CHAPTER 25

TEMPORALLY RELIABLE MOTION VECTORS FOR BETTER USE OF TEMPORAL INFORMATION

Zheng Zeng,[1] Shiqiu Liu,[2] Jinglei Yang,[3] Lu Wang,[1] and Ling-Qi Yan[3]
[1]Shandong University
[2]NVIDIA
[3]University of California, Santa Barbara

ABSTRACT

We present temporally reliable motion vectors that aim at deeper exploration of temporal coherence, especially for the generally believed difficult applications on shadows, glossy reflections, and occlusions. We show that our temporally reliable motion vectors produce significantly more robust temporal results than current traditional motion vectors while introducing negligible overhead.

25.1 INTRODUCTION

The state-of-the-art reconstruction methods [2, 9, 7, 10] all rely on temporal filtering. Though it is demonstrated to be powerful, robust temporal reuse has been very challenging due to the fact that motion vectors are sometimes not valid. For example, a static location in the background may be blocked by a moving object in the previous frame. In this case, the motion vector does not exist at the current location. Also, the motion vectors may be wrong for effects like shadows and reflections. For example, a static shadow receiver will always have a zero-length motion vector, but the shadows casted onto it may move arbitrarily along with the light source. In any of these cases, when correct motion vectors are not available but temporal filtering is applied anyway, ghosting artifacts (unreasonable leak or lag of shading over time) will emerge.

Although the temporal failures can be detected with smart heuristics [10], the temporal information in these cases will be *simply rejected* nonetheless. But we believe that the information can be *better utilized*. In this chapter, we

A. Marrs, P. Shirley, I. Wald (eds.), *Ray Tracing Gems II*, https://doi.org/10.1007/978-1-4842-7185-8_25

present different types of motion vectors for different effects, to make the seemingly unusable temporal information available again. Specifically, we introduce the following:

- A shadow motion vector for moving shadows.
- A stochastic glossy reflection motion vector for glossy reflections.
- A dual motion vector for occlusions.

25.2 BACKGROUND

In this section, we briefly go over the calculation of traditional motion vectors, and explain how they are used in temporal filtering.

When two consecutive frames $i-1$ (previous) and i (current) are given, the idea of back-projection is to find for each pixel X_i intersected by the primary ray, where its world-space shading point S_i was in the previous frame at X_{i-1}. Because we know the entire rendering process, the back-projection process can be accurately computed: first, project the pixel X_i back to its world coordinate in the ith frame, then transform it back to the $(i-1)$-th frame according to the movement of the geometry, and finally project the transformed world coordinate in the $(i-1)$-th frame back to the image space to get X_{i-1}. Denote $\mathbf{P} = \mathbf{M}_v \mathbf{M}_{mvp}$ as the viewport $\mathbf{M}_v$ times the model-view-projection transformation $\mathbf{M}_{mvp}$ per frame and $\mathbf{T}$ as the geometry transformation between frames; then, the back-projection process can be formally written as

$$X_{i-1} = \mathbf{P}_{i-1}\mathbf{T}^{-1}\mathbf{P}_i^{-1}X_i, \tag{25.1}$$

where the subscripts represent different frames. According to Equation 25.1, the motion vector $\mathbf{m}(X_i) = X_{i-1} - X_i$ is defined as the difference between the back-projected pixel and the current pixel in the image space. The following pseudocode shows how this traditional motion vector is calculated:

```
calcTradMotionVector(uint2 pixelIndex, float4 hitPos, uint hitMeshID){
    // For each pixel X_i (in image space) ...
    float2 X = pixelIndex + float2(0.5f, 0.5f);
    // ... find its world-space shading point S_i.
    float4 S = hitPos;
    // Then, transform S_i back to the previous frame to get S_(i-1).
    // The custom function getT(...) returns the geometry transformation for
         a given mesh, and inverse(...) inverts a 4x4 matrix.
    float4x4 invT = inverse(getT(hitMeshID));
    float4 prevS = mul(S, invT)
    // Finally, project it to screen space to get X_(i-1).
    // The custom function toScreen(...) projects a given world-space point
        to screen space to get corresponding the image-space pixel.
```

```
12      float2 prevX = toScreen(prevS);
13      // Return traditional motion vector.
14      return prevX - X;
15  }
```

The motion vector for each pixel X_i is computed together with the rendering process and can be acquired almost without any performance overhead. With the motion vectors, temporal filtering becomes straightforward. In practice, it is a simple linear blending between the current and previous pixel values:

$$\bar{c}_i(X_i) = \alpha \cdot \tilde{c}_i(X_i) + (1 - \alpha) \cdot \bar{c}_{i-1}(X_{i-1}), \tag{25.2}$$

where c is the pixel value, first filtered spatially per frame resulting in $\tilde{c}$, then blended with the pixel value of its previous correspondence $\bar{c}_{i-1}$. The α is a factor between 0 and 1 that determines how much temporal information is trusted and used, usually set to 0.1 to 0.2 in practice, indicating heavy temporal dependence on previous frames. The temporal filtering process continues as more and more frames are rendered, accumulating to a cleaner result. Thus, we use the $\bar{}$ and $\tilde{}$ symbols to indicate less and more noise, respectively.

