EUROGRAPHICS 2005 / M. Alexa and J. Marks
(Guest Editors)

Volume 24 (2005), Number 3

Fast Summed-Area Table Generation and its Applications

Justin Hensley[1], Thorsten Scheuermann[2], Greg Coombe[1], Montek Singh[1] and Anselmo Lastra[1]

[1]University of North Carolina, Chapel Hill, NC, USA — {hensley, coombe, montek, lastra}@cs.unc.edu
[2]ATI Research, Marlborough, MA, USA — thorsten@ati.com

Abstract

We introduce a technique to rapidly generate summed-area tables using graphics hardware. Summed area tables, originally introduced by Crow, provide a way to filter arbitrarily large rectangular regions of an image in a constant amount of time. Our algorithm for generating summed-area tables, similar to a technique used in scientific computing called recursive doubling, allows the generation of a summed-area table in O(log n) time. We also describe a technique to mitigate the precision requirements of summed-area tables. The ability to calculate and use summed-area tables at interactive rates enables numerous interesting rendering effects. We present several possible applications. First, the use of summed-area tables allows real-time rendering of interactive, glossy environmental reflections. Second, we present glossy planar reflections with varying blurriness dependent on a reflected object's distance to the reflector. Third, we show a technique that uses a summed-area table to render glossy transparent objects. The final application demonstrates an interactive depth-of-field effect using summed-area tables.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism: Color, shading, shadowing, and texture I.4.3 [Image Processing and Computer Vision]: Enhancement: Filtering

1. Introduction

There are many applications in computer graphics where spatially varying filters are useful. One example is the rendering of glossy reflections. Unlike perfectly reflective materials, which only require a single radiance sample in the direction of the reflection vector, glossy materials require integration over a solid angle. Blurring by filtering the reflected image with a support dependent on the surface's BRDF can approximate this effect. This is currently done by pre-filtering off line, which limits the technique to static environments.

Crow [Cro84] introduced summed-area tables to enable more general texture filtering than was possible with mip maps. Once generated, a summed-area table provides a means to evaluate a spatially varying box filter in a constant number of texture reads. Heckbert [Hec86] extended Crow's work to handle complex filter functions.

In this paper we present a method to rapidly generate summed-area tables that is efficient enough to allow multiple tables to be generated every frame while maintaining

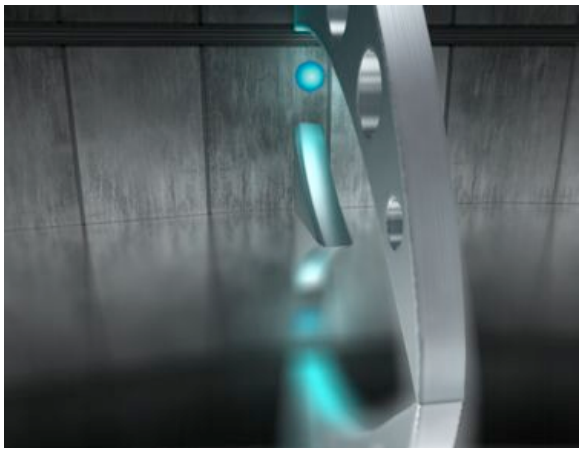

Figure 1: *An image illustrating the use of a summed-area table to render glossy planar reflections where the blurriness of an object varies depending on its distance from the reflector.*

interactive frame rates. We demonstrate the applicability of spatially varying filters for real-time, interactive computer graphics through four different applications.

The paper is organized as follows: Section 2 provides background information on summed-area tables. Next, in Section 3 we present our technique for generating summed-area tables. In Section 4 we describe a method for alleviating the precision requirements of summed-area tables. Section 5 presents several example applications using summed-area tables for real-time graphics followed by a discussion of summed-area table performance. Then future work is presented in Section 7, and Section 8 concludes the paper.

2. Background

Originally introduced by Crow [Cro84] as an alternative to mip maps, a summed-area table is an array in which each entry holds the sum of the pixel values between the sample location and the bottom left corner of the corresponding input image.

Summed-area tables enable the rapid calculation of the sum of the pixel values in an arbitrarily sized, axis-aligned rectangle at a fixed computational cost. Figure 2 illustrates how a summed-area table is used to compute the sum of the values of pixels spanning a rectangular region. To find the integral of the values in the dark rectangle, we begin with the pre-computed integral from (0,0) to (x_R, y_T). We subtract the integrals of the rectangles (0, 0) to (x_R, y_B) and (0, 0) to (x_L, y_T). The integral of the hatched box is then added to compensate for having been subtracted twice.

