7

A Spatial and Temporal Coherence Framework for Real-Time Graphics

Michal Drobot
Reality Pump Game Development Studios

With recent advancements in real-time graphics, we have seen a vast improvement in pixel rendering quality and frame buffer resolution. However, those complex shading operations are becoming a bottleneck for current-generation consoles in terms of processing power and bandwidth. We would like to build upon the observation that under certain circumstances, shading results are temporally or spatially coherent. By utilizing that information, we can reuse pixels in time and space, which effectively leads to performance gains.

This chapter presents a simple, yet powerful, framework for spatiotemporal acceleration of visual data computation. We exploit spatial coherence for geometry-aware upsampling and filtering. Moreover, our framework combines motionaware filtering over time for higher accuracy and smoothing, where required. Both steps are adjusted dynamically, leading to a robust solution that deals sufficiently with high-frequency changes. Higher performance is achieved due to smaller sample counts per frame, and usage of temporal filtering allows convergence to maximum quality for near-static pixels.

Our method has been fully production proven and implemented in a multiplatform engine, allowing us to achieve production quality in many rendering effects that were thought to be impractical for consoles. An example comparison of screen-space ambient occlusion (SSAO) implementations is shown in Figure 7.1. Moreover, a case study is presented, giving insight to the framework usage and performance with some complex rendering stages like screen-space ambient occlusion, shadowing, etc. Furthermore, problems of functionality, performance, and aesthetics are discussed, considering the limited memory and computational power of current-generation consoles.

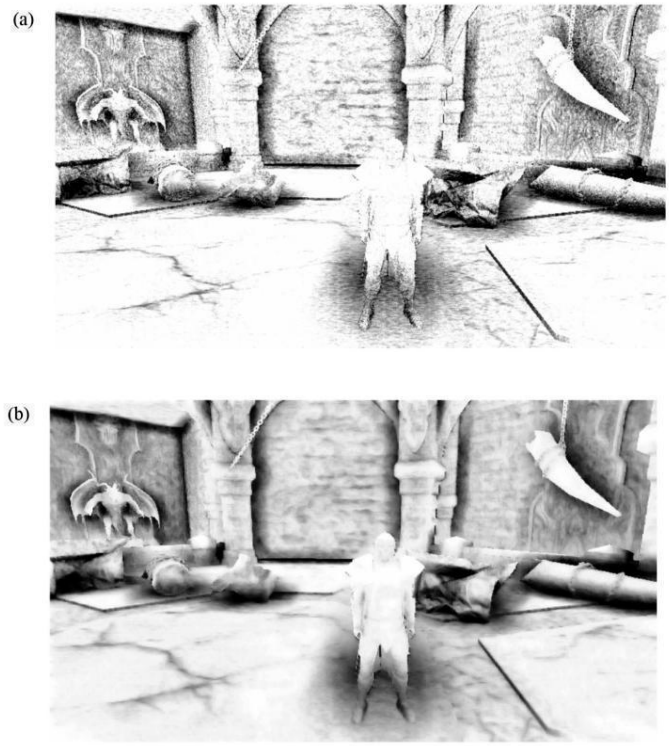


Figure 7.1. (a) A conventional four-tap SSAO pass. (b) A four-tap SSAO pass using the spatiotemporal framework.

7.1 Introduction

The most recent generation of game consoles has brought some dramatic improvements in graphics rendering quality. Several new techniques were introduced, like deferred lighting, penumbral soft shadows, screen-space ambient occlusion, and even global illumination approximations. Renderers have touched the limit of current-generation home console processing power and bandwidth. However, expectations are still rising. Therefore, we should focus more on the overlooked subject of computation and bandwidth compression.

Most pixel-intensive computations, such as shadows, motion blur, depth of field, and global illumination, exhibit high spatial and temporal coherency. With ever-increasing resolution requirements, it becomes attractive to utilize those similarities between pixels

[Nehab et al. 2007]. This concept is not new, as it is the basis for motion picture compression.

If we take a direct stream from our rendering engine and compress it to a level perceptually comparable with the original, we can achieve a compression ratio of at least 10:1. What that means is that our rendering engine is calculating huge amounts of perceptually redundant data. We would like to build upon that.

Video compressors work in two stages. First, the previous frames are analyzed, resulting in a motion vector field that is spatially compressed. The previous frame is morphed into the next one using the motion vectors. Differences between the generated frame and the actual one are computed and encoded again with compression. Because differences are generally small and movement is highly stable in time, compression ratios tend to be high. Only keyframes (i.e., the first frame after a camera cut) require full information.

We can use the same concept in computer-generated graphics. It seems attractive since we don't need the analysis stage, and the motion vector field is easily available. However, computation dependent on the final shaded pixels is not feasible for current rasterization hardware. Current pixel-processing pipelines work on a per-triangle basis, which makes it difficult to compute per-pixel differences or even decide whether the pixel values have changed during the last frame (as opposed to ray tracing, where this approach is extensively used because of the per-pixel nature of the rendering). We would like to state the problem in a different way.

