# Exploiting Temporal Coherence in Real-Time Rendering

Daniel Scherzer[*]
LBI for Virtual Archaeology,
Vienna University of Technology

Lei Yang[†]
The Hong Kong University
of Science and Technology

Oliver Mattausch[‡]
Vienna University
of Technology

## Abstract

*Temporal coherence* (TC), the correlation of contents between adjacent rendered frames, exists across a wide range of scenes and motion types in practical real-time rendering. By taking advantage of TC, we can save redundant computation and improve the performance of many rendering tasks significantly with only a marginal decrease in quality. This not only allows us to incorporate more computationally intensive shading effects to existing applications, but also offers exciting opportunities of extending high-end graphics applications to reach lower-spec consumer-level hardware.

This course aims to introduce participants to the concepts of TC, and provide them the working practical and theoretical knowledge to exploit TC in a variety of shading tasks. It begins with an introduction of the general notion of TC in rendering, as well as an overview of the recent developments in this field. Then it focuses on a key data structure – the *reverse reprojection cache*, which is the foundation of many applications. The course proceeds with a number of extensions of the basic algorithm for assisting in multi-pass shading effects, shader antialiasing, casting shadows and global-illumination effects. Finally, several more general coherence topics beyond pixel reuse are introduced, including visibility culling optimization and object-space global-illumination approximations. For all the major techniques and applications covered, implementation and practical issues involved in development are addressed in detail.

In general, we emphasize "know how" and the guidelines related to algorithm choices. After the course, participants are encouraged to find and utilize TC in their own applications and rapidly adapt existing algorithms to meet their requirements.

The version of the course notes you are currently reading was created at September 23, 2010. The newest version of these course notes can be downloaded from this URL.

## Course Prerequisites

This course should give a computer graphics practitioner, student or researcher the working and theoretical knowledge to implement state-of-the-art temporal coherence techniques. The content of this course is delivered by talks and course notes. Prerequisites incldue knowledge of basic real-time computer graphics, such as programmable shading pipeline, model transformation, rasterization and texture mapping. Experience in writing vertex and fragment shaders is preferred.

## Syllabus

1. Introduction – *Daniel Scherzer (10min)*:
   1.1. What is temporal coherence?
   1.2. When is it useful? Example scenarios
   1.3. Applications in real-time rendering.
2. Background – *Daniel Scherzer (20min)*:
   2.1. Overview of the related techniques
      2.1.1. CPU based techniques from the old time
      2.1.2. GPU based real-time techniques
   2.2. Taxonomies - different perspectives of tackling the problem
      2.2.1. Space - where do we store data
      2.2.2. Data flow - forward vs. reverse reprojection
      2.2.3. Data reuse locality
3. Image-space real-time reverse reprojection – *Lei Yang (45min)*:
   3.1. The essentials
   3.2. Implementation
      3.2.1. Determining cache coordinates
      3.2.2. Detecting cache misses
      3.2.3. Cache resampling and filtering
      3.2.4. Refreshing strategies
      3.2.5. Control flow optimization
   3.3. Determining what to cache
   3.4. Analysis
      3.4.1. Computational overhead
      3.4.2. Resampling error
      3.4.3. Quality - speed tradeoff
   3.5. Direct application examples
4. Applications and extensions of reverse reprojection (Part I) – *Lei Yang (30min)*:
   4.1. Multipass rendering effects
      4.1.1. Stereoscopic rendering
      4.1.2. Motion blur
      4.1.3. Depth of field effects
   4.2. Amortized sampling
      4.2.1. Theory
      4.2.2. Achieving subpixel accuracy: shader antialiasing
5. Break (15mins)
6. Applications and extensions of reverse reprojection (Part II) – *Daniel Scherzer (40min)*
   6.1. Application in discrete LOD blending
   6.2. Application in casting shadows
      6.2.1. Pixel correct shadows
      6.2.2. Soft shadows
   6.3. Application in global illumination
      6.3.1. Screen-space ambient occlusion
      6.3.2. Imperfect shadow maps
   6.4. Spatio-temporal upsampling
7. Temporal coherence in object space – *Oliver Mattausch (50min)*
   7.1. Temporal coherence in culling techniques
      7.1.1. Hardware occlusion queries
      7.1.2. Coherent hierarchical culling (CHC)
      7.1.3. Near optimal hierarchical culling (NOHC)
      7.1.4. Coherent hierarchical culling revisited (CHC++)
   7.2. Incremental instant radiosity
8. Wrap Up – *All (15 min)*
   8.1. Conclusion
   8.2. Panel style Q&A discussion

[*]e-mail: scherzer@cg.tuwien.ac.at
[†]yanglei@cse.ust.hk
[‡]matt@cg.tuwien.ac.at

