10 Image Processing

The addition of the compute shader to Direct3D 11 is arguably one of its most interesting new features. The compute shader provides a distinctly different technique for using the GPU hardware, and there are many new algorithms that can be implemented with these new abilities. One area in particular that can benefit from the compute shader is the image processing domain. Since the GPU's roots are in image generation and manipulation, it is a natural extension to include general image processing capabilities in the realm of available algorithms. Figure 10.1 demonstrates the results of a blurring filter applied to half of an image.

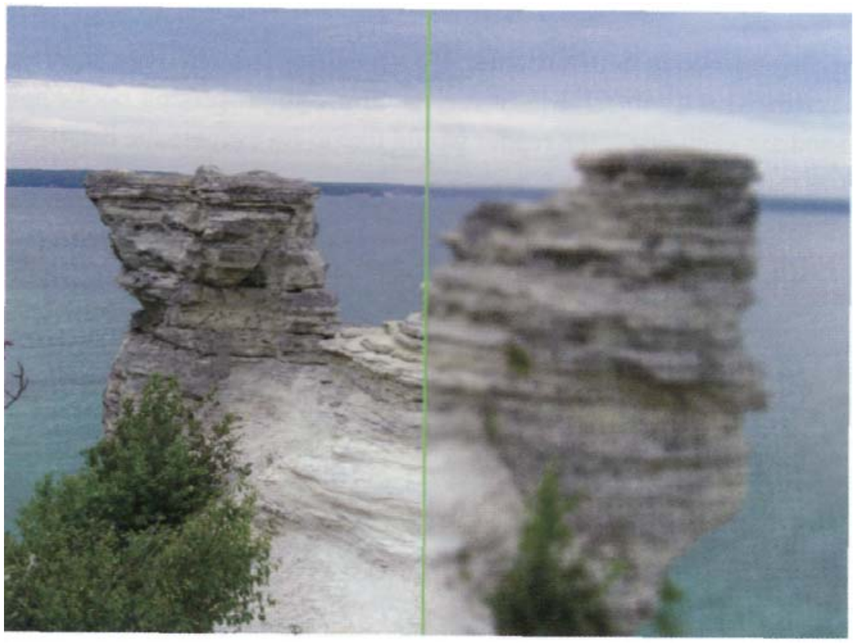

Figure 10.1. An example image with a blurring filter applied to its right side.

In fact, the compute shader threading model is easily mapped onto an image domain by using the *x* and *y* global thread addresses to index the image's pixels (the threading model itself is covered in detail in Chapter 5). This makes it simple to develop an implementation that can use the massively parallel processing capabilities of modern CPUs, and that can provide a fairly significant performance improvement over traditional CPU implementations. In addition, the compute shader provides HLSL atomic operations, group shared memory, device memory resources, and synchronization intrinsic functions to facilitate communication between threads. If these additional tools are used in an appropriate algorithm, even further performance increases can be achieved.

This chapter provides several sample image processing algorithms that attempt to take advantage of the compute shader stage's capabilities. The first algorithm that we inspect is the well-known *Gaussian blur* operation. Due to its wide range of uses and its desirable mathematical properties, it provides a good introduction to image processing with the compute shader. Next, a second filter that has become popular recently is examined—the *bilateral filter*. This is an edge preserving blur filter that can be rather expensive to compute on the CPU, which makes a GPU implementation a very desirable target.

10.1 Image Processing Primer

Before we begin the discussion of implementing image processing algorithms with the compute shader, it would be beneficial to provide a brief introduction to the topic itself. This section will provide some basic entry-level image processing concepts to frame the discussion of the algorithms in this chapter. However, this is by no means an extensive or complete overview of the subject. Many very good image processing books have already been written and are available if the reader would like to investigate the details of the topic further, such as (Gonzalez, 2008).

Image processing is a general term for signal processing operations performed on an image. In our context, the image is likely to be a 2D texture resource that has either been generated in real time or loaded from a file. The input to an image processing algorithm operation is always an image, and the result can be either another image or some information extracted from the input image. In this chapter we will investigate a subset of these algorithms that are used to perform image filtering, but the concepts that we see in this chapter can also be applied to other domains of image processing as well.

10.1.1 Images as Functions

To manipulate the contents of an image, it is helpful to have a more precise definition of what the input image represents. For our purposes, an image is a 2D digital representation of a signal, which is sampled at uniformly spaced sample locations. Each sample in the image is referred to as *a pixel,* and can contain up to four components to represent the sampled signal value at the location of a given pixel. All pixels within an image share the same value format. This description goes along with the familiar 2D grid that is commonly used to show an image, as seen in Figure 10.2, which shows the pixel addressing scheme used by Direct3D 11.

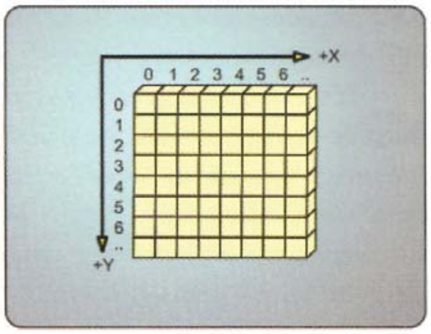

Figure 10.2. A 2D image demonstrating the pixel numbering scheme of Direct3D 11.

It is also noteworthy that there is no restriction to an image being two dimensional. There are many image processing algorithms that can be performed on ID, 2D, 3D, or even higher dimensional signals. However, since the focus of this book is on real-time rendering, we will restrict the discussion to 2D image processing algorithms.

10.1.2 Image Convolution

With this basic definition of an image, we can look a little deeper into the nature of how image processing algorithms function. Many filtering algorithms are implemented as a convolution between the input image itself and another function, referred to as *a filtering kernel.* For discrete domains like our images, the convolution operation is defined as shown in Equation (10.1):

$$w(x,y) * f(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) f(x-s, y-t). \qquad (10.1)$$

This mathematical definition may seem somewhat complex at first, but if we take a simplified view of it, we can more clearly see how this operation is applied to images. The convolution is an operation which takes two functions as input and produces an output function. In our image-processing domain, this means that each pixel in the output image is calculated as the summation of the product of the two input images' pixels at various shifted locations. The shifted locations make the two images conceptually move with respect to one another as each pixel is being processed. To visualize this

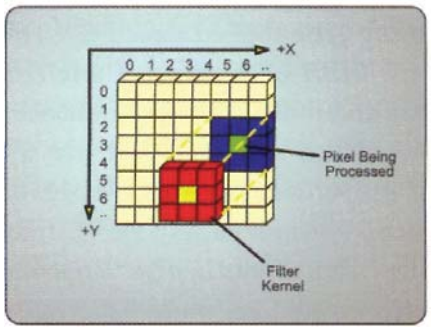

Figure 10.3. The conceptual view of an input image and the filter kernel during processing of a single pixel.

concept, consider Figure 10.3, where the original image is shown, as well as the filter kernel, during processing of a single pixel location.