Most rendering stages' performance to quality ratio are controlled by the number of samples used per shaded pixel. Ideally, we would like to reuse as much data as possible from neighboring pixels in time and space to reduce the sampling rate required for an optimal solution. Knowing the general behavior of a stage, we can easily adopt the compression concept. Using a motion vector field, we can fetch samples over time, and due to the low-frequency behavior, we can utilize spatial coherency for geometry-aware upsampling. However, there are several pitfalls to this approach due to the interactive nature of most applications, particularly video games.

This chapter presents a robust framework that takes advantage of spatiotemporal coherency in visual data, and it describes ways to overcome the associated problems. During our research, we sought the best possible solution that met our demands of being robust, functional, and fast since we were aiming for Xbox 360- and PlayStation 3-class hardware. Our scenario involved rendering large outdoor scenes with cascaded shadow maps and screen-space ambient occlusion for additional lighting detail. Moreover, we extensively used advanced

material shaders combined with multisampling as well as a complex postprocessing system. Several applications of the framework were developed for various rendering stages. The discussion of our final implementation covers several variations, performance gains, and future ideas.

7.2 The Spatiotemporal Framework

Our spatiotemporal framework is built from two basic algorithms, bilateral upsampling and real-time reprojection caching. (Bilateral filtering is another useful processing stage that we discuss.) Together, depending on parameters and application specifics, they provide high-quality optimizations for many complex rendering stages, with a particular focus on low-frequency data computation.

Bilateral Upsampling

We can assume that many complex shader operations are low-frequency in nature. Visual data like ambient occlusion, global illumination, and soft shadows tend to be slowly varying and, therefore, well behaved under upsampling operations. Normally, we use bilinear upsampling, which averages the four nearest samples to a point being shaded. Samples are weighted by a spatial distance function. This type of filtering is implemented in hardware, is extremely efficient, and yields good quality. However, a bilinear filter does not respect depth discontinuities, and this creates leaks near geometry edges. Those artifacts tend to be disturbing due to the high-frequency changes near object silhouettes. The solution is to steer the weights by a function of geometric similarity obtained from a highresolution geometry buffer and coarse samples [Kopf et al. 2007]. During interpolation, we would like to choose certain samples that have a similar surface orientation and/or a small difference in depth, effectively preserving geometry edges. To summarize, we weight each coarse sample by bilinear, normalsimilarity, and depth-similarity weights.

Sometimes, we can simplify bilateral upsampling to account for only depth discontinuities when normal data for coarse samples is not available. This solu tion is less accurate, but it gives plausible results in most situations when dealing with low-frequency data.

Listing 7.1 shows pseudocode for bilateral upsampling. Bilateral weights are precomputed for a 2×2 coarse-resolution tile, as shown in Figure 7.2. Depending on the position of the pixel being shaded (shown in red in Figure 7.2), the correct weights for coarse-resolution samples are

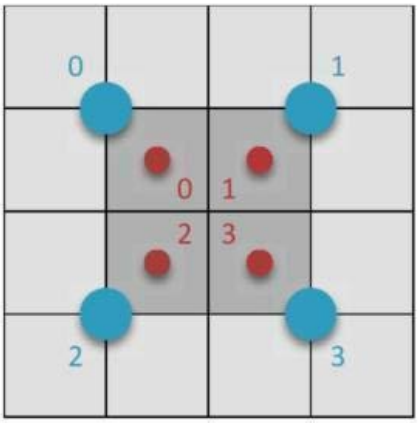
chosen from the table.

Listing 7.1. Pseudocode for bilateral upsampling.

```
for (int i=0; i < 4; i++)
{
  normalWeights[i] = dot(normalsLow[i] +normalHi);
  normalWeights[i] = pow(VNormalWeights[i], contrastCoef);
}

for (int i=0; i < 4; i++)
{
  float depthDiff = depthHi - depthLow[i];
  depthWeights[i] = 1.0 / (0.0001f + abs(depthDiff));
}

for (int i=0; i < 4; i++)
{
  float sampleWeight = normalWeights[i] * depthWeights[i] *
    bilinearWeights[texelNo][i];
  totalWeight += sampleWeight;
  upsampledResult += sampleLow[i] * fWeight;
}

upsampledResult /= totalWeight;
```

Reprojection Caching

Another optimization concept is to reuse data over time [Nehab et al. 2007]. During each frame, we would like to sample previous frames for additional data, and thus, we need a history buffer, or cache, that stores the data from previous frames. With each new pixel being shaded in the current frame, we check whether additional data is available in the history buffer and how relevant it is. Then, we decide if we want to use that data, reject it, or perform some additional computation. Figure 7.3 presents a schematic overview of the method.



Low-resolution sample

High-resolution sample

	0	1	2	3
0	9/16	3/16	3/16	1/16
1	3/16	9/16	1/16	3/16
2	3/16	1/16	9/16	3/16
3	1/16	3/16	3/16	9/16

Figure 7.2. Bilinear filtering with a weight table.