From Equation 25.2, we can see that one frame's contribution over time is an exponential falloff. Thus, if the motion vectors cannot accurately represent correspondence between adjacent frames, ghosting artifacts will appear. Various methods are designed to alleviate the temporal failure. Salvi [8] proposed to clamp the previous pixel value $\bar{c}_{i-1}(X_{i-1})$ to the neighborhood of the current pixel value, in order to suppress the ghosting artifacts and provide a faster rate of convergence to the current frame. The spatiotemporal variance-guided filtering (SVGF) method [9] focuses on a better spatial filtering scheme to acquire $\tilde{c}_i(X_i)$ by considering spatial and temporal variances together. And the adaptive SVGF (A-SVGF) method [10] detects rapid temporal changes to adjust the blending factor α to rely more or less on spatial filtering, trading ghosting artifacts for noise.

25.3 TEMPORALLY RELIABLE MOTION VECTORS

In this section, we describe our temporally reliable motion vectors. Specifically, we will focus on the three commonly encountered temporal failure cases: shadows, glossy reflections, and occlusions. In contrast to the previous methods, we intend to better utilize the previous information, i.e., we would like to find a more reliable $\bar{c}_{i-1}(X_{i-1})$ so that minimal special treatment is further needed. Our insight is that for shadows and glossy reflections, it is

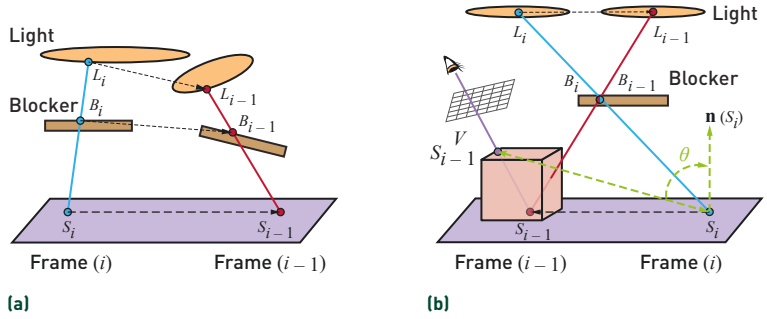

Figure 25-1. *(a) Computation of our shadow motion vectors. (b) The nonplanar shadow receiver issue.*

not the geometry in a pixel that we want to track in the previous frame, but the position of the shadow and the reflected virtual image. For occlusions, it is easier for the previously occluded regions in the background to find correspondences also in the background rather than on the occluder. After introducing the corresponding motion vectors, we then compare them with the state-of-the-art methods and report the computation cost.

25.3.1 SHADOWS

Inspired by percentage closer soft shadows (PCSS) [4], which estimate the shadow size based on the average blocker depth and light size, we propose to track the movement of shadows by following the blocker and light positions over time.

Figure 25-1a illustrates our scheme focusing on two consecutive frames ($i-1$) and i. We shoot one shadow ray per pixel toward a randomly chosen position on the light. For a pixel X_i (in image space) in shadow, we know exactly its shading point S_i, the blocker position B_i, and the light sample position L_i (all in world space). Because the blocker and the light sample positions are associated with certain objects, we immediately know their transformation matrices between these two frames, so we are able to find their world-space positions B_{i-1} and L_{i-1}, respectively, in the ($i-1$)-th frame. If the geometry around the shadow receiver S_i is a locally flat plane, we can find the intersection S_{i-1} between this plane and the line connecting L_{i-1} and B_{i-1} in the previous frame. Finally, we project this intersection to the screen space. In this way, this projected pixel X^V_{i-1} is our tracked shadow position from X_i:

$$X^V_{i-1} = \mathbf{P}_{i-1}\ \text{intersect}[\mathbf{T}^{-1}L_i \rightarrow \mathbf{T}^{-1}B_i; \mathbf{T}^{-1}\text{plane}(S_i)], \tag{25.3}$$

where the transformations **T** between frames can be different for the light sample, blocker, and shading point.