If we consider this filter shifting operation in the context of what our shader program must do to implement it, we can see that for every pixel in the output image, we must load the image data surrounding the current pixel, apply the filter image to the loaded data, and then sum the individual results and store the value in the output image. This is repeated for every pixel of the output image. With this in mind, we can clearly see how the size of the filter kernel can significantly impact the computational cost of performing a filtering operation, since each pixel requires a number of calculations proportional to the filter size.

This interpretation is relatively simple to visualize, and provides a sufficient view of image processing for the implementation discussions later in the chapter. However, this description is only valid for a subset of image processing filters. Many other filters perform different types of calculations over a pixel's surrounding neighborhood. Fortunately, the same basic concept is still used in the other algorithms as well, in which a small neighborhood of pixels surrounding the current pixel is loaded, some calculations are performed on them, and the results of those calculations are used to determine the value that will be stored in the output image. We will see this pattern in each of the algorithms that we implement in this chapter.

Of course, there is a significant amount of additional mathematical theory and analysis that will be useful to a developer of image filtering algorithms, including frequency domain representations, the various properties of filters in both spatial and frequency domains, and alternative filter implementations and their trade-offs. As mentioned earlier, the reader is invited to further explore these and other details in a complete image processing text.

10.1.3 Separable Filters

There is one final point to discuss before moving on to our sample algorithm implementations. Some filter kernels can be decomposed from a single 2D kernel into two ID kernels that can be applied separately. This is depicted graphically in Figure 10.4.

This filter kernel is referred to as *separable,* a very important property for a filter to exhibit due to its performance implications. In general, for a 2D filter of size $p \times q$ to be applied to an image of size $m \times n$, the number of operations that need to be performed is proportional to $p*q*m*n$. However, if a filter is separable, it can be performed in two steps—one for each of the ID filter kernels that the original 2D filter kernel is decomposed into. This results in $p*m*n$ operations in the first step, and $q*m*n$ operations in the second step. In comparison to the original 2D filter kernel, a separable filter reduces the number of operations from $p*q*m*n$ to $(p+q)*m*n$. This is a significant reduction in calculations, especially when the values of p and q are relatively large. We will utilize this separability property in our sample algorithms in this chapter.

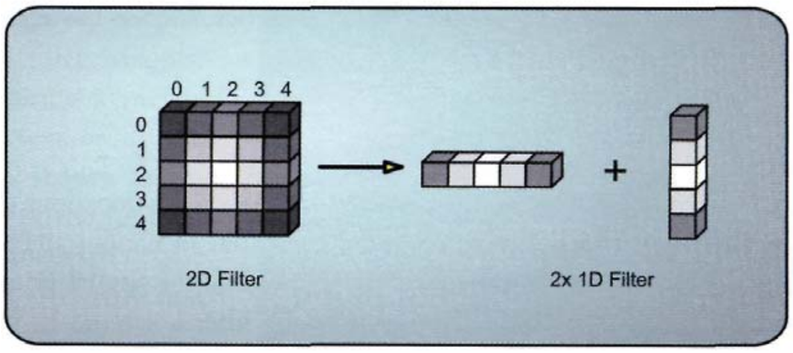

Figure 10.4. A 2D filter kernel decomposed into two ID filters.

10.2 Gaussian Filter

The first image processing algorithm that we will investigate is the *Gaussian filter*. This filter is one of the most widely known, due to the wide range of applications that can use it, and its relatively low computational cost. The filter itself is named after Carl Friedrich Gauss, a brilliant German mathematician (1777–1885) who contributed to many different areas. The term *Gaussian distribution* is often used to refer to a normal distribution in statistics, which is where the name has become associated with an image processing filter.

The Gaussian filter produces a blurred version of the input image, and is hence referred to as a *low-pass filter* in signal processing terms. This means that the lower-frequency content of the image (the portions that don't change rapidly) is mostly preserved, while the higher-frequency content (sharp edges or transitions) is attenuated.

10.2.1 Theory

The filter receives its name from the use of the Gaussian function for creating the filter kernel weights. The Gaussian function is shown in Equation (10.2) for two dimensional inputs, where x and y are the input variables, and σ is a constant factor determined by the developer(which will be clarified shortly). This equation has two main portions—the exponential, and a constant factor applied to it in the beginning of the equation. The exponent in the exponential term calculates a measure of the distance that an input sample location is from the origin by squaring both input parameters. Since these factors will always result in positive values, the final sign of the exponent will always be negative. This results in the exponential term always producing a value between 1 and 0, with 1 being produced when x and y are both zero, and the value falling off to 0 at some characteristic determined by the a parameter. The constant term in the front of the equation simply scales the magnitude of the exponential term to maintain several desirable properties, such as having the integral of the function over the entire input domain be 1:

$$g(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}. \tag{10.2}$$

The only changing variables in this equation are the *x* and *y* locations of the sample. So, the output filter weightings can be regarded as spatial in nature, since the only input into the equation for generating the weights is based on the spatial orientation of the samples. In general, the farther from the center of the filter a sample is, the smaller its resulting weighting on the overall result of the filtered pixel will be. The shape of the filtering kernel can be modified with the constant sigma parameter. Depending on the chosen value of sigma, the amount of blurring can be increased or decreased. The larger the sigma value, the greater the blur that is applied to an image. In Figure 10.5, we can see how an increasingly large sigma value will produce a progressively greater amount of blurring by examining the shape of the filter kernel.