The average value of a group of pixels can be calculated by dividing the sum by the area. Crow's technique amounts to convolution of an input image with a box filter. The power lies in the fact that the filter support can be varied at a per pixel level without increasing the cost of the computation. Unfortunately, since the value of the sums (and thus the dynamic range) can get quite large, the table entries require extended precision. The number of bits of precision needed per component is

$$P_s = log_2(w) + log_2(h) + P_i$$

where w and h are the width and height of the input image. P_s is the precision required to hold values in the summed-area table, and P_i is the number of bits of precision of the input. Thus, a 256x256 texture with 8-bit components would require a summed-area table with 24 bits of storage per component.

Another limitation of Crow's summed-area table technique is that it is only capable of implementing a simple box filter. This is because only the sum of the input pixels is stored; therefore it is not possible to directly apply a generic filter by weighting the inputs.

In [Hec86], Heckbert extended the theory behind

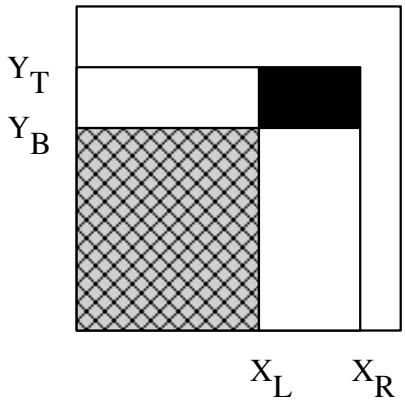

Figure 2: *(after [Crow84]) An entry in the summed-area table holds the sum of the values from the lower left corner of the image to the current location. To compute the sum of the dark rectangular region, evaluate $T[X_R,Y_T] - T[X_R,Y_B] - T[X_L,Y_T] + T[X_L,Y_B]$ where T is the value of the entry at (x, y)*

summed-area tables to handle more complex filter functions. Heckbert made two key observations. The first is that a summed-area table can be viewed as the integral of the input image, and the second that the sample function introduced by Crow was the same as the derivative of the box filter function. By taking advantage of those observations and the following convolution identity

$$f \otimes g = f'^n \otimes \int^n g$$

it is possible to extend summed-area tables to handle higher order filter functions, such as the Bartlett filter, or even a Catmull-Rom spline filter. The process is essentially one of repeated box filtering. Higher order filters approach a Gaussian, and exhibit fewer artifacts.

For instance, Bartlett filtering requires taking the second-order box filter, and weighting it with the following coefficients:

$$f = \begin{matrix} 1 & -2 & -1 \\ -2 & 4 & -2 \\ 1 & -2 & -1 \end{matrix}$$

Unfortunately, a direct implementation of the Bartlett filtering example requires 44 bits of precision per component, assuming 8-bits per component and a 256x256 input image.

In general, the precision requirements of Heckbert's method can be determined as follows:

$$P_s = n * (log_2(w) + log_2(h)) + P_i$$

where w and h are the width and height of the input texture, n is the degree of the filter function, P_i is the input image's precision, and P_s is the required precision of the n^{th}-degree summed-area table.

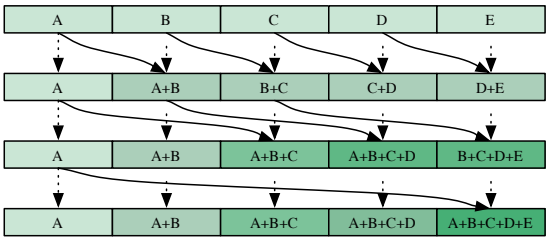

Figure 3: *The recursive doubling algorithm in 1D. On the first pass, the value one element to the left is added to the current value. On the second pass, the value two elements to the left is added the current value. In general, the stride is doubled for each pass. The output is an array whose elements are the sum of all of the elements to the left, computed in O(log n) time.*

A technique introduced by [AG02, YP03] combines multiple samples from different levels of a mip map to approximate filtering. This technique suffers from several problems. First, a small step in the neighborhood around a pixel does not necessarily introduce new data to the filter; it only changes the weights of the input values. Second, when the inputs do change, a large amount of data changes at the same time, due to the mip map, which causes noticeable artifacts. In [Dem04], the authors added noise in an attempt to make the artifacts less noticeable; the visual quality of the resulting images was noticeably reduced.

3. Summed-Area Table Generation

In order to efficiently construct summed-area tables, we borrow a technique, called recursive doubling [DR77], often used in high-performance and parallel computing. Using recursive doubling, a parallel gather operation amongst n processors can be performed in only $log_2(n)$ steps, where a single step consists of each processor passing its accumulated result to another processor.

In a similar manner, our method uses the GPU to accumulate results so that only O(log n) passes are needed for summed-area table construction. To simplify the following description, we assume that only two texels can be read per pass. Later in the discussion we explain how to generalize the technique to an arbitrary number of texture reads per pass.