Equation 25.3 implies that our shadow motion vector is $\mathbf{m}^V(X_i) = X^V_{i-1} - X_i$. The pseudocode for calculating the shadow motion vector looks as follows:

```
calcShadowMotionVector(uint2 pixelIndex, float4 hitPos, float4 hitNormal,
     uint hitMeshID, float4 blockerPos, uint blockerMeshID, float4 lightPos,
      uint lightMeshID){
    // For each pixel X_i in shadow (in image space) ...
    float2 X = pixelIndex + float2(0.5f, 0.5f);
    // ... find its shading point S_i, blocker point B_i, and light sample
         point L_i (all in world space).
    float4 S = hitPos;
    float4 B = blockerPos;
    float4 L = lightPos;
    // Then, transform B_(i-1) and L(i-1) back to the previous frame.
    // The custom function getT(...) returns the geometry transformation for
          a given mesh, and inverse(...) inverts a 4x4 matrix.
    float4 prevB = mul(B, inverse(getT(blockerMeshID)));
    float4 prevL = mul(L, inverse(getT(lightMeshID)));
    // Next, find the intersection of the virtual plane (defined by S_i and
         its normal in the previous frame) and the ray (from L_(i-1) to B_(i
         -1)).
    float4 origin = prevL;
    float4 direction = prevB - prevL;
    float4x4 invT = inverse(getT(hitMeshID));
    float4 prevNormal = mul(hitNormal, invT);
    float4 prevS = mul(S, invT);
    // The custom function rayPlaneIntersect(...) finds the intersection
         between a ray (defined by origin and direction) and a plane (
         defined by point and normal).
    float4 intersection = rayPlaneIntersect(origin, direction, prevS,
         prevNormal);
    // Finally, project it to screen space to find X^(V)_(i-1).
    // The custom function toScreen(...) projects a given world-space point
         to screen space to get corresponding the image-space pixel.
    float2 prevX = toScreen(intersection);
    // Return the shadow motion vector.
    return prevX - X;
}
```

To use our shadow motion vectors, we slightly modify the temporal filtering Equation 25.2 by adding a lightweight clean-up filtering pass (at most 9×9) after temporal blending. This is because the motion vectors $\mathbf{m}^V(X_i)$ can be noisy due to random sampling on the light, so the fetched $\bar{V}_{i-1}(X_{i-1})$ can be noisy as well, despite the smoothness of $\bar{V}_{i-1}$ itself in the previous frame. The same clean-up filter will be used for glossy reflections and occlusions.

To perform spatial filtering of the noisy shadows in the current frame i, we refer to Liu et al. [6], which accurately calculates the filter size. However, we notice that based on the previous computation, only those pixels in shadows in

the current frame are associated with our shadow motion vectors. To deal with this problem, and to achieve both efficient filtering performance and clean shadow boundaries, we conceptually interpret the filtering of shadows as the splatting of each in-shadow pixel's visibility, along with other associated properties.

DISCUSSION: NONPLANAR SHADOW RECEIVER The only assumption we make is that the geometry is locally flat for each shading point during the computation of our shadow motion vectors. However, as Figure 25-1b shows, after the back-projection in the $(i-1)$-th frame, it is possible that S_{i-1}^V, the shading point of pixel X_{i-1}^V, is not on the virtual receiver plane defined by S_i and its normal $\mathbf{n}(S_i)$ (inverse transformed if the shadow receiver moves over time, omitted here for simplicity). These two shading points may not have the same normals and could even be on different objects. In this case, it seems that our shadow motion vector could no longer be used.

To deal with the problem introduced because of nonplanar shadow receivers, we introduce a simple but effective falloff heuristic. That is, we measure the extent of the "nonplanarity." Figure 25-1b illustrates our idea. Once X_{i-1}^V is calculated, we measure the angle θ between the normal of the virtual receiver plane $\mathbf{n}(S_i)$ and the direction $S_i \rightarrow S_{i-1}^V$. Our key observation is that, only when θ is close to 90°, we can fully depend on our shadow motion vector. Otherwise, we should trust more on the spatially filtered result. So, we use θ to adjust the α, replacing it with a specific α^V in Equation 25.2 as

$$\alpha^V = 1 - G\left(\theta - \frac{\pi}{2}; 0, 0.1\right) \cdot (1 - \alpha), \tag{25.4}$$

where $G(x; \mu, \sigma)$ is a Gaussian function with its peak value normalized to 1, centered at μ, and with a standard deviation of σ. Finally, we achieve high-quality, non-lagging shadows. This process looks as follows:

```
// Once X^(V)_(i-1) is calculated ...
float2 prevX = X + shadowMotionVector;
// ... measure the angle theta between the normal of S_i (in the previous
    frame) and the direction S_i->S^V_(i-1).
float4 S = hitPos;
// The custom function getPrevHitPos(...) returns the world-space shading
    point for a given pixel in the previous frame.
float4 prevSV = getPrevHitPos(prevX);
float3 direction = normalize(float3(prevSV - S));
// The custom function getT(...) returns the geometry transformation for a
    given mesh, and inverse(...) inverts a 4x4 matrix.
float3 prevNormal = float3(hitNormal, inverse(getT(hitMeshID)));
float theta = acosf(dot(direction, prevNormal))
// Use theta to adjust the alpha^V.
```

```
12 // The custom function gaussian(...) returns a value between 0 to 1
       according to theta.
13 float alphaV = 1.0f - gaussian(theta - PI / 2, 0.f, 0.1f) * (1.0f - alpha);
```

We compare our results with the ones generated using traditional motion vectors, with and without the neighborhood clamping approach used in temporal antialiasing (TAA) [8]. The clamping methods represent the line of ideas that force the use of temporal information. We also compare our method with the SVGF and A-SVGF methods as representatives that balance the use of temporal and spatial information. Neither kind of these methods aims at better utilizing the temporal information.