## 1 Introduction

One of the driving forces of computer graphics is to render physically correct images with rich visual effects. This usually requires large scenes with highly detailed geometric models, as well as computationally intensive shading work to be incorporated in a modern rendering system. Real-time rendering has the conflicting goal of creating a sequence of such images fast enough to still allow for continuous animation and user interaction. Here a limit of at least 60 frames per second is considered as sufficiently smooth for the human observer, which means the time available for one frame is about 16 milliseconds. All calculations necessary to create a frame have to fit into this time budget. This not only includes all the rendering algorithms we are concerned with in computer graphics, but may also contain the domain specific code of an application, artificial intelligence, input processing and sound rendering.

Although computer graphics hardware has made staggering advances in terms of speed and programmability, there still exist a number of algorithms that are too expensive to be computed in this time budget. A few important examples include physically correct shadows, depth of field and motion blur effects, or even an ambient occlusion approximation to the exact global illumination solution. The situation becomes worse when these effects are combined with large and complex scenes, in which the hidden geometry often consumes a significant portion of render time but contributes nothing to the final images.

One way to circumvent this hard time limit is to capitalize on *temporal coherence* (TC) and avoid redundant computations over time. TC is hereby defined as the existence of a correlation in time of the output of a given algorithm. For example, in a scene rendered at high frame rates, there is usually very little difference in the shading over visible surfaces between two consecutive frames, and the majority of surfaces are mutually visible (see Figure 1). Therefore, computing everything from scratch in every frame is potentially wasteful. Exploiting the coherence between adjacent frames and reusing of intermediate or final shading result can therefore reduce the average shading cost of generating a single frame.

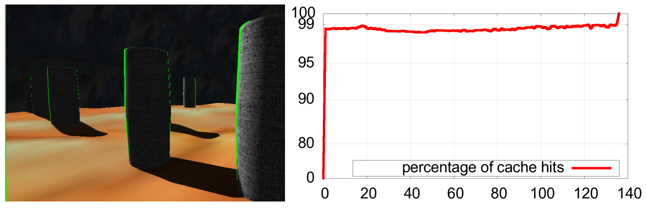


**Figure 1:** *Temporal coherence that exists in a game-like scene.* Left: *For a strafe-left movement the cache misses are shown in green.* Right: *Plot of the percentage of pixels found in the cache for each frame of the animation sequence.*

In general, TC can be applied for achieving either of the following goals:

- *Acceleration*: A given algorithm can be accelerated by reformulating it as incremental in time, thereby amortizing the total workload over several frames. The output quality may be marginally degraded but the overall speed improvement is often promising.

- *Quality improvement*: The results of a given algorithm can be augmented by taking into account results computed in previous frames. By a slight increase in render time, the quality of the result can often be significantly improved.

These goals have in common that for a drastic change in the input some latency in the output may be introduced. In the acceleration case this requires a major refresh in the previously computed results, which may cause a sudden drop of framerate. In the quality improvement case, this means that over several frames only an approximate solution can be displayed before the algorithm converges. Fortunately with relatively high framerates and careful algorithmic designs, these problems can often be handled smoothly and unnoticed by the viewer. In addition, ongoing animation may also cause information from previous frames to be outdated. This has to be accounted for in order to avoid temporal artifacts such as after-images or tailing.

Aside from the fact that the redesign of algorithms to account for TC can be challenging, special care has to be taken to fit these algorithms to the massively parallel nature of modern graphics architectures.

In the following text we want to give a detailed discussion of various approaches that exploit TC in real-time rendering. We start in Section 2 with a survey of were and how TC techniques have been employed in computer graphics. Section 3 explains the general theory behind the most commonly used data structure for exploiting TC in real-time rendering, the *reverse reprojection cache*. This data structure, although ubiquitous in real-time rendering, is often not enough to gain the maximum effect of TC. That's why many practical approaches also have intricate algorithms. We will describe the most prominent ones in Section 4. Image space methods are not the only way to make efficient use of TC. Object space approaches are widely used for visibility determination. We discuss them in Section 5.

## 2 Background

The term *frame-to-frame coherence* was first introduced by Sutherland et al. (1974) in his seminal paper "Characterization of Ten Hidden-Surface Algorithms", in which he describes various versions of coherence, like scan-line or area coherence that allow for more efficient rendering.

In an early paper by Hubschman and Zucker (1981), frame-to-frame coherence is investigated to accelerate visibility detection. They analyse animation sequences for static scenes and a continuously moving camera in scan-line rendering. From this they are able to derive a number of geometrical constrains in visibility for the case of closed, convex and non-intersecting polyhedra. Prediction of changes in visibility become possible and therefore only those parts of the scene need to be processed where a visibility change may occur. Coorg and Teller (1996) build on this work by introducing a data structure that helps to predict imminent visual events. TC allows them to maintain this data structure dynamically, only containing the data relevant for the current view port.