Our algorithm proceeds in two phases: first a horizontal phase, then a vertical phase. During the horizontal phase, results are accumulated along scan lines, and during the vertical phase, results are accumulated along columns of pixels. The horizontal phase consists of n passes, where $n = ceil(log2(imagewidth))$, and the vertical phase consists of m passes, where $m = ceil(log2(imageheight))$.

For each pass we render a screen-aligned quad that covers all pixels that do not yet hold their final sum and execute a fragment program on every covered pixel. The input image is stored in a texture named t_A. In the first pass of the horizontal phase we read two texels from t_A: the one corresponding to the pixel currently being computed and the one to the immediate left. Both are added together and stored into texture t_B.

For the second pass, we swap our textures so that we are reading from t_B and writing to t_A. Now the fragment program adds the texels corresponding to the one currently being computed and the one two pixels to the left. t_A now holds the sum of four pixels.

The third pass repeats this scheme, now reading from t_A and writing to t_B and summing two texels four pixels apart, resulting in the sum of eight pixels in t_B. This progression continues for the rest of the horizontal passes until all pixels are summed up in the horizontal direction. Note that in pass i the leftmost 2^i pixels already hold their final sum for the horizontal phase and thus are not covered by the quad rendered in this pass. Next the vertical phase proceeds in an analogous manner. Figure 3 shows the horizontal passes needed to construct a summed-area table of a 4x4 image. The following pseudo-code summarizes the algorithm.

$t_A = InputImage$
$n = log2(width)$
$m = log2(height)$
$//\ horizontal\ phase$
$for(i = 0; i < n; i = i + 1)$
  $t_B[x,y] = t_A[x,y] + t_A[x + 2^i, y]$
  $swap(t_A, t_B)$

$//\ vertical\ phase$
$for(i = 0; i < m; i = i + 1)$
  $t_B[x,y] = tA[x,y] + t_A[x, y + 2^i]$
  $swap(t_A, t_B)$

$//\ Texture\ t_A\ holds\ the\ result$

In practice, reading more than two texels per fragment, per pass is possible, which reduces the number of passes required to generate a summed-area table. Our current implementation supports reading 2, 4, 8, or 16 texels per fragment, per pass. This allows trading per-pass complexity with the number of rendering passes required. Adding 16 texels per pass enables us to generate a summed-area table from a 256x256 image in only four passes, two for the horizontal phase, and two for the vertical phase. As shown in Section 6, adjusting the per-pass complexity helps in optimizing summed-area generation speed for different input texture sizes. The following is the pseudo-code to generate a summed-area table when r reads per fragment are possible.

$t_A = Input\ Image$
$n = log_r(width)$
$m = log_r(height)$

$//\ horizontal\ phase$
$for(i = 0; i < n; i = i + 1)$
  $t_B[x,y] =$
    $t_A[x,y] +$
    $t_A[x + 1 * r^i, y] +$
    $t_A[x + 2 * r^i, y] +$
    $\cdots +$
    $t_A[x + r * r^i, y]$
  $swap(t_A, t_B)$
$//\ vertical\ phase\ similar\ to$
$//\ horizonal\ phase$
$//\ Texture\ t_A\ holds\ the\ result$

Note that near the left and bottom image borders the fragment program will fetch texels outside the image regions. To ensure correct summation of the image pixels, the texture units must be configured to use *clamp to border color* mode with the border color set to 0. This way texel fetches outside the image boundaries will not affect the sum. Alternatively, it is possible to render a black border around the input image and configure the texture units to use *clamp to edge* mode.

We have implemented our algorithm in both Direct3D and OpenGL, with similar results. In the OpenGL implementation we used a double buffered pbuffer to mitigate the cost of context switches. Instead of switching context between each pass, we simply swap the front and back buffers of the pbuffer. This allows us to efficiently *ping-pong* between two textures as results are accumulated. If implemented at the driver level, similar to the way that automatic mip-map generation is done, the costs of the passes could be mitigated even more.

4. Improving Computational Precision

A key challenge to the usefulness of the summed-area table approach is the loss of numerical precision, which can lead to significant noise in the resultant image. This section first discusses the source of such precision loss and then presents our approach to mitigating this problem. Example images are provided that demonstrate how our approach achieves significant reduction in noise: up to 31 dB improvement in signal-to-noise ratios.

4.1. Source of Precision Loss

One source of precision loss could come from the GPU's floating point implementation since current graphics hardware does not implement IEEE standard 754 floating point but, as shown by Hillesland [Hil05], current GPU implementations behave reasonably well.

The summed-area table approach can exhibit significant noise because certain steps in the algorithm involve computing the difference between two relatively large finite-precision numbers with very close values. This is especially true for pixels in the upper right portion of the image because the monotonically increasing nature of the summed-area function implies that the table values for that region are all quite high.