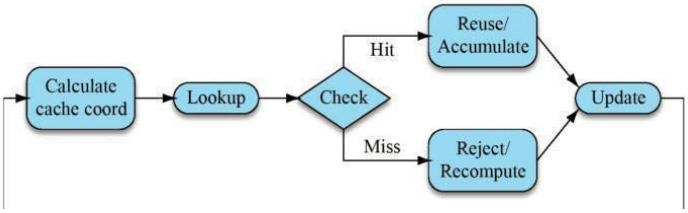


Figure 7.3. Schematic diagram of a simple reprojection cache.

For each pixel being shaded in the current frame, we need to find a corresponding pixel in the history buffer. In order to find correct cache coordinates, we need to obtain the pixel displacement between frames. We know that pixel movement is a result of object and camera transformations, so the solution is to reproject the current pixel coordinates using a motion vector. Coordinates must be treated with special care. Results must be accurate, or we will have flawed history values due to repeated invalid cache resampling. Therefore, we perform computation in full precision, and we consider any possible offsets involved when working with render targets, such as the half-pixel offset in DirectX 9.

Coordinates of static objects can change only due to camera movement, and the calculation of pixel motion vectors is therefore straightforward. We find the pixel position in camera space and project it back to screen space using the previous frame's projection matrix. This calculation is fast and can be done in one fullscreen pass, but it does not handle moving geometry.

Dynamic objects require calculation of per-pixel velocities by comparing the current camera-space position to the last one on a per-object basis, taking skinning and transformations into consideration. In engines calculating a frame-toframe motion field (i.e., for per-object motion blur [Lengyel 2010]), we can reuse the data if it is pixel-correct. However, when that information is not available, or the performance cost of an additional geometry pass for velocity calculation is too high, we can resort to camera reprojection. This, of course, leads to false cache coordinates for an object in motion, but depending on the application, that might be acceptable. Moreover, there are several situations where pixels may not have a history. Therefore, we need a mechanism to check for those situations.

Cache misses occur when there is no history for the pixel under consideration. First, the obvious reason for this is the case when we are sampling outside the history buffer. That often occurs near screen edges due to camera movement or objects moving into the view frustum. We simply check whether the cache coordinates are out of bounds and count it as a miss.

The second reason for a cache miss is the case when a point **p** visible at time t was occluded at time t - l by another point **q**. We can assume that it is impossible for the depths of **q** and **p** to match at time t – l. We know the expected depth of **p** at time t – l through reprojection, and we compare that with the cached depth at **q**. If the difference is bigger than an epsilon, then we consider it a cache miss. The depth at **q** is reconstructed using bilinear filtering, when possible, to account for possible false hits at depth discontinuities, as illustrated in Figure 7.4.

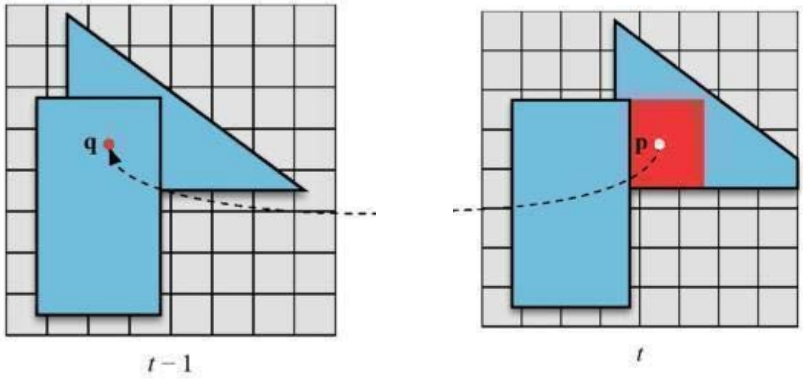


Figure 7.4. Possible cache-miss situation. The red area lacks history data due to occlusion in the previous frame. Simple depth comparison between projected **p** and **q** from t – l is sufficient to confirm a miss.

If there is no cache miss, then we sample the history buffer. In general, pixels do not map

to individual cache samples, so some form of resampling is needed. Since the history buffer is coherent, we can generally treat it as a normal texture buffer and leverage hardware support for linear and point filtering, where the proper method should be chosen for the type of data being cached. Lowfrequency data can be sampled using the nearest-neighbor method without significant loss of detail. On the other hand, using point filtering may lead to discontinuities in the cache samples, as shown in Figure 7.5. Linear filtering correctly handles these artifacts, but repeated bilinear resampling over time leads to data smoothing. With each iteration, the pixel neighborhood influencing the result grows, and high-frequency details may be lost. Last but not least, a solution that guarantees high quality is based on a higher-resolution history buffer and nearestneighbor sampling at a subpixel level. Nevertheless, we cannot use it on consoles because of the additional memory requirements.

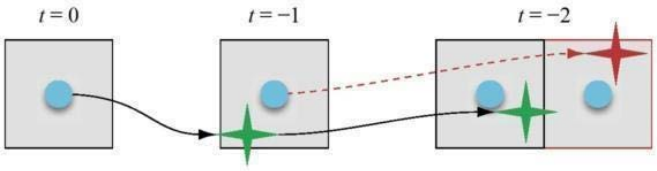


Figure 7.5. Resampling artifacts arising in point filtering. A discontinuity occurs when the correct motion flow (yellow star) does not match the approximating nearest-neighbor pixel (red star).