### 2.1 Ray-Tracing

Especially in ray-tracing, a number of algorithms exist that make use of information stored over time:

Badt Jr. (1988) introduced reprojection to accelerate ray-tracing for animations. His forward reprojection algorithm uses object space information stored from the previous frame. This allows him to approximate ray-traced animation frames of diffuse polygons. Adelson and Hodges (1995) later extend his approach to ray-tracing of arbitrary scenes.

Adelson and Hodges (1992) also use frame-to-frame coherence not in the temporal dimension, but to relate the two images of stereoscopic views when ray-tracing. They found that up to 95% of the pixels of the left-eye view can be reused for the right-eye view by reprojection.

Jevans (1992) and Davis (Davis, 1998; Davis and Davis, 1999) use TC in ray-tracing for predefined animation sequences in similar ways: They divide a given scene into an object-space grid. All voxels where objects move in/out/around at some point during the sequence are identified. Rays that pass through these changing voxels have to be recalculated in all the frames where these changes occur. While Jevans' algorithm is formulated in a serial manner, Davis' algorithm accelerates parallel ray-tracing on a distributed system by rendering different sub-regions of the output image on different nodes and employing TC locally on each node.

Havran et al. (2003a) use TC to reuse ray/object intersections in ray casted walkthroughs. They do this by reprojecting and splatting visible point samples from the last frame into the current, thereby avoiding the costly ray traversal for more than $78\%$ of the pixels in their test scenes.

## 2.2 Image-Based Rendering

Many publications focus on using TC for replacing parts of a scene with image-based representations: Gröller (1992) and Schaufler (1996) reuse image data generated from previous frames. Replacement of complex distant geometry with impostors allows them to reduce rendering times.

Lengyel and Snyder (1997) employ frame-to-frame coherence to guide factorization of a scene into multiple layers. Image warping is used to rerender layers over multiple frames. Differences in perception of fore-/background objects, as well as differences in the motion of objects are factors to redistribute rendering resources adaptively. More rendering resources are spent for fast moving foreground objects, which thereby gain in fidelity, while a slowly changing background receives less rendering resources. They targeted their implementation on a hardware termed Talisman (Torborg and Kajiya, 1996) that natively supports rendering of layers.

Regan and Pose (1994) propose an address recalculation hardware for head mounted displays. This hardware allows for orientation viewport mapping after rendering, which minimizes latency caused by head rotations. With this hardware they implement priority rendering: The scene is divided into layers with increasing distance to the eye. Layers that are farther away are updated less often than nearer layers.

Leaving the concept of frame-based rendering behind, Bishop et al. (1994) introduce frameless rendering, which heavily relies upon TC for sensible output. Here each pixel is rendered independently based on the most recent input, thereby minimizing lag. There is no wait period till all pixels of a frame are drawn, but individual pixels stay visible for a random time-span, until they are replaced with an updated pixel. Note that this approach does not use the object coherency that is such an integral part of many polygon renderers. To avoid image tearing pixels are rendered in a random order.

## 2.3 Image Warping

Another class of approaches that use TC is image warping. Here images are used as a cache to be reused and warped into different views. The most common approach to warping is to forward map the individual pixels of the image into the new view and then splat them with a Gaussian kernel.

Chen and Williams (1993) calculate in-between views by morphing a number of reference images. Image morphing is defined as the simultaneous interpolation of shape and texture. Two steps are necessary: first, identification of correspondences between the two images, resulting in a mapping and second, interpolation by blending the corresponding pixels. The first step is the more intricate one, but in this special case of in-between views it can be resolved by using the camera transformation between image spaces (forward mapping).

McMillan and Bishop (1995) extend this to arbitrary viewing positions by an image warping approach. This technique is used in stereoscopic displays with only two hyper-stereo reference images as input. Mark et al. (1997) build on McMillan and Bishop's image warping algorithm to render a new view from two stored views. For each view color and depth are stored. Both images are warped into the new view to allow for small camera movements. Then the two images are composited together to compensate for most disocclusions. The authors report an increase in apparent rendering speed of about one order-of-magnitude to regular rendering.