Equation (10.2) can be used to produce the desired set of filter weights for a filter kernel of more or less any size. Since we can't use infinitely sized kernels, we would typically choose an appropriate filter size for the operation being implemented, to produce the appropriate image quality. With a truncated filter kernel size, we can eliminate the constant term from Equation (10.2), since it assumes an infinite input domain. Instead, we will calculate the filter weights at each filter kernel location and then renormalize their total weight by dividing each filter weight by the sum of all of the weights. This maintains the integral-of-1 property with a more computationally friendly filter kernel size. This is required to ensure that the total energy contained within the image signal remains the same before and after the blurring is performed.[1]

The Gaussian filter is used in many areas. Typical uses include performing image blur effects such as blooming, and performing up- and down-sampling operations. In addition, many non-rendering applications also use the Gaussian-filter, such as a blur operation being performed prior to a dilation operation in a character recognition system.

10.2.2 Implementation Design

The basic implementation of the Gaussian filter is relatively simple. The desired filter kernel is filled in with values generated by the Gaussian function with a sigma value chosen for the desired level of blurring. This is typically done at design time, so that the filtering kernel is not recalculated for every pixel being processed; but if a dynamic filter size is needed,

[1] It is possible to simultaneously achieve other effects by changing this total filter weight as well. For example, if the kernel weights sum to a value less than or greater than 1.0, the resulting image will be either darkened or brightened, respectively.

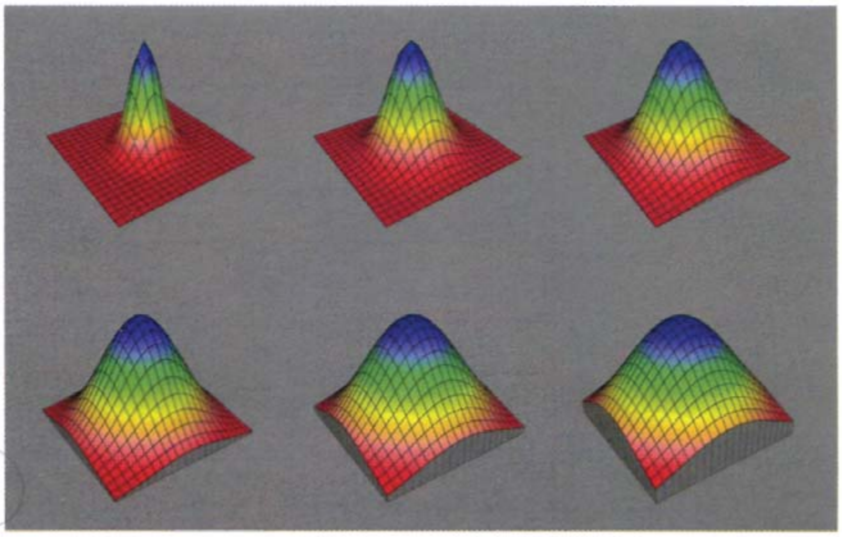

Figure 10.5. Visualizations of various filter kernels with increasing values of sigma, which will produce increasingly blurred output images.

it is not mandatory to pre-calculate the filter weights. Once the filtering kernel is available, we can easily use the compute shader to invoke a single thread for each pixel to be processed on the input image. We will first consider a nai've, brute-force approach, before moving on to a more elegant solution.

One other consideration must be made for when implementing filter weights. When it is near an edge of the input image, the filter will sample image locations that are actually outside of the image. This must be considered and handled appropriately for a given situation, and is typically compensated for in one of several ways. The exterior samples can be clamped to the edge of the image, which will essentially make the edge pixels a little bit sharper than the interior pixels after the blurring process. Another way to handle this would be to eliminate samples that fall outside of the image. To eliminate a sample, the algorithm must not only skip the addition of the weight filter sample, but must also reduce the overall sample amount. Since we choose the filter weights to add up to 1, we are performing an implicitly neutral operation on the image in the regular, interior pixel cases. If we eliminate some samples, the overall filter weight will add up to less than 1, and thus the resulting value must be renormalized once again by dividing the output value by the new total filter weights. This adds some complexity to the calculation process, and must be implemented with care to ensure that the performance of the algorithm is not significantly reduced. The implementations demonstrated in this chapter simply neglect this effect, since the focus of the chapter is to use the computer shader in an efficient manner.

Brute Force Approach

To invoke the compute shader on each thread, we must select a thread group size that is within the upper limits of the thread count (which is less than 1024 total threads), but that can be invoked in a dispatch size that can cover the entire input image. For example, if we choose the thread groups to be of size [32,32,1], we are within the thread count limit, and our dispatch call can choose the appropriate number of thread groups to request based on

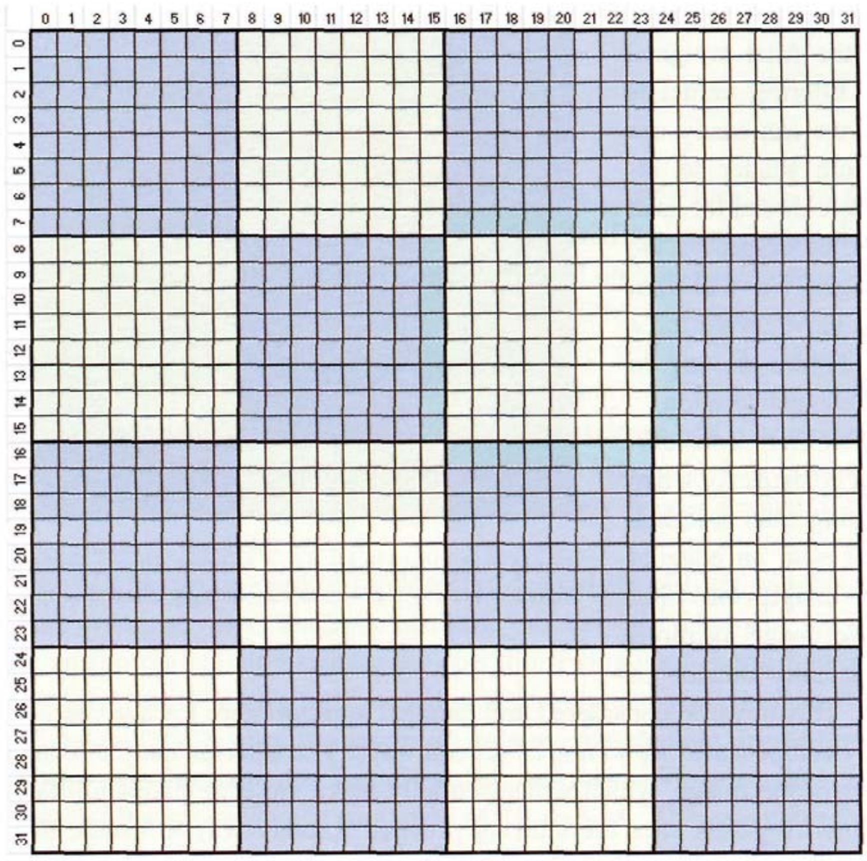

Figure 10.6. Processing an image with a square tile sized thread group, with an example size of 8×8 tiles.

the size of the image to be filtered. If we are processing a 640×480 image, the dispatch size would be [20,15,1]. This tile-based thread group size is demonstrated in Figure 10.6. In general, choosing the thread group size to be a common factor of the available render target sizes is a good initial choice, which will allow the filter shader program to be reused for multiple image sizes.

