a highpass along this shifted contour
// this increases sharpness along the contour
result = colorB * (-0.25) + (colorA + colorC) * (0.5 * 1.25);

// fetch from three positions along shifted gradient direction
// spacing of fetches is 0.125 source pixels
// sampling in this case from the highpass texture
vec4 hiA = textureLod(texHi, shifted - dir * pix * 0.125, 0.0);
vec4 hiB = textureLod(texHi, shifted, 0.0);
vec4 hiC = textureLod(texHi, shifted + dir * pix * 0.125, 0.0);
```

```
  // take the lowpass of these three values
  // this filters out high frequency edges which are in the direction of the contour
  // the base result already increased sharpness in that direction
  // leaving edges in the direction of the gradient
  vec4 hiL = hiB * (0.5) + (hiA + hiC) * (0.25);

  // convert highpass from {0,1} to {-1,1}
  vec4 hi = hiL * 2.0 - 1.0;

  // add to result the filtered highpass
  result += hi;
}
```