As an example, consider the images of Fig. 4, which are 256x256 images with 8-bit components. The middle and right columns show the image after being filtered through an "identity filter," i.e., a 1-bit filter kernel that is ideally supposed to produce a resultant image that is a replica of the original image. To avoid loss of computational precision, a summed-area table with 24 bits of storage per component per pixel would be sufficient, since the maximum summed-area value at any pixel cannot exceed 256x256x256. However, the summed-area table used in this example used only 16 and 24 bit FP values. As a result, significant noise is seen in the filtered image, with worsening image quality in the direction of increasing xy.

4.2. Our Approach to Improving Precision

In order to mitigate the loss of computational precision, our approach modifies the original summed-area table computation in two ways.

4.2.1. Using Signed-Offset Pixel Representation

The first modification is to represent pixel values in the original image as *signed* floating-point values (e.g., values in the range -0.5 to 0.5), as opposed to the traditional approach that uses unsigned pixel values (from 0.0 to 1.0).

This modification improves precision in two ways: (i) there is a 1-bit gain in precision because the sign bit now becomes useful, and (ii) the summed-area function becomes non-monotonic, and therefore the maximum value reached has a relatively lower magnitude.

We have investigated two distinct methods for converting the original image to a signed-offset representation: (i) centering the pixel values around the 50% gray level, and (ii) centering them around the mean image pixel value. The former involves less computational overhead and gives good precision improvement, but the latter provides even better results with modest computational overhead.

Centering around 50% gray level. This method modifies the original image by subtracting 0.5 from the value at every pixel, thereby making the pixel values lie in the -0.5 to 0.5 range. The summed-area table computation proceeds as usual, but with the understanding that the table entry at pixel position (x,y) will now be 0.5xy less than the actual summed-area value. The net impact is a significant gain in precision because the table entries now have significantly lower magnitudes, and therefore computing the differences yields a greater precision result.