Shade et al. (1998) present two warping methods. The first is for warping a sprite with depth information. The major problem here is that backward mapping cannot be applied directly and forward mapping may produce holes. The authors suggest therefore to first forward map the displacements given by the depth information from the sprite and then backward map these warped displacements. Note that in the now possible backward mapping step reconstruction filters can be employed. This sprite approach works well for planar or smoothly varying surfaces. For more complex surfaces the authors propose a new data structure: the *layered depth image* (LDI). Here each pixel may contain multiple depth and color informations, allowing to deal with large parallax and general disocclusions. An LDI can be rendered by splatting all pixels into the output image using the *over* compositing operation.

Also in terrain rendering image based approaches have been used together with TC. Chen et al. (1999) render the full-resolution terrain mesh into an image sprite and use this sprite as a texture (employing projective texture mapping) for a low resolution terrain mesh in subsequent frames. Each frame the error this produces is measured and if too high the image sprite is recreated from the view of the current camera. This unfortunately introduces uneven frame rates, if the image sprite has to be recreated. Although this approach was applied to terrains in the paper, it could also be used for other geometry.

Qu et al. (2000) use image warping to accelerate ray-casting. The idea is to warp the output image of the previous frame into the current frame. Due to the warping, pixels may fall between the grid positions of the pixels of the current frame, therefore an offset buffer is used to store the exact positions. Due to disocclusions, holes can occur at some pixels. Here ray-casting is used to generate these missing pixels. The authors propose to use an age stored with each pixel, which is increased with each warping step to account for the lower quality of pixels that have been warped (repeatedly). Upon rendering a new output frame this age can be used to decide if a pixel should be re-rendered or reused.

Simmons and Sequin (2000) introduce a mesh-based reconstruction called a tapestry for dynamically sampled environments. It allows the reuse of radiance values across views by reprojecting them into the new view.

Stamminger et al. (2000) augment interactive walkthroughs (calculated by rendering hardware) with photorealistic results calculated by an asynchronous process. This has the advantage that interactivity of the walkthrough is not hindered by the processing time of the high quality calculations. The differences between high quality and interactive results are stored in so-called *corrective textures* and applied to the scene objects with projective texturing. This has the advantage that the TC of such scenes allows for lazy updates of corrective textures. Additionally the texture resolution can be adapted to the number of available samples and hardware texture filtering can be used to resolve under- as well as oversampling.

Wimmer et al. (1999) accelerate the rendering of complex environments by using two different rendering methods for the near and far field: The near field is rendered using the traditional rendering pipeline, while ray casting is used for the far field. To minimize the number of rays cast, they use a panoramic radiance cache and estimate the horizon, to avoid to ray cast sky pixels. If an upright viewer is assumed, finding the horizon can be solved by casting a 2d ray through a precomputed height field.

Walter et al. (1999) introduce the *render cache*. It is intended as an acceleration data structure for renderers that are too slow for interactive use. The *render cache* is a point based structure, which stores previous results, namely 3d coordinates and shading information. By using reprojection, image space sparse sampling heuristics and by exploiting spatio-temporal image coherence these results can be reused in the current frame. Progressive refinement allows decoupling the rendering and display frame rates, enabling high interactivity. Walter et al. (2002) later extend this approach with predictive sampling and interpolation filters. Finally Velázquez-Armendáriz et al. (2006) and Zhu et al. (2005) accelerate the *render cache* on the GPU.

Smky et al. (2005) explore TC to speed up irradiance calculations using a cache structure called anchor. The idea is to permanently store and update (if needed) all the incoming radiance samples used to estimate the irradiance of an irradiance record. Thereby they not only accelerate the process, but also reduce temporal artifacts, like flickering in methods that render each frame independently.

In a related approach, Gautron (2008) present a temporal caching scheme for glossy global illumination: *temporal radiance caching* of animated environments (camera, objects and light sources move). They reuse part of the global illumination solution of previous frames by introducing temporal gradients, which estimate contribution of a record within its lifespan.

## 3 Image-Space Real-Time Reverse Reprojection

In this section, we describe a simple and lightweight method – the *Reverse Reprojection Cache* (RRC), which reuses shading results from previously rendered frames to reduce shading costs. This framework is used by many applications described later in this tutorial. The basic idea was proposed individually by Nehab et al. (2006, 2007) and Scherzer et al. (2007). RRC stores previous shading results in screen space, thereby avoiding complex data structures and parametrization problems, and allowing efficient implementation on programmable graphics hardware. We will discuss the method in detail from theory to implementation, including the various alternatives for mapping different stages of the algorithm onto shader programs. We also give a detailed analysis of the associated computational overhead, output quality, and the tradeoff between the two. Since the method is not limited to caching the final shading color, we give general guidelines for determining which part of the shader to cache. Finally, we show several examples of directly applying this method to accelerate fill-bound scenes. Some of the materials are adapted from Nehab et al. (2007) with permission.