Motion, change in surface parameters, and repeated resampling inevitably degrade the quality of the cached data, so we need a way to refresh it. We would like to efficiently minimize the shading error by setting the refresh rate proportional to the time difference between frames, and the update scheme should be dependent on cached data.

If our data requires explicit refreshes, then we have to find a way to guarantee a full cache update every n frames. That can easily be done by updating one of n parts of the buffer every frame in sequence. A simple tile-based approach or, littering could be used, but without additional processing like bilateral upsampling or filtering, pattern artifacts may occur.

The reprojection cache seems to be more effective with accumulation functions. In computer graphics, many results are based on stochastic processes that combine randomly distributed function samples, and the quality is often based on the number of computed samples. With reprojection caching, we can easily amortize that complex process over time, gaining in performance and accuracy. This method is best suited for rendering stages based on multisampling and lowfrequency or slowly varying data.

When processing frames, we accumulate valid samples over time. This leads to an exponential history buffer that contains data that has been integrated over several frames back in time. With each new set of accumulated values, the buffer is automatically updated, and we control the convergence through a dynamic factor and exponential function. Variance is related to the number of frames contributing to the current result. Controlling that number lets us decide whether response time or quality is preferred.

We update the exponential history buffer using the equation

$$h(t) = h(t-1)w + r(1-w)\,,$$

where $h(t)$ represents a value in the history buffer at time t, w is the convergence weight, and r is a newly computed value. We would like to steer the convergence weight according to changes in the scene, and this could be done based on the per-pixel velocity. This would provide a valid solution for temporal ghosting artifacts and high-quality convergence for near-static elements.

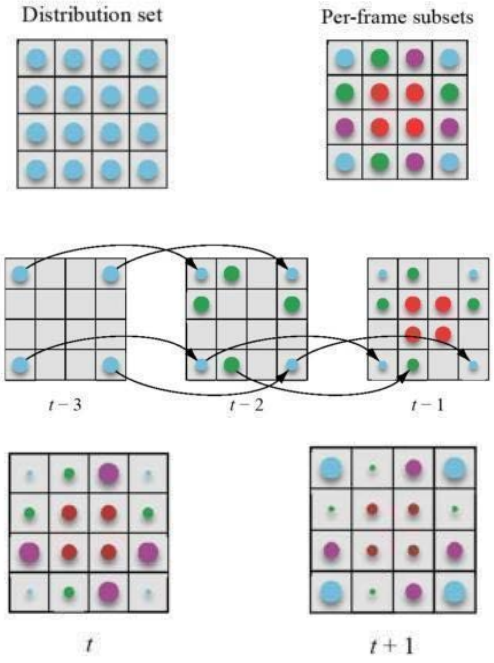


Figure 7.6. We prepare the expected sample distribution set. The sample set is divided into *N* subsets (four in this case), one for each consecutive frame. With each new frame, missing subsets are

sampled from previous frames, by exponential history buffer look-up. Older samples lose weight with each iteration, so the sampling pattern must repeat after N frames to guarantee continuity.

Special care must be taken when new samples are calculated. We do not want to have stale data in the cache, so each new frame must bring additional information to the buffer. When rendering with multisampling, we want to use a different set of samples with each new frame. However, sample sets should repeat after N frames, where N is the number of frames being cached (see Figure 7.6). This improves temporal stability. With all these improvements, we obtain a highly efficient reprojection caching scheme. Listing 7.2 presents a simplified solution used during our SSAO step.

Listing 7.2. Simplified reprojection cache.

```
float4 ReproCache(vertexOutput IN)
{
  float4 ActiveFrame = tex2D(gSamplerOPT, IN.Uv.xy);
  float freshData = ActiveFrame.x;
  float3 WSpacePos = WorldSpaceFast(IN.WorldEye, ActiveFrame.w);
  float4 LastClipSpacePos = mul(float4(WorldSpacePos.xyz, 1.0),
    IN_CameraMatrixPrev);
  float lastDepth = LastClipSpacePos.z;
  LastClipSpacePos = mul(float4(LastClipSpacePos.xyz, 1.0),
    IN_ProjectionMatrixPrev);
  LastClipSpacePos /= LastClipSpacePos.w;
  LastClipSpacePos.xy *= float2(0.5, -0.5);
  LastClipSpacePos.xy *= float2(0.5, 0.5);

  float4 reproCache = tex2D(gSamplerPT, LastClipSpacePos.xy);

  float reproData = reproCache.x;
  float missRate = abs(lastDepth - reproCache.w) - IN_MissThreshold;

  if (LastClipSpacePos.x <0.0 || LastClipSpacePos.x >1.0 ||
    LastClipSpacePos.y < 0.0 || LastClipSpacePos.y > 1.0)
     missRate = 0.0;
```

```
  missRate = saturate(missRate * IN_ConvergenceTime);
  freshData += (reproData - freshData.xy) * missRate;

  float4 out = freshData;
  out.a = ActiveFrame.w;
  return out;
}
```