Figure 25-2 shows the *fence* scene with a rapidly moving fence in front and an area light behind it. In this example, we demonstrate that our shadow motion vectors are able to produce shadows that are closely attached to the fence.

In comparison, traditional motion vectors produce significant ghosting artifacts. This is expected because they will always be zero in this case. With clamping, the results are less lagging but much more noisy. However, the noise is aggressively filtered by SVGF, resulting in overblur. The A-SVGF method discards temporal information, resulting in color blocks similar to the typical "smearing" artifact in bilateral image filtering and leaving behind low-frequency noise that can be easily observed in a video sequence.

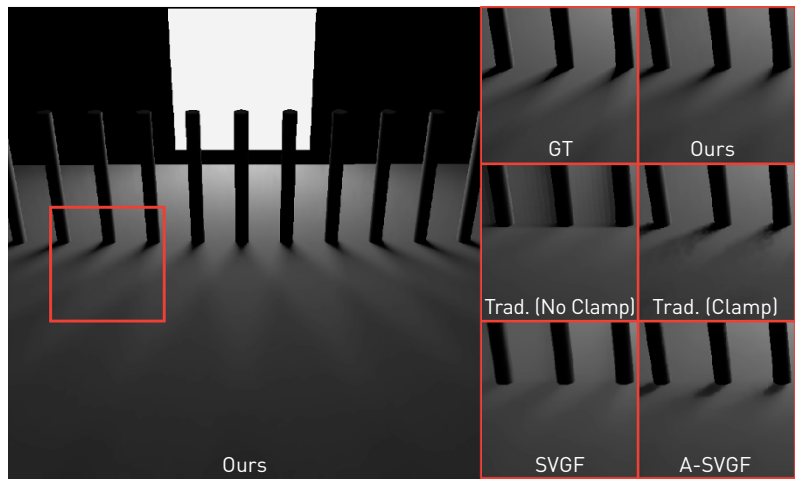

Figure 25-2. *The fence scene with a rapidly moving fence in front and an area light behind it. Our shadow motion vectors are able to produce shadows that are closely attached to the fence.*

25.3.2 GLOSSY REFLECTIONS

Similar to tracking the shadows, we can also track the movement of glossy reflections.

Our insight is that no matter whether we are using the specular or sampled direction, what we need to do is still find the corresponding pixel in the previous frame of the secondary hit point H_i. However, because glossy BRDFs model a non-delta distribution, multiple pixels may reflect to the same hit point, forming a finite area in the screen space. This indicates that there will be multiple pixels from the previous frame that correspond to X_i in the current frame. Our stochastic glossy motion vector aims at finding one at a time.

Given that the center of a glossy BRDF lobe is usually the strongest, we always have a valid choice with which to start. That is, we first assume that the glossy BRDF degenerates to pure specular, then we can immediately find the corresponding point S^C_{i-1} similar to Zimmer et al. [12]. Then, our insight is that, as the glossy lobe gradually emerges, a region will appear around S^C_{i-1} in which all the points are able to reflect to the same hit point H_{i-1}. This region can be approximated by tracing a glossy lobe (with the same roughness at S^C_{i-1}) from the virtual image of H_{i-1} toward S^C_{i-1}.

Figure 25-3 illustrates the way that we find one corresponding pixel X^R_{i-1} in the previous frame. We start from the importance-sampled secondary ray at the shading point S_i and the secondary hit point H_i in the world space. We transform H_i to the previous frame $(i - 1)$ in the world space, find its mirror-reflected image, then project it to the screen space to retrieve S^C_{i-1} in the world space again.

Then, similar to the shadow case, we assume a locally flat virtual plane around S^C_{i-1} and find the intersected region between this plane and the glossy lobe traced from the image of H_{i-1} toward S^C_{i-1}. In practice, there is no need to trace any cones, and we simply assume that the glossy lobe is a Gaussian in directions and that the intersected region is a Gaussian in positions as well as in the image space, which can be efficiently approximated by tracking the endpoints of major and minor axes. In this region, our stochastic motion vector for glossy reflection randomly finds X_i's correspondence at

$$X^R_{i-1} = \text{sample}\left(\mathbf{P}_{i-1}\ \text{mirror}[\mathbf{T}^{-1}H_i, \mathbf{T}^{-1}\text{plane}(S_i)], \Sigma\right), \tag{25.5}$$

where sample(μ, Σ) importance-samples a Gaussian function with center μ and covariance Σ, and $\mathbf{T}$ still represents different transformations at different