For the purposes of our first sample, we will assume a 7×7 filter kernel size. The filter is provided as static constants in the shader file, which can be accessed as a two dimensional array of values. Within the shader program, we will use the dispatch thread ID to allow each thread to choose which pixel it will be processing. This is trivial to perform, since the dispatch thread ID will span a range that exactly matches the size of image. Indeed, this is done by design to ensure that we can use this thread addressing scheme! Once each thread is aware of which pixel it should be processing, we can read the input image data for each of the pixels that are needed in our filter kernel. Each pixel value is multiplied by the corresponding filter kernel weight, and all of these products are summed to produce the resultant pixel value for the output image. The output value is stored in the output texture at the pixel location indicated by the dispatch thread ID in much the same way that was used to read the appropriate data from the input image. This initial implementation is shown in Listing 10.1 and visualized for a single pixel in Figure 10.7.

```
// Declare the input and output resources
Texture2D<float4>     InputHap  :  register( t0 );
RWTexture2D<float4> OutputMap  :  register( u0 );

// Group size
#define size_x 32
#define size_y 32

// Declare the filter kernel coefficients
static const float filter[7][7]  = {
    0.000904706, 0.00315-7733, 0.00668492, 0.008583607, 0.00668492,
0.003157733, 0.000904706/
    0.003157733, 0.01102157, 0.023332663, 0.029959733, 0.023332663,
0.01102157, 0.003157733,
    0.00668492, 0.023332663, 0.049395249, 0.063424755, 0.049395249,
0.023332663, 0.00668492,
    0.008583607, 0.029959733, 0.063424755, 0.081438997, 0.063424755,
0.029959733, 0.008583607,
    0.00668492, 0.023332663, 0.049395249, 0.063424755, 0.049395249,
0.023332663, 0.00668492,
    0.003157733, 0.01102157, 0.023332663, 0.029959733, 0.023332663,
0.01102157, 0.003157733,
    0.000904706, 0.003157733, 0.00668492, 0.008583607, 0.00668492,
0.003157733, 0.000904706
};

// Declare one thread for each texel of the current block size.
 [numthreads(size_x, size_y, 1)]

void CSMAIN( uint3 DispatchThreadID : SV_DispatchThreadID )
{
    // Offset the texture location to the first sample location
    int3 texturelocation = DispatchThreadID - int3( 3, 3, 0 );

    // Initialize the output value to zero, then loop through the
    // filter samples, apply them to the image samples, and sum
    // the results.
    float4 Color = float4( 0.0, 0.0, 0.0, 0.0 );

    for ( int x = 0; x < 7; x++ )
        for ( int y = 0; y < 7; y++ )
            Color += InputMap.Load( texturelocation + int3(x,y,0) ) * filter[x][y];

    // Write the output to the output resource
    OutputMap[DispatchThreadID.xy] = Color;
}
```

Listing 10.1. The brute force approach to implementing the Gaussian filter.

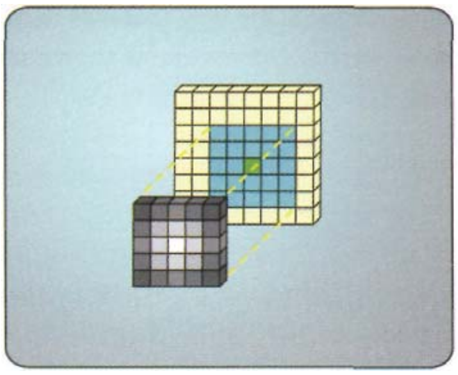
Figure 10.7. A visualization of applying the filter kernel to a single pixel.

This implementation is perfectly adequate for producing a blurred output image in the mathematically correct way. With the highly parallel nature of modern GPUs, this operation is likely to be performed quite quickly. However, there is almost always a need to perform more operations in every rendered frame, to allow either for better image quality or for higher throughput of an operation if it used in a GPGPU application. Therefore, we will attempt to improve the performance of our algorithm, while still producing the same output image.

Separable Gaussian Filter

In our brief image processing primer, we saw that some filter kernels are separable. The Gaussian filter is in fact a separable filter, which we can exploit to gain a significant performance advantage. When a 2D filter is decomposed into two ID filters, this requires us to execute two processing passes on our image —one for each of the two ID filters. In the case of the Gaussian filter, both of the ID filters are simply the cross section of the 2D filter through its center. In the 7×7 filter from our example above, the decomposition into two ID filters will produce a 1×7 filter and a 7×1 filter. This can be thought of as producing filters in both the *x*- and *y*-directions, due to the shape of the resulting filters. This is shown graphically in Figure 10.8.

The first processing pass will read the input image, apply the first filter, and produce an intermediate image result which must be stored in a new texture resource. The second processing pass will read the intermediate result, apply the second filter, and produce the final filtered image. This means that we will require an additional texture resource, which means that additional memory is used by this modified algorithm. However, the potential performance improvement is typically a worthy tradeoff for the extra memory consumption.

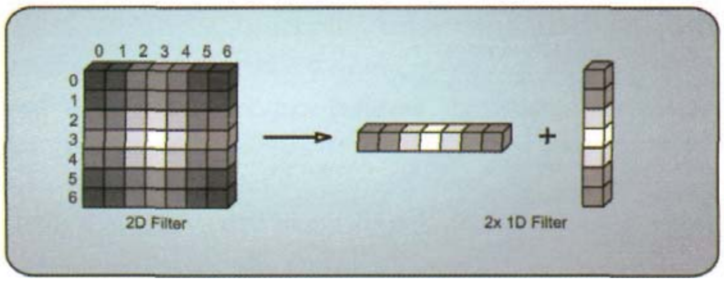

Figure 10.8. Decomposing our 7×7 filter into two ID filters.