### 3.1 The Essentials

The key concept in the RRC method is to establish a reverse pixel mapping by reprojection. The shading result from the previous frame (also referred to as *payload* hereafter) is stored in a separate framebuffer (cache). For each pixel generated in the new frame, we know the surface point from which it originated. We also know where this surface point was, in 3D space, at the time the previous frame was rendered. From this knowledge, we can easily find where it previously projected to, and test whether it was visible at that time. We can then fetch whatever surface information we stored in the previous framebuffer, and use it while rendering the new frame. In summary, the basic idea here is to let the rendering of the current frame gather and reuse shading information from surfaces visible in the previous frame.

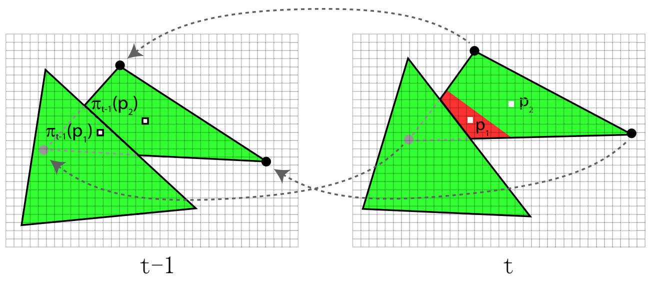


**Figure 2:** *The reverse reprojection operator. The shading result and pixel depths of time $t$-1 are stored in screen-space framebuffers (left). For each pixel $\mathbf{p}$ at time $t$ (right), its reprojected position $\pi_{t\text{-}1}(\mathbf{p})$ is computed to locate the corresponding position at frame $t$-1. The recomputed scene depth is compared to the stored pixel depth. A pair of matching depths indicate a cache hit ($\mathbf{p}_2$), whereas inconsistent depths indicate a cache miss ($\mathbf{p}_1$).*

Formally, let $f_t$ denote the framebuffer at time $t$. This buffer holds the cached pixel attribute that can be reused in a later frame. We also keep an accompanying buffer $d_t$ which holds the scene depth in screen space. Let $f_t(\mathbf{p})$ and $d_t(\mathbf{p})$ denote the buffer values at pixel $p \in \mathbb{Z}^2$. For each pixel $\mathbf{p} = (x, y)$ at time $t$, we determine the 3D clip-space position of its generating scene point at frame $t$-1, denoted $(x', y', z') = \pi_{t\text{-}1}(\mathbf{p})$. Here the reprojection operator $\pi_{t\text{-}1}(\mathbf{p})$ correlates point $\mathbf{p}$ with its previous position at frame $t$-1. Note that with this reprojection operation, we also obtain the depth of the generating scene point $z'$ at frame $t$-1. This is used to test whether the current point is visible in the previous frame. If the reprojected depth $z' \approx d_{t\text{-}1}(x', y')$ within some tolerance, we conclude that the current pixel $\mathbf{p}$ and the reprojected pixel $f_{t\text{-}1}(x', y')$ are indeed generated by the same surface point, thereby the previous value can be reused. Otherwise no correspondence exists and we denote this by $\pi_{t\text{-}1}(\mathbf{p}) = \varnothing$, referred to as a cache miss. The reprojection operation is illustrated in Figure 2.

Applying the reprojection operator to accelerate expensive pixel shading computation is then straightforward. Figure 3 shows the schematic diagram of how we achieve this. When each pixel $\mathbf{p}$ is generated, the reprojection shader fetch the value at $\pi_{t\text{-}1}(\mathbf{p})$ in the cache and tests if the result is valid (i.e. cache hit). If so, the shader can reuse this value in the calculation of the final pixel color. Otherwise, the shader executes the normal pixel shading. Whichever route the shader follows, it always stores the cacheable value for potential reuse during the next frame.

Next we will map this scheme into a simple and efficient implementation.

### 3.2 Implementation

In this section, we provide detailed description as well as code snippets for mapping the different stages of the RRC algorithm into a

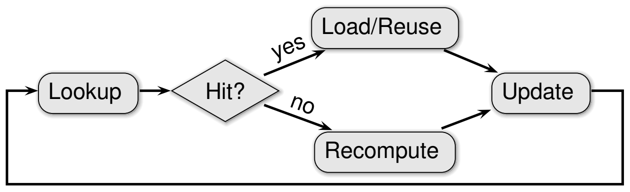


**Figure 3:** *Schematic diagram of the reverse reprojection caching process (Courtesy of Diego Nehab).*

minimum yet complete implementation. We also include several details and improvements with hindsight that are not described in the original papers (Nehab et al., 2006, 2007; Scherzer et al., 2007).