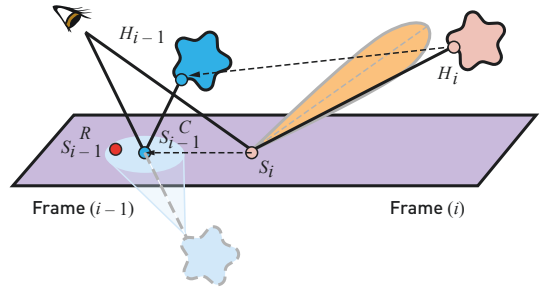

Figure 25-3. *The computation of our stochastic glossy reflection motion vectors. For an importance-sampled secondary ray, we back-project the virtual image of its hit point in the previous frame.*

places. The pseudocode for calculating our stochastic glossy motion vector looks as follows:

```
calcGlossyMotionVector(uint2 pixelIndex, float4 hitPos, float4 hitNormal,
     uint hitMeshID, float4 secondaryHitPos, uint secondaryHitMeshID){
    // For each pixel X_i, find the secondary hit point H_i.
    float2 X = pixelIndex + float2(0.5f, 0.5f);
    float4 H = secondaryHitPos;
    // Transform H_i back to the previous frame.
    // The custom function getT(...) returns the geometry transformation for
         a given mesh, and inverse(...) inverts a 4x4 matrix.
    float4 prevH = mul(H, inverse(getT(secondaryHitMeshID)));
    // Then, find its mirror-reflected image using the plane defined by S_i
        and its normal in the previous frame.
    float4 S = hitPos;
    float4x4 invT = inverse(getT(hitMeshID));
    float4 prevNormal = mul(hitNormal, invT);
    float4 prevS = mul(S, invT);
    // The custom function mirror(...) finds the mirror-reflected image
        behind a given plane for a given point.
    float4 mirroredPrevH = mirror(prevH, prevS, prevNormal);
    // Next, project it to screen space to find the "center."
    // The custom function toScreen(...) projects a given world-space point
        to screen space to get the corresponding image-space pixel.
    float2 center = toScreen(mirroredPrevH);
    // Finally, sample one position in image space around this center.
    // The key idea here is to simply approximate a 2D anisotropic Gaussian
        on the plane, then project it to screen space.
    // Assume that the glossy lobe is a Gaussian in directions with variance
         sigma (using custom function GaussianDist(...)); then, we can
        immediately find a 2D Gaussian on the plane according to the
        distance from the lobe to the plane (using custom function toPlane
        (...)).
    float2x2 covariance = toPlane(gaussianDist(sigma), prevS, prevNormal)
    // Then, project it to screen space to get the covariance of this 2D
        anisotropic Gaussian on screen.
    covariance = toScreen(covariance);
```

```
    // The custom function sampleGaussian2D(...) samples a 2D point in image
         space according to the Gaussian function with given center and
        covariance.
    float2 prevX = sampleGaussian2D(center, covariance);
    // Return the glossy motion vector.
    return prevX - X;
}
```

The usage of our stochastic glossy reflection motion vectors is similar to the shadow motion vectors. Also, the "natural hierarchy" still exists, i.e., when the roughness is high, the glossy reflection motion vectors will be more noisy, but the temporally filtered result will be then spatially cleaned up in a more aggressive manner.

We show the *sun temple* scene in Figure 25-4, which contains glossy reflections of various objects. We compare our method with four approaches: (1) using the traditional reflectors' motion vectors but without clamping of previous pixel values, (2) using traditional motion vectors with clamping, (3) using traditional motion vectors with the temporal component of A-SVGF (the temporal gradient method), and (4) using specular reflected rays' hit points' motion vectors, also with clamping. For all the comparisons, we use the spatial filter discussed in [6] as the spatial component of our denoising pipeline.

The comparison indicates that our glossy reflection motion vectors do not introduce ghosting artifacts. However, with traditional motion vectors, naive filtering produces significant ghosting. Clamping relieves the lagging but introduces severe discontinuous artifacts. With specular motion vectors, the results look plausible in most regions, but discontinuous artifacts can still be found around the edges of the reflected objects. The A-SVGF will always result in noisy results because the temporal gradient changes so fast that it

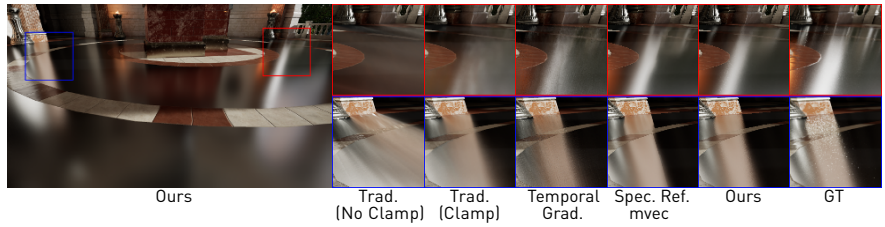

Figure 25-4. *The sun temple scene with a rapidly moving camera. Our stochastic glossy reflection motion vector is able to produce accurate reflections, whereas the traditional motion vectors result in significant ghosting artifacts.*

mostly uses only the noisy current frame. Our method produces the closest result to the ground truth (ray traced without firefly removal, thus always looks brighter).