Since we will be modifying the filter kernel that we will be executing, it makes sense to reevaluate the threading setup that we have used in the previous implementation. Instead of using a square thread group size, it makes more sense to flatten out our thread groups to match the shape of the processing kernel being used in each pass. Thus, for the first pass we will use a thread group size of [640,1,1], and the second pass will use a thread group size of [1,480,1]. This will allow us to use the group shared memory to even further reduce the required device memory bandwidth.

In our previous implementation, each thread read all of the input texture values it needed to calculate its own output value. For a 7×7 filter, this amounts to 49 individual values to read from the input image per thread. In a separable implementation, we can expect each thread to only need to read the data for either a 1×7 or a 7×1 filter, for a total of 7 + 7 = 14. This is already a significant reduction of explicit memory reads, although the effective reduction may be somewhat smaller due to the GPU's texture data caching helping the naive implementation. In any case, we can further reduce the number of reads from the input texture resource by using the GSM. If each thread reads its own input pixel data and then stores it in the group shared memory for all the threads in the thread group to use, the effective number of device memory reads per thread is reduced from 7 + 7 to 1 + 1! Of course we are adding some overhead to this implementation by writing to and reading from the group shared memory, but in general, this should be a performance improvement over performing many more read operations from device memory. After all of the threads have loaded their data into the GSM, we perform a group memory barrier with thread sync to ensure that all of the needed data has been written to the GSM before moving on with the filtering operations. The updated compute shader program for the horizontal filter is shown in Listing 10.2. The vertical version is omitted, since it is identical to the shown version except that it samples in a vertical pattern and declares a different amount of group shared memory.

```
// Declare the input and output resources
Texture2D<float4>    InputMap : register( t0 );
RWTexture2D<float4>  OutputMap : register( u8 );

// Image sizes
#define size_x 648
#define size_y 480

// Declare the filter kernel coefficients
static const float filter[7] = {
    0.030078323, 0.104983664, 0.222250419, 0.285375187, 0.222250419,
0.104983664, 0.030078323
};

// Declare the group shared memory to hold the loaded data
groupshared float4 horizontalpoints[size_x];
```

```
// For the horizontal pass, use only a single row of threads
[numthreads(size_x, 1, 1)]

void CSMAINX( uint3 DispatchThreadID : SV_DispatchThreadID )
{
    // Load the current data from input texture
    float4 data = InputMap.Load( DispatchThreadID );

    // Stor the data into the GSM for the current thread
    horizontalpoints[DispatchThreadID.x] = data;

    // Synchronize all threads
    GroupMemoryBarrierWithGroupSync();

    // Offset the texture location to the first sample location
    int3 texturelocation = DispatchThreadID - int3( 3, 0, 0 );

    // Initialize the output value to zero, then loop through the
    // filter samples, apply them to the image samples, and sum
    // the results.
    float4 Color = float4( 0.0, 0.0, 0.0, 0.0 );

    for ( int x = 0; x < 7; x++ )
        Color += horizontalpoints[texturelocation.x + x] * filter[x];

    // Write the output to the output resource
    OutputMap[DispatchThreadID.xy] = Color;
```

Listing 10.2. The horizontal pass compute shader listing for our separable implementation.

One important point to note in this listing is that the thread group size has been modified to reflect our desired ID shape. Since the limit on thread group size is currently 1024, it is possible that one horizontal line will not fit in a single thread group if the input image is sufficiently large. However, this can be overcome by simply modifying the dispatch size to perform each pass with two thread groups for each row or column to be processed. Also, note that the group shared memory is declared such that/it can hold the appropriate number of pixels for the current thread group size. Finally, notice that we use the GroupMemoryBarrierWithGroupSync() function to ensure that all threads have executed to this point and that the group memory accesses have been completed before continuing with the filtering process.

Data Sharing with GSM

In the separable Gaussian implementation, we used the group shared memory to significantly reduce the number of device memory accesses needed to perform the desired filtering operation. By using the GSM as a memory cache, we can trade device memory

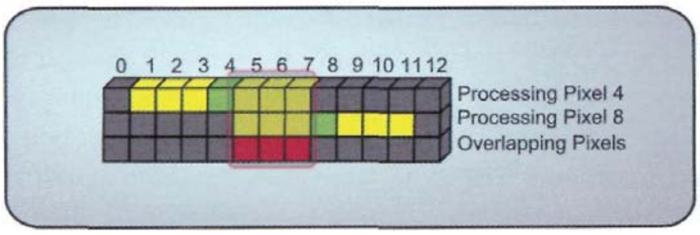

Figure 10.9. A visualization of overlapping calculations in neighboring pixels.

accesses for increasing the number of GSM accesses, plus a memory barrier. However, the group shared memory can be used to share more than cached resource data. We can further modify our current implementation to allow threads to share the results of some calculations to reduce the overall computational burden on the GPU. This may become beneficial after optimizing an algorithm to reduce memory accesses—if the computational portion of the algorithm becomes proportionally large enough, it can be helpful to attempt to minimize even further the number of mathematic operations.

In the case of the Gaussian filter, we can take advantage of the fact that the two individual 1D processing passes use filter weights that are symmetric. This, coupled with our row-and-column based threading scheme, allows us to share intermediate calculations between two different pixels. To aid in explaining this possibility, Figure 10.9 shows the calculations needed for two pixels that are near each another. The key to this concept is that any two pixels equidistant from a pixel between them will both perform the same calculation on that center pixel.

To avoid having duplicate calculations, we can have each thread precalculate the weighted versions of its own pixel value and store it in the GSM before the memory barrier is performed. The precalculated values can then be read by whichever thread needs to use them, which should effectively reduce the number of multiplications needed by a factor of roughly 2. This requires the use of additional group shared memory, as well as additional read and writes to this additional memory. Figure 10.10 shows the layout that will be used to store the additional precalculated values; the modified compute shader program is shown in Listing 10.3.