Fig. 4 demonstrates the usefulness of this approach. The

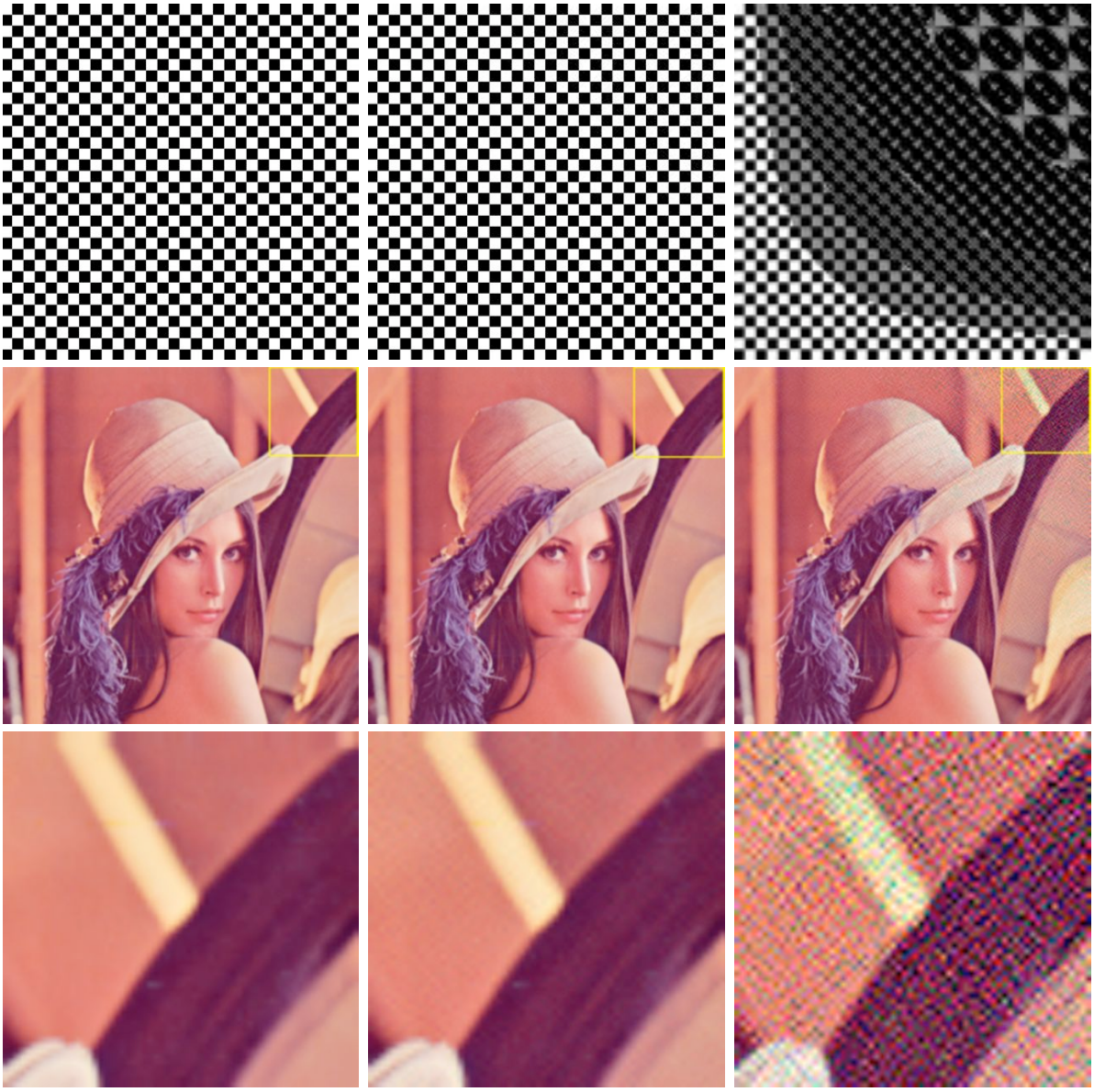

Figure 4: *The left column shows the original input images, the middle column are reconstructions from summed-area tables (SATs) generated using our method, and the right column are reconstructions from SATs generated with the old method. For the first row, the SATs are constructed using 16 bit floats, for the second row the SATs are constructed using 24 bit floats, and the final row shows a zoomed version of second row (region-of-interest highlighted)*

first row shows three versions of a checkerboard. The image on the right, generated using the traditional method, exhibits unacceptable noise throughout much of the image. In contrast, the middle image, generated by our method, barely shows error.

Centering around image pixel average. While centering pixel values around the 50% gray level proved to be quite useful, an even better approach is to store offsets from the image's average pixel value. This is especially true of images such as Lena for which the image average can be quite different from 50% gray. For such images, centering around 50% gray could still result in sizable magnitudes at each pixel position, thereby increasing the probability that the summed-area values could appreciably grow in magnitude. Centering the pixel values around the actual image average guarantees that the summed-area value is equal to 0 both at the origin and at the upper right corner (modulo floating-point rounding errors).

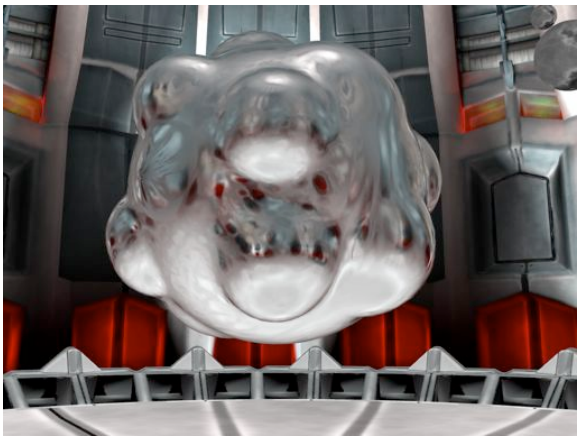

Figure 5: *Environment map filtered with a spatially varying filter.*

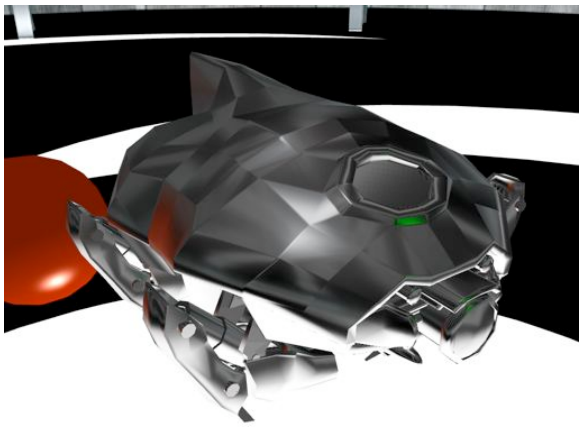

Figure 6: *An object textured using four samples from a pair of summed-area tables generated from an environment map in real-time.*

The computational overhead of this approach is fairly modest as the image average is easily computed in hardware using mip mapping.

4.2.2. Using Origin-Centered Image Representation

The second modification involves anchoring the origin of the coordinate system to the center of the image, instead of to the bottom-left image corner. In effect, this simple modification reduces in half the maximum values of x and y over which summed areas are accumulated. As a result, for a given precision level, images of double the width and height can be handled.