### 3.2.1 Determining Cache Coordinates

The main computational challenge we face is to efficiently compute the reprojection operator, which maps the location of a pixel's corresponding scene point in the previous frame. Fortunately, this operator only need to be computed at vertex level and we leverage the hardware interpolator between the vertex and the pixel stage to deliver the per-vertex reprojection result to the pixel level.

Assume the cache buffer $f_{t-1}$ computed in the previous frame is valid. At frame $t$, the homogeneous projection space coordinates $(x_t, y_t, z_t, w_t)_{vert}$ of each vertex $\mathbf{v}$ are calculated in the vertex shader, to which the application has provided the world, view and projection matrices and any animation parameters (such as blending matrices and factors used for skinning). To perform correct reprojection, the application also has to provide these matrices and animation parameters at $t-1$. A duplicated transformation code in the vertex shader uses these data and compute the projection-space coordinates $(x_{t\text{-}1}, y_{t\text{-}1}, z_{t\text{-}1}, w_{t\text{-}1})_{vert}$ of the same vertex at frame $t-1$. These coordinates become attributes of each transformed vertex, which are passed down to the pixel shader, causing the the hardware to interpolate them automatically. This automatically gives each pixel $\mathbf{p}$ access to the projection-space coordinates $(x_{t\text{-}1}, y_{t\text{-}1}, z_{t\text{-}1}, w_{t\text{-}1})_{pix}$ of the generating surface point at time $t$-1. The final cache coordinates $p_{t\text{-}1}$ are obtained with a simple division by $(w_{t\text{-}1})_{pix}$ within the pixel shader. Figure 4 shows the example shader code for this process. Note that although only rigid transformation is shown, the algorithm can be combined with arbitrary complex vertex tasks.

*Vertex Shader*

```
1  VS_OUTPUT RenderSceneVS(...)
2  {
3     VS_OUTPUT Out;
4     // proj-space coordinate for the current frame
5     Out.Pos = mul(vPos, g_mWVP);
6     // proj-space coordinate for the previous frame
7     Out.PosPrev = mul(vPos, g_mWVP_Prev);
8     return Output;
9  }
```

*Pixel Shader*

```
1  float4 RenderScenePS(VS_OUTPUT In)
2  {
3      // perspective division
4      In.PosPrev /= In.PosPrev.w;
5      // transform coordinates from NDC to screen
6      In.PosPrev.xy = (In.PosPrev.xy + 1.f) * 0.5f;
7      In.PosPrev.y = 1.f - In.PosPrev.y;
8      // fetch the previous value from cache
9      float4 cache_val = g_txCache.Sample(
           BiLinearSampler, In.PosPrev.xy);
10     ...
11 }
```

**Figure 4:** *Sample shader code for computing the cache coordinate and fetch the corresponding data in the cache.*

*Pixel Shader*

```
1  bool bHit =
2      // clipped case (within [0,1]x[0,1])
3      saturate(In.PosPrev.xy) == In.PosPrev.xy &&
4      // occlusion case (depth match)
5      abs(In.PosPrev.z-cache_val.w) < g_fZThres;
```

**Figure 5:** *Sample shader code for detecting a cache miss.*

### 3.2.2 Detecting Cache Misses

Cache miss refers to those occasions when a surface point at frame $t$ was invisible at frame $t-1$, therefore no cached value can be obtained. This could happen when the previous position of the point lies outside the viewport (clipped), or was blocked by an irrelevant object closer to the camera at the time (occluded). Both cases can be detected by examining the reprojected clip-space 3D position $\pi_{t\text{-}1}(\mathbf{p})$ of the point of interest, as discussed briefly in Section 3.1. For the clipped case, we simply test if the reprojected screen space XY coordinate (Figure 4, line 9 in the pixel shader) is beyond the viewport boundaries $[0,1] \times [0,1]$. Fortunately this can be efficiently done in a single comparison instruction. For the occluded case, we need to compare the reprojected depth $\pi_{t\text{-}1}(\mathbf{p}).z$ with the previous depth $d_{t\text{-}1}(x', y')$ fetched from the cache. If the cache depth is within a small tolerance $\sigma$ of the reprojected (expected) depth, we conclude that cache value is indeed from the same piece of geometry that we are rendering, and report a cache hit. Otherwise we report a cache miss. Figure 6 illustrates cache miss due to occlusion changes between consecutive frames. A sample shader code fragment of this process is shown in Figure 5.

We use a bilinear texture fetch to interpolate the depth stored in the cache. For smooth surfaces, this is supposed to interpolate the nearest four depth values as if they are on a plane. Note that since depth is usually stored non-linearly, this interpolation is not accurate but usually provides an acceptable approximation. For discontinuity boundaries, the mixed values normally do not match the depth of either side, thus automatically lead to a cache miss (this is further discussed in Section 3.2.3).