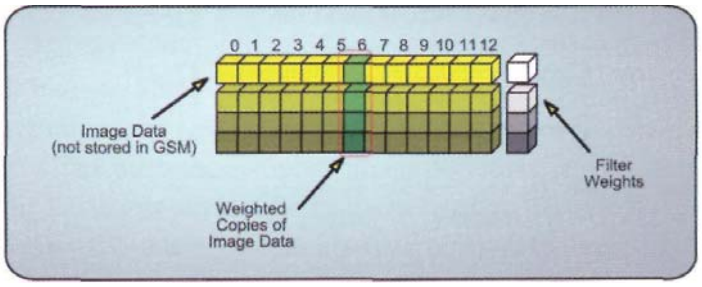

Figure 10.10. The group shared memory layout for caching shared values.

```
// Declare the input and output resources
Texture2D<float4>   InputMap : register( t0 );
RWTexture2D<float4> OutputMap : register( u0 );

// Image sizes
#define size_x 640
#define size_y 480

// Declare the filter kernel coefficients
static const float filter[7] = {
    0.030078323, 0.104983664, 0.222250419, 0.285375187, 0.222250419,
    0.104983664, 0.030078323
};

// Declare the group shared memory to hold the loaded and calculated data
groupshared float4 horizontalpoints[size_x][3];

// For the horizontal pass, use only a single row of threads
[numthreads(size_x, 1, 1)]

void CSMAINX( uint3 DispatchThreadID : SV_DispatchThreadID )
{
    // Load the current data from input texture
    float4 data = InputMap.Load( DispatchThreadID );

    // Stor the data into the GSM for the current thread
    norizontalpoints[DispatchThreadID.x][0] = data * filter[0];
    horizontalpoints[DispatchThreadID.x][1] = data * filter[1];
    horizontalpoints[DispatchThreadID.x][2] = data * filter[2];

    // Synchronize all threads
    GroupMemoryBarrierWithGroupSync();

    // Offset the texture location to the first sample location
    int3 texturelocation = DispatchThreadID - int3( 3, 0, 0 );

    // Initialize the output value to zero, then loop through the
    // filter samples, apply them to the image samples, and sum
    // the results.
    float4 Color = float4( 0.0, 9.0, 0.0, 0.0 );

    Color += horizontalpoints[texturelocation.x +       0][0];
    Color += horizontalpoints[texturelocation.x + 1][1];
    Color += horizontalpoints[texturelocation.x + 2][2];
    Color += data * filter[3];
    Color += horizontalpoints[texturelocation.x + 4][2];
    Color += horizontalpoints[texturelocation.x + 5][1];
    Color += horizontalpoints[texturelocation.x + 6][0];
    // Write the output to the output resource
    OutputMap[DispatchThreadID.xy] = Color;
}
```

Listing 10.3. The cached Gaussian implementation.

While this change to the algorithm does indeed reduce the number of arithmetic operations performed by each thread, it also increases the number of memory access operations as well. The group shared memory is supposed to be a very fast memory area, but there is still some performance penalty for accessing it. In addition, the arithmetic operations we are attempting to minimize in this case are simple multiplications, which modern GPUs can perform quite quickly. This may mean that any potential performance gains will be relatively small. This type of calculation would be more beneficial if we were replacing longer computational strings or more expensive instructions. In the end, the final results will likely vary by the GPU being used and its individual performance characteristics for arithmetic and memory access operations.

10.2.3 Conclusion

The Gaussian filter is a very widely used filter, which provides an adjustable amount of blurring and can be used with a variety of different filter sizes. This allows the filter's performance to be adjusted to the given situation. By using the separable nature of the Gaussian filter, we greatly reduced the number of memory access and arithmetic operations, and then further reduced the number of memory accesses by using the group shared memory as a customized memory cache. We even further reduced the number of arithmetic operations by precalculating some filter elements that are shared by more than one pixel. This type of optimization trades arithmetic operations for memory access operations, and can be beneficial in some cases, when sufficiently heavy arithmetic operations are being replaced. However, it may not always produce a net performance improvement. This means that the algorithm should be carefully designed to use the GSM only when it is beneficial to do so.

10.3 Bilateral Filter

The second filter algorithm we will investigate is the *bilateral filter*. This filter was originally proposed in (Tomasi, 1998). It is similar in nature to the Gaussian filter and provides another form of a blurring filter. However, the bilateral filter blurs an image while still preserving the sharp edges and object boundaries in the image content. This is in contrast to the Gaussian filter, which blurs the complete image regardless of the content. If a Gaussian filter is used to blur the image several times in succession, the image can become washed out, and it can become difficult to discern between objects. Figure 10.11 demonstrates the difference between the two filter outputs after performing several passes of the filter.

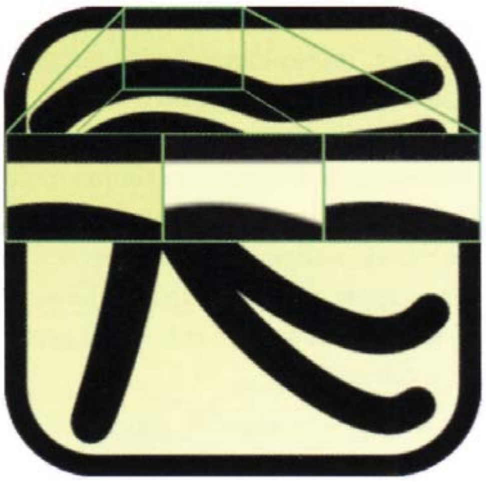
Figure 10.11. A comparison of the original image, Gaussian filtered (center) and the bilateral filtered (right).

Due to its edge-preserving properties, the bilateral filter can be used to perform a different class of functions than the Gaussian filter. For example, algorithms that use stochastic methods to sample the properties of a complex object often result in a noisy output, due to the randomized nature of the sampling technique. The bilateral filter can be used to reduce or minimize the noise in the signal, while still retaining the important boundaries that define various objects in the image. A common example of this is using a bilateral filter to smooth the results of screen space ambient occlusion calculations.