Bilateral Filtering

Bilateral filtering is conceptually similar to bilateral upsampling. We perform Gaussian filtering with weights influenced by a geometric similarity function [Tomasi and Manduchi 1998]. We can treat it as an edge-aware smoothing filter or a high-order reconstruction filter utilizing spatial coherence. Bilateral filtering proves to be extremely efficient for content-aware data smoothing. Moreover, with only insignificant artifacts, a bilateral filter can be separated into two directions, leading to $O(n)$ running time. We use it for any kind of slowly-varying data, such as ambient occlusion or shadows, which needs to be aware of scene geometry. Moreover, we use it to compensate for undersampled pixels. When a pixel lacks samples, lacks history data, or has missed the cache, it is reconstructed from spatial coherency data. That solution leads to more plausible results compared to relying on temporal data only. Listing 7.3 shows a separable, depthaware bilateral filter that uses hardware linear filtering.

Listing 7.3. Directional bilateral filter working with depth data.

```
float Bilateral3D5x5(sampler2D inSampler, float2 texelSize,
                 float2 UV, float2 Dir)
{
  const float centerWeight = 0.402619947;
  const float4 tapOffsets = float4(-3.5, -1.5, 1.5, 3.5);
  const float4 tapWeights = float4(0.054488685, 0.244201342,
    0.244201342, 0.054488685);

  const float E = 1.0;
  const float diffAmp = IN_BilateralFilterAmp;
```

```
  float2 color;
  float4 pSamples, nSamples;
  float4 diffIp, diffIn;
  float4 pTaps[2];

  float2 offSize = Dir * texelSize;

  pTaps[0] = UV.xyxy + tapOffsets.xxyy * offSize.xyxy;
  pTaps[1] = UV.xyxy + tapOffsets.zzww * offSize.xyxy;

  color = tex2D(inSampler, UV.xy).ra;

  //r - contains data to be filtered
  //a - geometry depth
  pTaps[0].xy = tex2D(inSampler, pTaps[0].xy).ra;
  pTaps[0].zw = tex2D(inSampler, pTaps[0].zw).ra;
  pTaps[1].xy = tex2D(inSampler, pTaps[1].xy).ra;
  pTaps[1].zw = tex2D(inSampler, pTaps[1].zw).ra;

  float4 centerlD = color.y;

  diffIp = (1.0 /(E + diffAmp * abs(centralD - float4(pTaps[0].y,
     pTaps[0].w, pTaps[1].y, pTaps[1].w)))) * tapWeights;
  float Wp = 1.0 / (dot(diffIp, 1) + centerWeight);

  color.r *= centerWeight;
  color.r = Wp * (dot(diffIp, float4(pTaps[0].x, pTaps[0].z,
    pTaps[1].x,pTaps[1].z)) + color.r);

  return color.r;
}
```

Spatiotemporal Coherency

We would like to combine the described techniques to take advantage of the spatiotemporal coherency in the data. Our default framework works in several steps:

1. Depending on the data, caching is performed at lower resolution.
2. We operate with the history buffer (HB) and the current buffer (CB).
3. The CB is computed with a small set of current samples.
4. Samples from the HB are accumulated in the CB by means of reprojection caching.
5. A per-pixel convergence factor is saved for further processing.
6. The CB is bilaterally filtered with a higher smoothing rate for pixels with a lower convergence rate to compensate for smaller numbers of samples or cache misses.
7. The CB is bilaterally upsampled to the original resolution for further use.
8. The CB is swapped with the HB.

The buffer format and processing steps differ among specific applications.

7.3 Applications

Our engine is composed of several complex pixel-processing stages that include screen-space ambient occlusion, screen-space soft shadows, subsurface scattering for skin shading, volumetric effects, and a post-processing pipeline with depth of field and motion blur. We use the spatiotemporal framework to accelerate most of those stages in order to get the engine running at production-quality speeds on current-generation consoles.

Screen-Space Ambient Occlusion

Ambient occlusion *AO* is computed by integrating the visibility function over a hemisphere *H* with respect to a projected solid angle, as follows:

$$AO = \frac{1}{\pi}\int_{H} V_{\mathrm{p},\omega}(\mathbf{N}.\omega)\, d\omega\,,$$

where **N** is the surface normal and $V_{p,\omega}$, is the visibility function at p (such that $V_{p,\omega} = 0$ when occluded in the direction **ω**, and $V_{p,\omega}=1$ otherwise). It can be efficiently computed in screen space by multiple occlusion checks that sample the depth buffer around the point being shaded. However, it is extremely taxing on the GPU due to the high sample count and large kernels that trash the texture cache. On current-generation consoles, it seems impractical to use more than eight samples. In our case, we could not even afford that many because, at the time, we had only two milliseconds left in our frame time budget.