25.3.3 OCCLUSIONS

Different from the previous cases on shadows and glossy reflection, when occlusion happens, in theory there are no temporal correspondences X^0_{i-1} of the pixels X_i appearing from the previously occluded regions. The back-projected motion vectors of these pixels will always land on the occluders, thus the previous pixel values cannot easily be used.

To alleviate this issue, we start from the clamping technique by Salvi [8] in TAA methods. Our insight is that if the color value at X^0_{i-1} is closer to the value at X_i, the issues produced by the clamping method can be better resolved. Also, close color values often appear on the same object. Inspired by Bowles et al. [1], we propose a new motion vector for the just-appeared region to find a similar correspondence in the previous frame. To do that, we refer to the relative motion.

As Figure 25-5a shows, the traditional motion vector gives the $X_i \rightarrow Y$ correspondence, but, unfortunately, cannot be easily used. Our method continues to track the movement of $Y \rightarrow Z$ from the previous frame to the current, using the motion of the occluder. Then, based on the relative positions of X_i and Z, we are able to find the location X^0_{i-1} in the previous

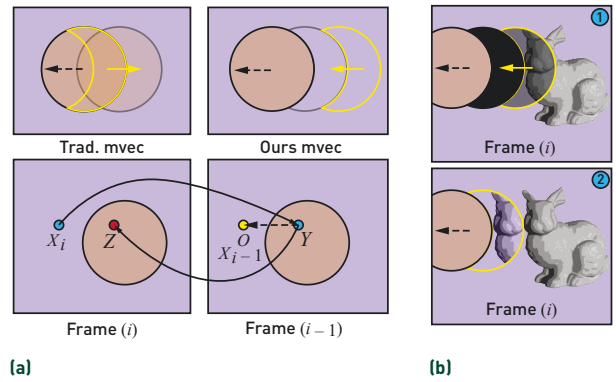

Figure 25-5. *(a) The computation of our dual motion vectors for occlusions, and comparison with traditional motion vectors. (b) How the repetitive pattern is produced when simply reusing colors with our dual motion vectors.*

frame. This process can be simply represented as

$$X^{O}_{i-1} = Y + (X_i - Z), \tag{25.6}$$

where $Y = \mathbf{P}_{i-1}\mathbf{T}^{-1}(X_i)\mathbf{P}_i^{-1}X_i$ and $Z = \mathbf{P}_i\mathbf{T}(Y)\mathbf{P}_{i-1}^{-1}Y$.

Equation 25.6 indicates that we have applied a back-projection followed by a forward-projection, essentially using two motion vectors. Thus, we name our approach *dual motion vectors* for occlusions. In this way, we are able to find a correspondence X^{O}_{i-1} with a much closer color value to X_i. Note that because we use $\mathbf{P}$ of two frames to track X_i, we are able to support the movement of the camera and objects simultaneously. The pseudocode for calculating our occlusion motion vector looks as follows:

```
calcOcclusionMotionVector(uint2 pixelIndex, float4 hitPos, uint hitMeshID,
     uint occluderMeshID){
    // For each pixel X_i that previously occluded ...
    float2 X = pixelIndex + float2(0.5f, 0.5f);
    // ... find Y on the occluder in the previous frame.
    float4 S = hitPos;
    // The custom function getT(...) returns the geometry transformation for
         a given mesh, and inverse(...) inverts a 4x4 matrix.
    float4 prevS = mul(S, inverse(getT(hitMeshID)));
    // The custom function toPrevScreen(...) projects a given world-space
        point to screen space to get the corresponding image-space pixel
        for the previous frame.
    float2 Y = toPrevScreen(prevS);
    // Then, find Z on occluder in the current frame.
    // The custom function getPrevHitPos(...) returns the world-space
        shading point for a given pixel in the previous frame.
    float4 SY = getPrevHitPos(Y);
    float4 curSY = mul(SY, getT(occluderMeshID));
    // The custom function toScreen(...) projects a given world-space point
        to screen space to get the corresponding image-space pixel.
    float2 Z = toScreen(curSY);
    // Return the occlusion motion vector.
    return Y + X - Z;
}
```

Note that because we deal with different effects separately for shadows and glossy reflection, and the shading part can already be reasonably approximated in a noise-free way (e.g., using the Linear Transformed Cosines (LTC) method [5]), we only have to apply our dual motion vectors to indirect illumination. Moreover, as glossy indirect illumination has been elegantly addressed using our glossy reflection motion vectors, we can now focus only on diffuse materials.