10.3.1 Theory

The bilateral filter shares some properties with the Gaussian filter. With the bilateral filter, weights are based on the spatial distances from the center of the filter, as with the Gaussian filter, but an additional factor is added to the weight calculation. The filter weights are also scaled by the relative difference in color values between the current pixel being processed and the sample pixel. If the two colors are somewhat similar, the scaling of the weight leaves the weight value mostly unchanged. However, if there is a large difference between the two color values, the weight is scaled to a smaller value, which has less influence on the overall result of the pixel. Thus, the bilateral filter modifies its weights based not only on the spatial arrangement of samples, but also on the intensity values contained within the image. The equation for calculating the weights of each individual sample is shown in Equation (10.3):

$$BF[I_p] = \frac{1}{W_p}\sum_{q\in S} G_S(\| p - q \|)G_r(I_p - I_q)I_q. \tag{10.3}$$

As seen in Equation (10.2) (Paris), the bilateral filter weights are a product of a Gaussian function based on the spatial distance (indicated by G_s), and a Gaussian function based on the color delta value (indicated by G_r). This dependence on the intensity values is the key to performing a filtering operation that preserves edges in the image. However, since every pixel in the image has a potentially different value, the weights must be calculated dynamically for every sample that is used to produce an output pixel value. This is a relatively large difference from the Gaussian filter, where we could easily precalculate the desired filter weight based on the desired sigma value. In addition, since the weights change from pixel to pixel, the bilateral filter is a nonlinear filter, which also means that it is generally not separable. This can have profound performance implications when sufficiently large filter sizes are used.

Once all of the weights have been calculated, the resulting output pixel value is determined in the same way as in as the Gaussian filter. The appropriate neighboring pixels are sampled, and then scaled by the current pixel's corresponding weighting value. All of these weighted samples are then summed to produce an output pixel value. Another very important point to consider when calculating the bilateral filter weights is that the content of the image may produce filter weights that add up to significantly less than 1. This can produce an abnormally darkened area in an area that is otherwise relatively bright, which would appear out of place. This is in contrast to Gaussian filter weights, which are chosen to always add up to a total of 1. To compensate for this fluctuation in total weight, the total contributions from all of the pixels must be renormalized with one another. This is done by trivially dividing the resulting pixel value by the sum of the weights themselves. This is depicted graphically in Figure 10.12.

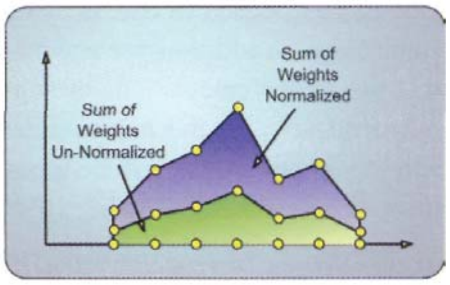

Figure 10.12. Renormalizing of the bilateral filter weights.

Another complication to using this filter is the fact that there are three independent channels in a normal color texture. This means that the intensity parameter that is used for the Gaussian filter weighting must either be performed for all three channels independently, or some mechanism must be used to convert from a three-dimensional color to a one-dimensional equivalent value prior to calculating the weight values. Both possibilities will increase the number of calculations needed to determine the weight values, and the latter option will reduce the image quality of the filter somewhat, by approximating the color space with a single scalar value. In principle, since the GPU is capable of performing vector style calculations, it should be possible to calculate each of the color channels independently with the same number of instructions.[2]

[2] In the past, all GPU architectures were vector-register based. However, this is not the case any longer, with recent NVIDIA architectures using a scalar execution pipeline (while recent AMD architectures remain vector based). Thus, the comments presented here about vector operations may or may not apply to a particular GPU.

10.3.2 Implementation Design

The basic implementation setup for the bilateral filter is very similar to that of the Gaussian filter. It will execute in the compute shader, with the input image bound with a shader resource view, and the resulting image will be bound with an unordered access view so the shader program can write the appropriate data to it. We will consider how a simple brute force implementation looks, and then later develop a more efficient approach.

Brute Force Approach

The straightforward technique is quite similar to the Gaussian method, with the exception that the color-based weight value must be calculated and applied to each sample used in the summation. In addition, we assume that the input color image will use all four channels and will use float4 calculations throughout the shader to allow each channel to be processed independently. For this naive implementation, each pixel will process a single output pixel, and the thread groups will be arranged in the 32×32 sized groups once again. The implementation is shown in Listing 10.4.

```
// Declare the input and output resources
Texture2D<float4>   InputMap : register( t0 );
RWTexture2D<float4> OutputMap : register( u0 );

// Group size
ttdefine size_x 32
#define size_y 32

// Declare the filter kernel coefficients
static const float  filter[7][7]  = {
    0.000904706, 0.003157733, 0.00668492, 0.008583607, 0.00668492,
0.003157733, 0.000904706,
    0.003157733, 0.01102157, 0.023332663, 0.029959733, 0.023332663,
0.01102157, 0.003157733,
    0.00668492, 0.023332663, 0.049395249, 0.063424755, 0.049395249,
0.023332663, 0.00668492,
    0.008583607, 0.029959733, 0.063424755, 0.081438997, 0.063424755,
0.029959733, 0.008583607,
    0.00668492, 0.023332663, 0.049395249, 0.063424755, 0.049395249,
0.023332663, 0.00668492,
    0.003157733, 0.01102157, 0.023332663, 0.029959733, 0.023332663,
0.01102157, 0.003157733,
    0.000904706, 0.003157733, 0.00668492, 0.008583607, 0.00668492,
0.003157733, 0.000904706
};

// Declare one thread for each texel of the current block size.
 [numthreads(size_x, size_y, 1)]
```

```
void CSMAIN( uint3 GroupID : SV_GroupID, uint3 DispatchThreadID :
SV_DispatchThreadID, uint3 GroupThreadID : SV_GroupThreadID, uint GroupIndex
: SV_GroupIndex )
{
    // Offset the texture location to the first sample location
    int3 texturelocation = DispatchThreadID - int3( 3, 3, 0 );

    // Each thread will load its own depth/occlusion values
    float4 CenterColor = InputMap.Load( DispatchThreadID );

    // Range sigma value
    const float rsigma = 0.051f;

    float4 Color = 0.0f;
    float4 Weight = 0.0f;

    for ( int x = 0; x < 7; x++ )
    {
        for ( int y = 0; y < 7; y++ )
        {
            // Get the current sample
            float4 SampleColor = InputMap.Load( texturelocation + int3( x, y, 0 ) );

            // Find the delta, and use that to calculate the range weighting
            float4 Delta = CenterColor - SampleColor;
            float4 Range = exp( ( -1.0f * Delta * Delta )
                               / ( 2.0f * rsigma * rsigma ) );

            // Sum both the color result and the total weighting used
            Color += SampleColor * Range * filter[x][y];
            Weight += Range * filter[x][y];
        }
    }

    // Store the renormalized result to the output resource
    OutputMap[DispatchThreadID.xy] = Color / Weight;
}
```

Listing 10.4. The brute force bilateral filter implementation.