After applying the spatiotemporal framework, we could get away with only four samples per frame, and we achieved even higher quality than before due to amortization over time. We computed the SSAO at half resolution and used bilateral upsampling during the final frame combination pass. For each frame, we changed the SSAO kernel sampling pattern, and we took care to generate a uniformly distributed pattern in order to minimize frame-to-frame inconsistencies. Due to memory constraints on consoles, we decided to rely only on depth information, leaving the surface normal vectors available only for SSAO computation. Furthermore, since we used only camera-based motion blur, we lacked per-pixel motion vectors, so an additional pass for motion field computation was out of the question. During caching, we resorted to camera reprojection only. Our cachemiss detection algorithm compensated for that by calculating a running convergence based on the distance between a history sample and the predicted valid position. That policy tended to give good results, especially considering the additional processing steps involved. After reprojection, ambient occlusion data was bilaterally filtered, taking convergence into consideration when available (PC only). Pixels with high temporal confidence retained high-frequency details, while others were reconstructed spatially depending on the convergence factor. It is worth noticing that we were switching history buffers after bilateral filtering. Therefore, we were filtering over time, which enables us to use small kernels without significant quality loss. The complete solution required only one millisecond of GPU time and enabled us to use SSAO in real time on the Xbox 360. Figure 7.7 shows final results compared to the default algorithm.

Figure 7.7. The left column shows SSAO without using spatial coherency. The right column shows our final Xbox 360 implementation.

Soft Shadows

Our shadowing solution works in a deferred manner. We use the spatiotemporal framework for sun shadows only since those are computationally expensive and visible all the time. First, we draw sun shadows to an offscreen low-resolution buffer. While shadow testing against a cascaded shadow map, we use a custom percentage closer filter. For each frame, we use a different sample from a welldistributed sample set in order to leverage temporal coherence [Scherzer et al. 2007]. Reprojection caching accumulates the samples over time in a manner similar to our SSAO solution. Then the shadow buffer is bilaterally filtered in screen space and bilaterally upsampled for the final composition pass. Figures 7.8 and 7.9 show our final results for the Xbox 360 implementation.

Figure 7.8. Leveraging the spatiotemporal coherency of shadows (bottom) enables a soft, filtered, look free of undersampling artifacts, without raising the shadow map resolution of the original scene (top).

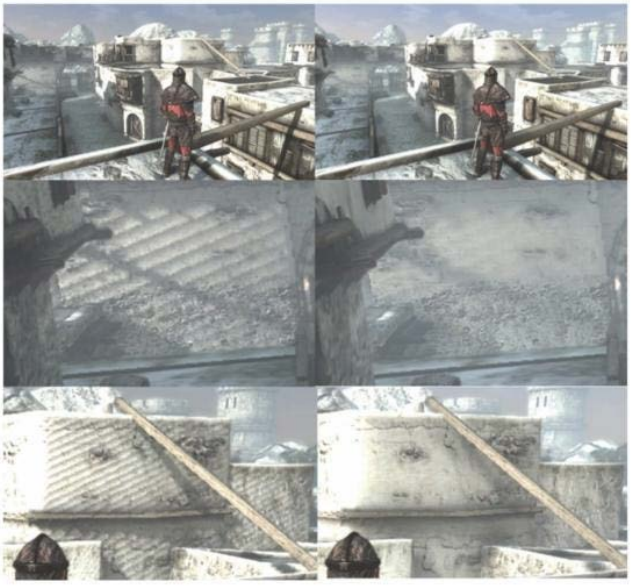


Figure 7.9. The spatiotemporal framework efficiently handles shadow acne and other flickering artifacts (right) that appear in the original scene (left).

Shadows and Ambient Occlusion Combined

Since our shadowing pipeline is similar to the one used during screen-space ambient occlusion, we integrate both into one pass in our most efficient implementation running on consoles. Our history buffer is half the resolution of the back buffer, and it is stored in RG16F format. The green channel stores the minimum depth of the four underlying pixels in the Z-buffer. The red channel contains shadowing and ambient occlusion information. The fractional part of the 16-bit floating-point value is used for occlusion because it requires more variety, and the integer part holds the shadowing factor. Functions for packing and unpacking these values are shown in Listing 7.4.