A tricky case of of depth separation is that when multiple objects intersect. At such points, pixels across the intersection boundary may have very similar depths, such that samples from an incorrect object may slip through the depth match test. This often makes moving objects leave a trail over the background. One method to avoid this is to assign ID to different objects and test if the IDs match as well. If the number of objects are small, the IDs can even be conveniently packed with the depth value as the most significant bits, so that depth comparison automatically fails if the IDs do not match.

To improve robustness, the depth match threshold $\epsilon$ can be set as the resolution of the depth buffer. Due to the error introduced in interpolating non-linear Z value or by surfaces with high curvature, it may be necessary to increase this value. However, with a 16-bit-per-channel cache buffer, we found this error negligible. Alternatively, with a floating point complementary Z-buffer, the threshold can be set according to Akeley and Su (2006) using their formula based on a theoretical analysis of the error.

### 3.2.3 Cache Resampling and Filtering

In general, when retrieving values from the cache, the reprojected position $\pi_{t\text{-}1}(\mathbf{p})$ lies somewhere between the set of discrete samples in the cache buffer $f_{t\text{-}1}$ and thus some form of resampling is required. This resampling often involves computing a weighted sum of the values in some vicinity of $\pi_{t\text{-}1}(\mathbf{p})$. The most common approach for reconstructing pixel value at fractional positions is bilinear filtering, which is directly supported by the hardware. In most situations this suffices for practical use. However, if one wishes to reuse cached values over many (e.g. $> 5$) frames, then the resampling error accumulates repeatedly in each frame and finally leads to an overblurred result. This is further analyzed in Section 3.4.2. To remedy this, higher-order reconstruction can be used. The most commonly used high-quality image resampling filters are the Lanczos2 and the Mitchell-Netravali $(1/3, 1/3)$ kernel. By applying either of these kernels to a local $4 \times 4$ neighborhood of the reconstruction center and compute the convolution, we can obtain a value of better accuracy and less smoothing. Typically we observe 1/3 to 1/2 times less blur by switching to such kernels.

Neither bilinear nor the higher-order reconstruction methods described above handles motions that involve minification and magnification. In such cases the reprojected pixel size differs from that in the cache, so aliasing or overblur artifact may appear if not filtered properly. In the case of minification, the pixel of interest may cover multiple pixels in the cache. It is often worthwhile to precompute a mip-chain of the cache and leverage the hardware trilinear texture fetch to sample the correct mip level. Magnification, on the other hand, is usually more difficult to handle, since the cached image do not have the information to resample to higher resolution without blur. Typically, pixels that reproject to fractional pixel positions in the cache are inaccurate. The closer to the half-way between pixels (i.e. grid center), the larger error. It is therefore beneficial to force refresh (reshade) these pixels with large error. This can be achieved by comparing the offset distance in the pixel grid with the reprojected pixel size. If the former is larger, then trigger a reshade. A sample code of this is provided in Figure 7. Note, however, that such a strategy may trigger large and incoherent regions of refreshment, which may significantly lower the performance. In practice the minification/magnification scale is often small because there is usually little change in the scene between frames. Therefore, it is recommended to apply this only when strictly necessary.

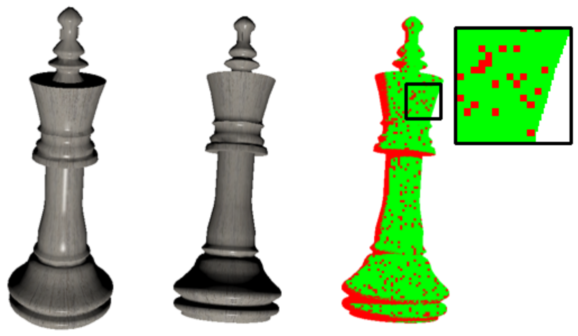


**Figure 6:** *Illustration of the regions of reuse and refresh. Pixels that are reused are shown in green in the hitmap. Pixels previously occluded (cache miss) or explicitly refreshed are computed from scratch and shown in red. Note that explicit refreshes occur along a random pattern in the framebuffer (Section 3.2.4).*

*Pixel Shader*

```
// "integer" value of the reprojected position
float2 IPrev = In.PosPrev.xy * ScnSz.xy;
// reprojected pixel radius
float2 PixR = max(ddx(IPrev), ddy(IPrev))*0.5;
// trigger a cache miss if the distance from
// the reprojected position to the nearest
// pixel center is larger than PixR
bool bHit = bHit &&
    all(abs(IPrev - round(IPrev)) <= PixR);
```

**Figure 7:** *Sample shader code for detecting magnification in motion and reporting a cache miss for potentially inaccurate pixels.*