5. Example Applications

Since our technique is fast enough to generate summed-area tables every frame, their use becomes feasible to generate real-time, interactive effects. We present four example applications. The first is a method to generate glossy environmental reflections. The second application uses a summed-area table to render glossy planar reflections, where the blurriness of an object's reflection varies depending on its distance from the reflecting plane. The third application presents a technique to render glossy transparency, and finally, the fourth application, previously presented by Greene [Gre03], renders images with a depth-of-field effect. We believe that these applications are a compelling demonstration of the power of real-time summed-area table generation.

5.1. Glossy Environmental Reflections

In [KVHS00], Kautz et al. presented a method for real-time rendering of glossy reflections for static scenes. They rendered a dual-paraboloid environment map and pre-filtered it in an offline process. Instead of pre-filtering, we create a summed-area table for each face of a dual-paraboloid map on the fly, and use them to filter the environment map at run time. This enables real-time, interactive environmental glossy reflections for dynamic scenes.

Figure 5 is an image of an object where the environment map has been filtered with a spatially varying filter function; in this case the filter support has been modulated by another texture. The image is rendered in real time, at a rate of over 60 frames per second. The filter function, scene geometry and environment map can change every frame.

There are several compelling reasons for using dual-paraboloid environment mapping over the more commonly used cube mapping. First, Kautz et al. showed that when filtering in image space, as opposed to filtering over the solid angles, a dual-paraboloid environment map has lower error than a cube map or a spherical map. Second, it is only necessary to generate two summed-area tables as opposed to six summed-area tables. Finally, for large filters, a dual-paraboloid map will require data from only two textures, whereas it is possible that data might be required from all six faces of a cube map.

5.1.1. Box Filtering

A coarse approximation to a glossy BRDF is a simple box filter. A single box-filter evaluation takes four texture reads from the summed-area table. Two evaluations are required on current hardware when a filter is supported by both the front and the back of a dual-paraboloid map. On future hardware it may be possible to conditionally evaluate the filters for both maps only when necessary.

As is common when storing a spherical map in a square texture, our implementation uses the alpha channel to mark the pixels that are in the dual-paraboloid map. A pixel is considered to be in the map if its alpha value is one. We also use the alpha value to count the area covered by the filter. After combining the result of the evaluation from the front and back maps, the alpha channel holds the total count of

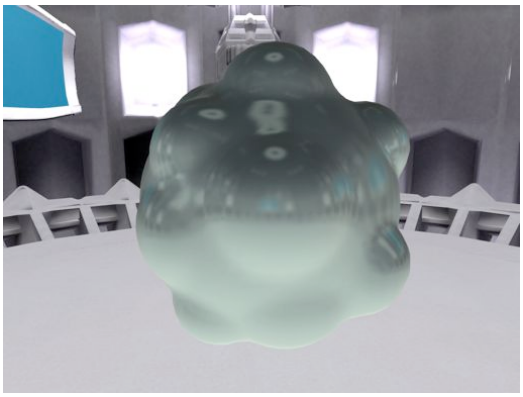

Figure 7: *A set of four box filters stacked to approximate a Phong BRDF.*

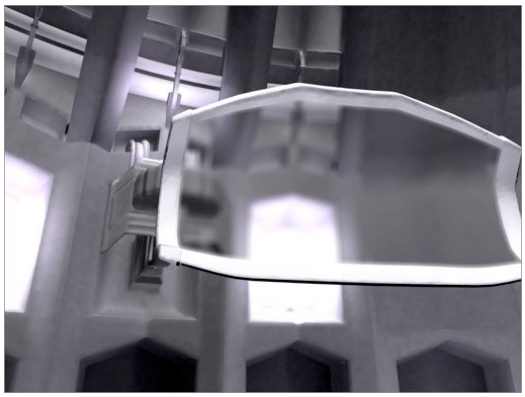

Figure 8: *Example of translucency using a summed-area table to filter the view seen through the glass.*

summed texels, which is then used to normalize the filter value.

The basic algorithm for rendering glossy environmental reflections is

```
renderCubeMap();
generateParaboloidMapFromCubeMap();
generateSummedAreaTable( FrontMap );
generateSummedAreaTable( BackMap );
setupTextureCoordinateGeneration();

renderScene
{
  foreach fragment on reflective object:
  {
    front = evaluateSAT(FrontSAT, filter_size);
    front = evaluateSAT(BackSAT, filter_size);

    // computer filter area
    filtered.alpha = front.alpha + back.alpha;

    // combine front and back color
    result = front + back;

    // divide by the area of the filter
    result /= filtered.alpha;

    computeFinalColor(filtered);
  }
}
```

While our current implementation creates a dual-paraboloid map from a cube map, it is possible to directly generate the dual-paraboloid map by using a vertex program to project the scene geometry as in [CHL04].