We can see from Listing 10.4 that the color-weighting factor is based on the delta value between the center pixel and the sample pixel, which is then used as the input parameter to a Gaussian function. The spatial weights are precalculated in the same way we saw in the Gaussian filter implementation, but they could just as easily be converted to be dynamically calculated, or even provided in a constant buffer by the application. All of the samples are looped through to build the overall weighted combination, and the total combination of the weights that are applied to the samples is summed in the Weight variable. This is then used to renormalize the summed output combination of the samples, producing a final result that is written to the output resource through the unordered access view.

Like to the Gaussian brute force approach, this implementation performs a number of operations roughly proportional to the number of samples used in the filter kernel. For our 7×7 filter example, this means that the primary portion of our algorithm is executed 49 times, with each loop performing a device memory resource read, followed by some computations on the data; and finally, the result is written to the device memory resource. Clearly, this is not an optimal solution, since memory accesses can introduce significant time delays while the algorithm waits for the requested data to be fetched by the GPU memory system.

Separable Bilateral Filter

In our Gaussian filter implementation, we used the filter's separable nature to reduce the amount of work required to process a pixel. This allows us to perform the algorithm in two steps, which not only reduces the number of operations, but also lets us use the group shared memory to reduce the number of device memory accesses even further. Overall, this provides a significant performance improvement over the naive implementation. Ideally, we would like to use a similar technique to reduce the number of calculations and memory accesses needed for the bilateral filter as well.

However, as we mentioned earlier, the bilateral filter is nonlinear and is generally not separable. Strictly speaking, this means that we would not be able to perform the same style of optimizations with the bilateral filter. With this in mind, in many cases, it is still possible to use a separable implementation, even though it is not mathematically correct. The resulting output image will not be identical to the basic implementation, but since this filter performs a blurring operation, it is less noticeable if the results are not precisely the same as those of the true algorithm. The performance benefits of using a separable filter generally outweigh the imperfect results, and we will make this tradeoff in the next implementation.

After deciding to use a separable version of the filter, we will set up the algorithm in much the same way as the separable Gaussian filter, with the exception of the new per-sample calculations that are required for the bilateral filter. We can also use the group shared memory to cache the required color values, just as we have seen in the Gaussian implementation. The remainder of the filter remains the same, "with the exception that we must execute two passes to perform the algorithm now, instead of a single pass, as before. The updated filter implementation is shown in Listing 10.5.

```
// Declare the input and output resources
Texture2D<float4>   InputMap  : register( t0 );
RWTexture2D<float4> OutputMap : register( u0 );

// Image sizes
#define size_x 640
#define size_y 480
```

```
// Declare the filter kernel coefficients
static const float filter[7] = {
    0.030078323, 0.104983664, 0.222250419, 0.285375187, 0.222250419,
    0.104983664, 0.030078323
};

// Declare the group shared memory to hold the loaded data
groupshared float4 horizontalpoints[size_x];

// For the horizontal pass, use only a single row of threads
[numthreads(size_x, 1, 1)]

void CSMAINX( uint3 DispatchThreadID : SV_DispatchThreadID )
{
    // Load the current data from input texture
    float4 CenterColor = InputMap.Load( DispatchThreadID );

    // Stor the data'vinto the GSM for the current thread
    horizontalpoints[DispatchThreadID.x] = CenterColorj

    // Synchronize all threads
    GroupMemoryBarrierWithGroupSync();

    // Offset the texture location to the first sample location
    int3 texturelocation = DispatchThreadID - int3( 3, 0, 0 );

    // Range sigma value
    const float rsigma = 0.051f;

    float4 Color = 0.0f;
    float4 Weight = 0.0f;

    for ( int x = 0; x < 7; x++ )
    {
        // Get the current sample
        float4 SampleColor = horizontalpoints[texturelocation.x + x];

        // Find the delta, and use that to calculate the range weighting
        float4 Delta = CenterColor - SampleColor;
        float4 Range = exp( ( -1.0f * Delta * Delta )
                          / ( 2.0f * rsigma * rsigma ) );

        // Sum both the color result and the total weighting used
        Color += SampleColor * Range * filter[x];
        Weight += Range * filter[x];
    }

    // Store the renormalized result to the output resource
    OutputMap[DispatchThreadID.xy] = Color / Weight;
```

Listing 10.5. A separable implementation of the bilateral filter.

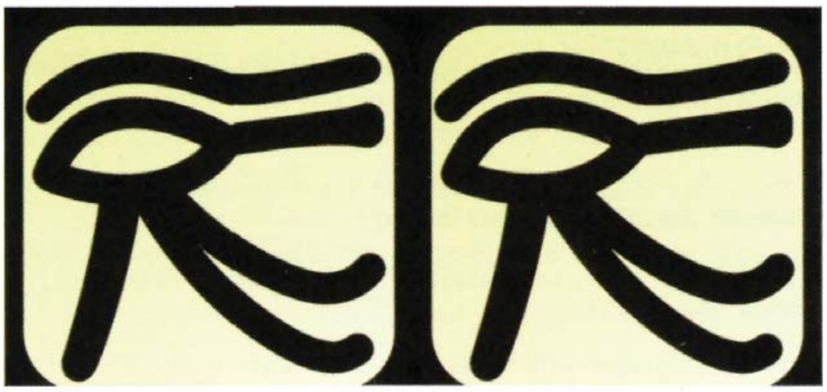

Figure 10.13. A comparison of the standard implementation and the separable implementation of the bilateral filter.

The results from this version of the algorithm are significantly faster than the brute force method. In this case, the resulting image quality is similar to what it would have been and does not produce objectionable artifacts in the output image. Figure 10.13 shows a sample image with both the result of the original filter and the approximated result from the separable implementation in Figure 10.13.

10.3.3 Conclusion

The bilateral filter is a very important tool to have in any image processing library. Its ability to perform a blurring operation that mostly preserves strong edges makes it quite versatile. It has applications in post-processing, with a particularly strong case for being used to smooth the results of stochastic, or randomized, sampling techniques. While the brute force implementation produces a slightly higher image quality result, by use the separable filter techniques, we can gain a big performance advantage and perhaps even use larger filter sizes, because of the computational savings.