DISCUSSION: REUSING COLOR VERSUS INCIDENT RADIANCE As Figure 25-5b indicates, simply applying the color values as in Bowles et al. [1], using the dual motion vector will result in a clear repetitive pattern because it

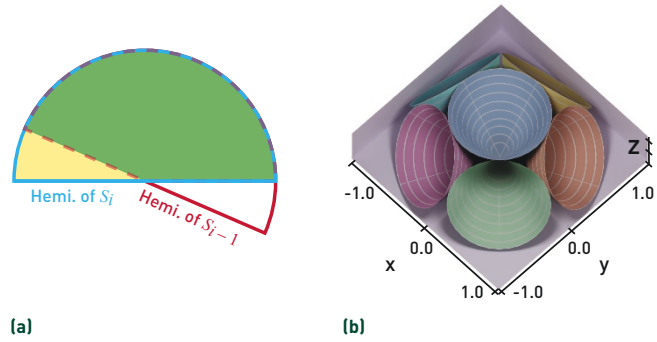

Figure 25-6. *(a) Our partial temporal reuse scheme. Only the same directions in the overlapped solid angle will share temporally coherent radiance. (b) We use six slightly overlapping cones on a hemisphere to store incident radiance approximately.*

is essentially copying and pasting image contents. This is especially problematic when the normals at S_i and S^0_{i-1} are different. Removing the textures from colors (i.e., *demodulation*) does not help because when the normals differ, the intensity of the shading result can differ significantly.

To address this issue, we propose to temporally reuse the incident radiance instead of the shading result. Specifically, for the application of diffuse indirect illumination, we record the 2D indirect incident radiance per pixel.

Figure 25-6a shows an example. Suppose that we have recorded the incident radiance of S_i and S^0_{i-1}, each on a hemisphere; then, we immediately know that all the directions in the overlapped regions of these two hemispheres (marked as green) could be temporally reused, while the non-overlapping part (marked as yellow) should remain using only the spatial content from frame i. Then, the radiance from both parts will be used to re-illuminate the shading point X_i, leading to an accurate temporally accumulated shading result.

For storing and blending the incident radiance, we refer to the representation in the voxel cone tracing approach [3]. We subdivide a hemisphere into six slightly overlapping cones with equal solid angles $\pi/3$ pointing in different directions, and we assume that the radiance remains constant within each cone. During spatial or temporal filtering, instead of averaging the shading result, we filter for each cone individually, using the overlapping solid angles between each pair of cones as the filtering weight. Figure 25-6b illustrates our idea. Finally, we are able to achieve a much cleaner result for diffuse indirect illumination.

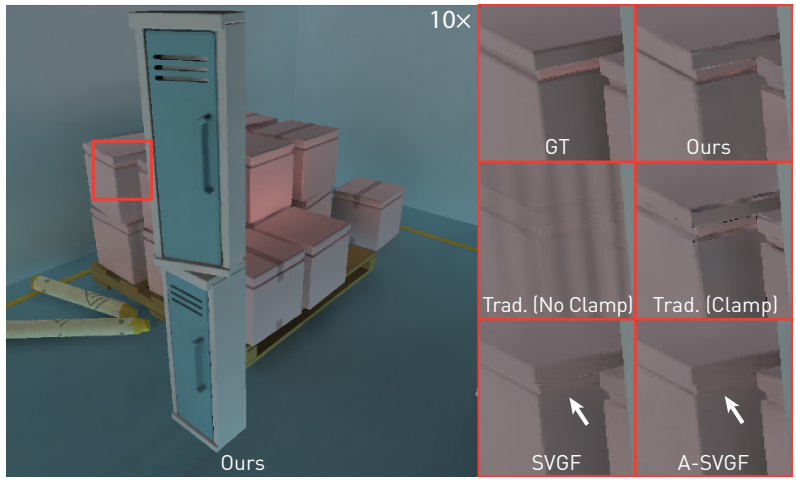

Figure 25-7. *A view of the PICA scene with objects moving from left to right.*

We demonstrate the effectiveness of our occlusion motion vectors in the *PICA* scene with moving objects in Figure 25-7. Only indirect illumination is shown, and its intensity is scaled ten times for better visibility. New unoccluded regions will appear around the boundaries of foreground objects. In these regions, the temporal information will simply be rejected with traditional motion vectors. Therefore, in this case the SVGF still results in a significant amount of overblur, whereas A-SVGF again appears to be smeared spatially and loses temporal stability. The clamping approach tries to use pixel values from the occluders; however, because the pixel values on the foreground and background usually differ drastically in the occlusion case, this will still introduce ghosting artifacts.

25.4 PERFORMANCE

Figure 25-8 shows the average computation cost of each step of denoising individual effects using our motion vectors. The average cost of an individual step was estimated from 500 frames rendered at 1920×1080 on an NVIDIA TITAN RTX. Compared with SVGF (traditional motion vectors), it can denoise different effects with a similar cost, around 3.11 ms per frame.