5.1.2. Box Filtering

More complex filter functions can be constructed at the cost of more texture reads by *stacking* multiple box filters on top of each other. The stacked boxes approximate the shape of smoother filters. For a single summed-area table, each filter in the stack requires eight texture reads (four for each of the front and back maps). So a complex filter created from a stack of four box filters would perform thirty-two texture reads per fragment.

Both OpenGL and Direct3D provide a means to automatically generate texture coordinates based on the normal direction and reflection direction. By combining box filters generated from both the reflection direction and the normal direction, it is possible to compute an approximation of the Phong BRDF. Figure 7 shows an image generated using a stack of two large box filters centered on the normal direction to approximate the diffuse component of the Phong BRDF and a stack of two smaller box filters centered on the reflection direction to approximate the specular component.

5.2. Glossy Planar Reflections

Since the summed-area table enables filtering with arbitrary support, it is relatively easy to render glossy reflections where the blurriness of an object varies depending on the distance of the reflected object from the reflector. This effect is often seen when an object is placed on a glossy table top. The object's reflection is much sharper where the object and table top meet than elsewhere. Figure 1 shows an image where the floor is a glossy reflector, and the blurriness of the reflection depends on the object's distance from the floor.

The effect is accomplished by augmenting the standard planar reflection algorithm. The pass for rendering the reflected scene from the virtual viewpoint outputs both the color and the distance to the reflection plane to a texture. A summed-area table is generated from the color data. Then the planar reflector is rendered from the summed-area table, using the previously saved distance to modulate the filter width.

5.3. Translucency

Approaches to rendering translucent materials include those of [Arv95, Die96]. We are able to render real-time interac-

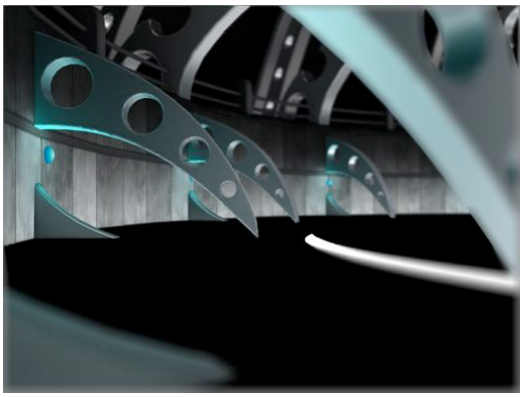

Figure 9: *Simulated depth of field.*

tive translucent objects using a summed-area table; this technique can be used to render such effects as etched and milky glass. Figure 8 shows a scene with multiple translucent objects.

The effect is achieved by first rendering the scene, to texture memory, without the translucent objects. A summed-area table is generated from the resulting image. Then we render the translucent objects with a fragment program that uses the summed-area table to blur the regions of the scene behind the objects.

5.4. Depth of Field

In [Gre03], Greene presents a technique to render an image with a depth-of-field effect using summed-area tables. His summed-area table generation technique is problematic since it requires that a texture be read from and written to at the same time. Unfortunately graphics hardware — due to its parallel streaming architecture — makes no guarantees about the execution sequence of read-modify-write operations.

In [Demers04], a technique to render a depth of field effect was presented that used mip maps to approximate a simple filter. Because of the artifacts introduced by the mip-map filtering technique, the authors add noise to reduce the perceptible Mach bands.

Unlike mip maps, summed-area tables are able to average arbitrary rectangular sections of an image, allowing us to implement a real-time, interactive version of the depth-of-field effect, without having to add noise to mask filtering artifacts. However, our implementation does have the same drawbacks as other image filtering techniques for generating a depth-of-field effect, such as the bleeding of sharp in-focus objects onto blurry backgrounds. Figure 9 shows an image rendered with depth of field. A 1024x768 image renders at a rate of 23 frames per second. A lower resolution version renders at much higher frame rates.

The effect is accomplished by first rendering the scene from the camera's point-of-view and saving the color and depth buffers to texture memory. Next a summed-area table is generated from the saved color buffer. As in [Dem04], the depth buffer is used to determine the circle of confusion. Finally, a screen-filling quad is rendered, and a fragment program is used to blur the color buffer based on the circle of confusion.

	Summed-area table size		
	256x256	512x512	1024x1024
Radeon 9800 XT[1]	3.1 ms (8)	14.2 ms (4)	70.1 ms (4)
Radeon X800XT PE[1]	1.4 ms (8)	7.3 ms (4)	36.2 ms (4)
Geforce 6800 Ultra[2]	4.3 ms (8)	32.4 ms (4)	95.3 ms (4)

[1] $24-bit\ floats$ [2] $32-bit\ floats$

Table 1: *Shortest time to generate summed-area tables of different sizes. The number of samples per pass are given in parentheses.*

	Summed-area table size		
Samples/pass	256x256	512x512	1024x1024
2	2.3 ms	9.9 ms	44.3 ms
4	1.8 ms	7.3 ms	36.2 ms
8	1.4 ms	9.9 ms	45.6 ms
16	2.7 ms	12.4 ms	53.3 ms

Table 2: *Time to generate summed-area tables of different sizes using different number of samples per pass on a Radeon X800XT Platinum Edition graphics card.*

6. Summed-Area Table Generation Performance

Table 1 shows the time required to generate summed-area tables of different sizes on a number of graphics cards using DirectX 9. We list the shortest time we could achieve for each card and input size along with the number of samples per pass used to get the best performance. Table 2 shows performance based on input size and the number of samples per pass for one of the cards used in our test.