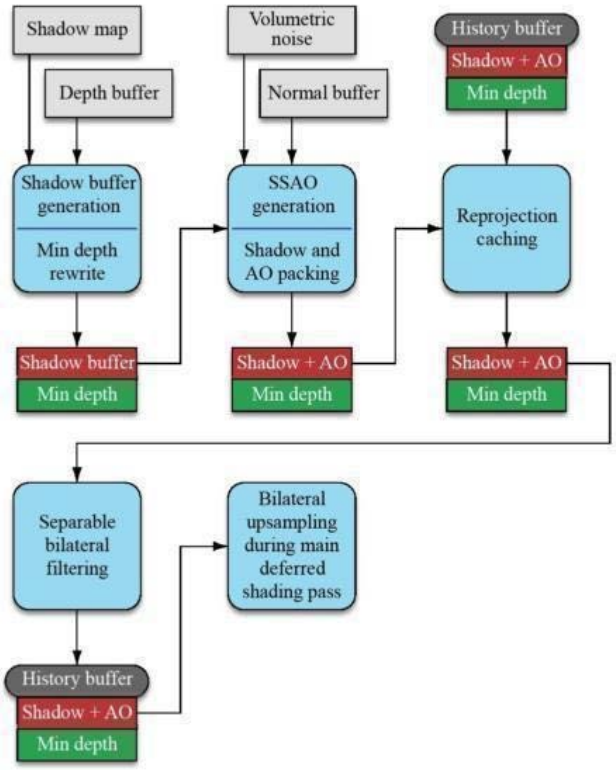


Figure 7.10. Schematic diagram of our spatiotemporal framework used with SSAO and shadows.

Every step of the spatiotemporal framework runs in parallel on both the shadowing and ambient occlusion values using the packing and unpacking functions. The last step of the framework is bilateral upsampling combined with the main deferred shading pass. Figure 7.10 shows an overview of the pipeline. The performance gained by using our technique on the Xbox

360 is shown in Table 7.1.

Listing 7.4. Code for shadow and occlusion data packing and unpacking.

```
#define  PACK_RANGE   31.0
#define  MIN_FLT     0.01

float PackOccShadow(float Occ, float Shadow)
{
  return (floor(saturate(Occ) * PACK_RANGE) +
    clamp(Shadow, MIN_FLT, 1.0 - MIN_FLT));
}

float2 UnpackOccShadow(float OccShadow)
{
  return (float2((floor(OccShadow)) / PACK_RANGE, frac(OccShadow)));
}
```

Table 7.1. Performance comparison of various stages and a reference solution in which shadowing is performed in full resolution with 2 × 2 jittered PCF, and SSAO uses 12 taps and upsampling. The spatiotemporal (ST) framework is 2.5 times faster than the reference solution and still yields better image quality.

Stage	ST Framework	Reference
Shadows	0.7 ms	3.9 ms
SSAO generation	1.1 ms	3.8 ms
Reprojection caching	0.35 ms	-
Bilateral filtering	0.42 ms (0.2 ms per pass)	-
Bilateral upsampling	0.7 ms	0.7 ms
Total	3.27 ms	8.4 ms

Postprocessing

Several postprocessing effects, such as depth of field and motion blur, tend to have high spatial and temporal coherency. Both can be expressed as a multisampling problem in time and space and are, therefore, perfectly suited for our framework. Moreover, the mixed frequency nature of both effects tends to hide any possible artifacts. During our tests, we were able to

perform productionready postprocessing twice as fast as with a normal non-cached approach.

Additionally, blurring is an excellent candidate for use with the spatiotemporal framework. Normally, when dealing with extremely large blur kernels, hierarchical downsampling with filtering must be used in order to reach reasonable performance with enough stability in high-frequency detail. Using importance sampling for downsampling and blurring with the spatiotemporal framework, we are able to perform high-quality Gaussian blur, using radii reaching 128 pixels in a 720p frame, with no significant performance penalty (less than 0.2 ms on the Xbox 360). The final quality is shown in Figure 7.11.

First, we sample nine points with linear filtering and importance sampling in a single downscaling pass to 1/64 of the screen size. Stability is sustained by the reprojection caching, with different subsets of samples used during each frame. The resulting image is blurred, cached, and upsampled. Bilateral filtering is used when needed by the application (e.g., for depth-of-field simulation where geometry awareness is required).

Figure 7.11. The bottom image shows the result of applying a large-kernel (128-pixel) Gaussian blur used for volumetric water effects to the scene shown in the top image. This process is efficient and stable on the Xbox 360 using the spatiotemporal coherency framework.

7.4 Future Work

There are several interesting concepts that use the spatiotemporal coherency, and we performed experiments that produced surprisingly good results. However, due to project deadlines, additional memory requirements, and lack of testing, those concepts were not implemented in the final iteration of the engine. We would like to present our findings here and improve upon them in the future.

Antialiasing

The spatiotemporal framework is also easily extended to full-scene antialiasing (FSAA) at a reasonable performance and memory cost [Yang et. Al 2009]. With deferred renderers, we normally have to render the G-buffer and perform lighting computation at a higher resolution. In general, FSAA buffers tend to be twice as big as the original frame buffer in both the horizontal and vertical directions. When enough processing power and memory are available, higher-resolution antialiasing schemes are preferred.

The last stage of antialiasing is the downsampling process, which generates stable, artifact-free, edge-smoothed images. Each pixel of the final frame buffer is an average of its subsamples in the FSAA buffer. Therefore, we can easily reconstruct the valid value by looking back in time for subsamples. In our experiment, we wanted to achieve 4X FSAA. We rendered each frame with a subpixel offset, which can be achieved by manipulating the projection matrix. We assumed that four consecutive frames hold the different subsamples that would normally be available in 4X FSAA, and we used reprojection to integrate those subsamples over time. When a sample was not valid, due to unocclusion, we rejected it. When misses occurred, we could also perform bilateral filtering with valid samples to leverage spatial coherency.