As one would expect from the simple computation in Section 25.3, in practice we observed only a negligible performance cost by replacing with our motion vectors, which is always less than 0.23 ms. Besides, we have also noticed that our implementation of denoising shadows and glossy reflections is already

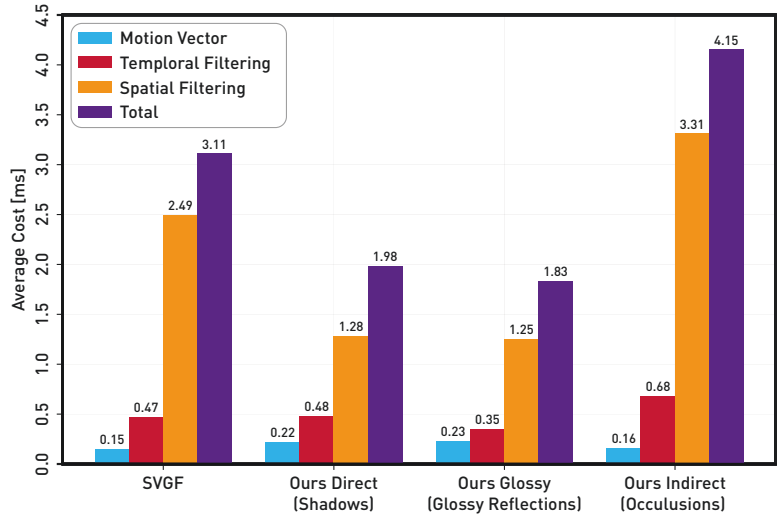

Figure 25-8. *Runtime breakdown of our methods.*

much faster than SVGF, as we use much simpler spatial filters and fewer levels or passes. Finally, when denoising indirect illumination, it is worth mentioning the additional cost when we introduce the cones for storing and filtering the incident radiance. For this, compared with reusing colors, we found that the typical cost of denoising each frame increased around 1.5 ms.

25.5 CONCLUSION

In this chapter, we have proposed multiple types of motion vectors for better utilization of temporal information in real-time ray tracing. With our motion vectors, we are able to track the movement of shadows and glossy reflections and to find similar regions to blend with previously occluded regions. We showed that our motion vectors are temporally more reliable than traditional motion vectors and presented cleaner results compared to the state-of-the-art methods with negligible performance overhead.

For more information about implementation details, comparisons, and limitations, we refer readers to our paper [11] and the accompanying video.

REFERENCES

[1] Bowles, H., Mitchell, K., Sumner, R. W., Moore, J., and Gross, M. Iterative image warping. *Computer Graphics Forum*, 31(2pt1):237–246, 2012. DOI: 10.1111/j.1467-8659.2012.03002.x.

[2] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics*, 36(4):98:1–98:12, 2017. DOI: 10.1145/3072959.3073601.

[3] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. Interactive indirect illumination using voxel cone tracing. *Computer Graphics Forum*, 30(7):1921–1930, 2011. DOI: 10.1111/j.1467-8659.2011.02063.x.

[4] Fernando, R. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*, page 35, 2005. DOI: 10.1145/1187112.1187153.

[5] Heitz, E., Dupuy, J., Hill, S., and Neubelt, D. Real-time polygonal-light shading with linearly transformed cosines. *ACM Transactions on Graphics*, 35(4):411:–41:8, 2016. DOI: 10.1145/2897824.2925895.

[6] Liu, E., Llamas, I., Kelly, P., et al. Cinematic rendering in UE4 with real-time ray tracing and denoising. In E. Haines and T. Akenine-Möller, editors, *Ray Tracing Gems*, pages 289–319. Apress, 2019.

[7] Mara, M., McGuire, M., Bitterli, B., and Jarosz, W. An efficient denoising algorithm for global illumination. In *High Performance Graphics*, pages 3–1, 2017.

[8] Salvi, M. Anti-aliasing: Are we there yet? Open Problems in Real-Time Rendering, SIGGRAPH Course, 2015.

[9] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, page 2, 2017.

[10] Schied, C., Peters, C., and Dachsbacher, C. Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(2):24:1–24:16, 2018. DOI: 10.1145/3233301.

[11] Zheng, Z., Shiqiu, L., Jinglei, Y., Lu, W., and Ling-Qi, Y. Temporally reliable motion vectors for real-time ray tracing. *Computer Graphics Forum (Proceedings of Eurographics 2021)*, 40(2):79–90, 2021. DOI: 10.1111/cgf.142616.

[12] Zimmer, H., Rousselle, F., Jakob, W., Wang, O., Adler, D., Jarosz, W., Sorkine-Hornung, O., and Sorkine-Hornung, A. Path-space motion estimation and decomposition for robust animation filtering. *Computer Graphics Forum*, 34(4):131–142, 2015. DOI: 10.1111/cgf.12685.