Our benchmark results show that finding a good balance between the number of rendering passes and the amount of work performed during each pass is important for the overall performance of summed-area table generation. The optimal tradeoff between the number of passes and per-pass cost is largely dependent on the overhead of render target switches and the design of the texture cache on the target platform.

Computing summed-area tables directly on the graphics card is better than performing this computation on the CPU for several reasons. First, the input data is already present in GPU memory. Transferring the data to the CPU for pro-

cessing and then and back again would put an unnecessary burden on the bus and can easily become a bottleneck because many graphics drivers are unable to reach full theoretical bandwidth utilization when reading back data from the GPU [GPU]. Moreover, moving data back and forth between GPU and CPU would break GPU-CPU parallelism because each processor would end up waiting for new results from the other processor.

In our opinion, the particularly good performance of generating 256x256 summed-area table on modern graphics hardware makes dynamic glossy reflections using dual-paraboloid maps (as outlined in Section 5) very feasible.

7. Future Work

In the future we plan to quantify how closely a set of stacked box filters can approximate an arbitrary BRDF, and develop a set of criteria to generate the box-filter stack that best represents a given BRDF. While the techniques presented in this paper substantially reduce the precision requirements of summed-area tables, work is needed on techniques to reduce them even further. Doing so will make it feasible to generate second and third order summed-area tables, which would allow more complex filter functions, such as a Barlett filter or a parabolic filter.

8. Conclusion

We have introduced a technique to rapidly generate summed-area tables, which enable constant-time space varying box filtering. This capability can be used to simulate a variety of effects. We demonstrate glossy environmental reflections, glossy planar reflections, translucency, and depth of field.

The biggest drawback to summed-area tables is the high demand they make on numerical precision. To ameliorate this problem, we have developed some techniques to more effectively use the limited precision available on current graphics hardware.

Acknowledgements

Financial support was provided by an ATI Research fellowship, the National Science Foundation under grants CCF-0306478, and CCF-0205425. Equipment was provided by NSF grant CNS-0303590. We would like to thank Eli Turner for the artwork used in our demo application, and Paul Cintolo who helped us with gathering performance data.

References

[AG02] ASHIKHMIN M., GHOSH A.: Simple blurry reflections with environment maps. *J. Graph. Tools 7*, 4 (2002), 3–8. 3

[Arv95] ARVO J.: Applications of irradiance tensors to the simulation of non-lambertian phenomena. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1995), ACM Press, pp. 335–342. 7

[CHL04] COOMBE G., HARRIS M. J., LASTRA A.: Radiosity on graphics hardware. In *GI '04: Proceedings of the 2004 conference on Graphics interface* (School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004), Canadian Human-Computer Communications Society, pp. 161–168. 7

[Cro84] CROW F. C.: Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1984), ACM Press, pp. 207–212. 1, 2

[Dem04] DEMERS J.: *GPU Gems*. Addison Wesley, 2004, pp. 375–390. 3, 8

[Die96] DIEFENBACH P.: *Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions*. PhD thesis, University of Pennsylvania, Philadelphia, 1996. 7

[DR77] DUBOIS P., RODRIGUE G.: An analysis of the recursive doubling algorithm. In *High Speed Computer and Algorithm Organization*. 1977, pp. 299–305. 3

[GPU] Gpubench: http://graphics.stanford.edu/projects/gpubench/. 9

[Gre03] GREENE S.: Summed area tables using graphics hardware. Game Developers Conference, 2003. 6, 8

[Hec86] HECKBERT P. S.: Filtering by repeated integration. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (New York, NY, USA, 1986), ACM Press, pp. 315–321. 1, 2

[Hil05] HILLESLAND K.: *Image Streaming to build Image-Based Models*. PhD thesis, University of North Carolina at Chapel Hill, 2005. 4

[KVHS00] KAUTZ J., VAZQUEZ P.-P., HEIDRICH W., SEIDEL H.-P.: A unified approach to prefiltered environment maps. In *Eurographics Workshop on Rendering* (2000). 6

[YP03] YANG R., POLLEFEYS M.: Multi-resolution real-time stereo on commodity graphics hardware, 2003. 3