Our solution proved to be efficient and effective, giving results comparable to 4X FSAA for near-static scenes and giving results of varying quality during high-frequency motion. However, pixels in motion were subject to motion blur, which effectively masked any artifacts produced by our antialiasing solution. In general, the method definitely proved to be better than 2X FSAA and slightly worse than 4X FSAA since some high-frequency detail was lost due to repeated resampling. Furthermore, the computational cost was insignificant compared to standard FSAA, not to mention that it has lower memory requirements (only one additional full-resolution buffer for caching). We would like to improve upon resampling schemes to avoid additional blurring.

High-Quality Spatiotemporal Reconstruction

We would like to present another concept to which the spatiotemporal framework can be applied. It is similar to the one used in antialiasing. Suppose we want to draw a full-resolution frame. During each frame, we draw a l/*n*-resolution buffer, called the *refresh buffer*, with a different pixel offset. We change the pattern for each frame in order to cover the full frame of information in *n* frames. The final image is computed from the refresh buffer and a high-resolution history buffer. When the pixel being processed is not available in the history or refresh buffer, we resort to bilateral upsampling from coarse samples. See Figure 7.12 for an overview of the algorithm. This solution speeds up frame computation by a factor of *n*, producing a properly resampled high-resolution image, with the worst-case per-pixel resolution being l/*n* of the original. Resolution loss would be mostly visible near screen boundaries and near fast-moving objects. However, those artifacts may be easily masked by additional processing, like motion blur. We found that setting $n = 4$ generally leads to an acceptable solution in terms of quality and performance. However, a strict rejection and bilateral upsampling policy must be used to avoid instability artifacts, which tend to be disturbing when dealing with high-frequency details. We found it useful with the whole light accumulation buffer, allowing us to perform lighting four times faster with similar quality. Still, several instability issues occurred that we would like to solve.

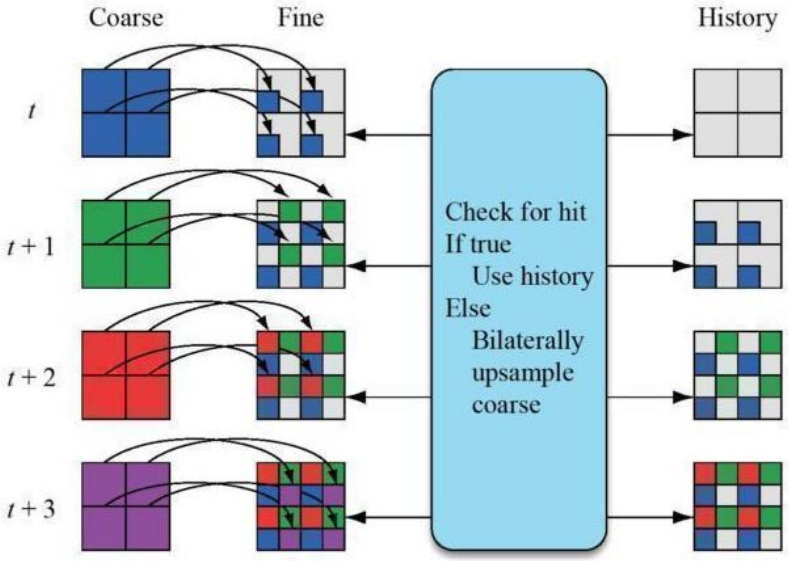


Figure 7.12. Schematic diagram of spatiotemporal reconstruction.

References

[Kopf et al. 2007] Johannes Kopf, Michael F. Cohen, Dani Liscinski, and Matt Uyttendaele. "Joint Bilateral Upsampling." *ACM Transactions on Graphics* 26:3 (July 2007).

[Lengye12010] Eric Lengyel. "Motion Blur and the Velocity-Depth-Gradient Buffer." *Game Engine Gems* 1, edited by Eric Lengyel. Sudbury, MA: Jones and Bartlett, 2010.

[Nehab et al. 2007] Diego Nehab, Padro V. Sander, Jason Lawrence, Natalya Tararchuk, and John R. Isidoro. "Accelerating Real-Time Shading with Reverse Reprojection Caching." *Graphics Hardware,* 2007.

[Scherzer et al. 2007] Daniel Scherzer, Stefan Jeschke, and Michael Wimmer. "PixelCorrect Shadow Maps with Temporal Reprojection and Shadow Test Confidence." *Rendering Techniques*, 2007, pp. 45-50.

[Tomasi and Manduchi 1998] Carlo Tomasi, and Roberto Manduchi. "Bilateral Filtering for Gray and Color Images." *Proceedings of International Conference on Computer Vision*, 1998, pp. 839-846.

[Yang et al. 2009] Lei Yang, Diego Nehab, Pedro V. Sander, Pitchaya Sitthi-Amorn, Jason Lawrence, and Hugues Hoppe. "Amortized Supersampling." *Proceedings of SIGGRAPH Asia* 2009, ACM.