In addition, the traditional filtering schemes do not produce correct results at discontinuity boundaries even when the pixels match perfectly with those in the cache. When filtered, the edge pixels are always reported as cache miss because the averaged depth do not match the depths on either sides. This can be improved by introducing a bilateral filter (Tomasi and Manduchi, 1998) for edge-aware filtering. Before averaging the neighboring samples, they are first tested if they match the desired depth (with a threshold looser than $\epsilon$). Samples that do not pass the test are rejected. The rest are averaged with spatial weights as usual. In this way, only the samples from the correct geometry are averaged and filtered. This will significantly reduce the false cache-miss pixels at geometry boundaries. However, a bilateral filter cannot be implemented using the hardware bilinear filtering mechanism and therefore can be much slower to execute. In most applications bilinear is still more preferable since usually the additional false-miss edge pixels only occupy a small percentage of the screen.

### 3.2.4 Refreshing Strategies

The cached value cannot be reused forever, since the shading may change over time and the cached signal may be degraded after repeated resampling over frames. It is usually necessary to refresh the cache (i.e. recompute shading) periodically in order to prevent the shading error from accumulating excessively. Of course, for rendering acceleration the refreshment of the whole screen should be amortized over many frames relatively evenly in order to reduce the shading cost and ensure a smooth framerate. Therefore, for a fixed refresh rate, we divide the screen into $n$ parts and update them in a round-robin fashion in each frame using the condition

$$(t+i) \mod n = 0 \qquad (1)$$

by testing it in the pixel shader. There are two basic ways of dividing the screen:

**Tiled refresh regions**. We can partition the screen into a grid of $n$ non-overlapping tiles. A global clock $t$ is incremented at each frame and passed into the pixel shader. For each pixel, the pixel shader computes $i$ as the tile index simply using its position.

**Randomly distributed refresh regions**. The screen pixels are randomly partitioned into $n$ sets with relatively same size. This is achieved by precomputing and storing a randomly distributed integral offset $i$ with each $2 \times 2$ pixel block (explained below).

Tiled refresh provides excellent coherence of refreshed pixels and can be efficiently executed on the GPU. This works best for signals changing very slowly with a relatively small refresh period $n$, so that the difference across tile boundaries are not noticeable. Randomly distributed refresh on the other hand, trades the clear boundaries between grids for high-frequency noise that is evenly distributed across the image. If the signal changes fast or the refresh period is large, then visible random block pattern will still be visible, but is usually much less objectionable than clear tile boundaries. However, one has to be careful implementing this on the GPU, since per-pixel refresh decision does not work well with GPU pixel-quad shading mechanism. We found that enforcing the same offset $i$ in $2 \times 2$ pixel regions significantly improves the performance. This pattern is also visualized in Figure 6. Larger block size may lead to slightly better performance on some GPUs because of potential misalignment of refreshment block with the GPU shading quad. However, block noise can be generally more noticeable with larger blocks.

In certain applications it is possible to predict the underlying signal changes. It is therefore desirable to dynamically change $n$ (per pixel block) to force quicker update in response to signal change in certain regions. Note that for an arbitrary $n$, the screen pixels has to be divided into enough groups (i.e. the range of $i$ is larger than $n$) for Equation 1 to function correctly. In addition, some damping of $n$ change is needed for the algorithm to complete at least one update round of all the pixels.

Due to arbitrary scene motion, it is not strictly guaranteed that all the pixels are refreshed after $n$ frames. This is normally less of a problem since it rarely happens and is often obscured by random noises. If there is a free channel in the cache (e.g. the cache is only storing grayscale values or depth is stored separately rather than in the alpha channel), it can be used to store a counter that is increased in every frame and reset when refreshed. In this way, any pixels that are misses a refresh in one cycle can be immediately detected (counter $> n$) and gets refreshed.

Finally, note that many shaders and applications of RRC do not need an explicit refreshing strategy. For example, multi-pass rendering effects such as motion blur, depth of field or stereoscopic rendering only caches data in a single frame. Another example is amortized sampling, in which the exponential smoothing filter automatically attenuates the weights of older samples.

### 3.2.5 Control Flow Optimization

The need to refresh in either cache miss or explicit refresh cases directs the shading task into two branches: reuse and refresh. Every pixel will follow only one of the branches. Although there is a substantial progress in improving branching efficiency on modern GPUs, it is still only effective at larger granularities (i.e. the decision is relatively coherent across the screen). Therefore, the actual implementation of this branching has significant impact on the efficiency of the RRC algorithm. According to Sitthi-amorn et al. (2008a), there are three different strategies for this task.

**One-pass algorithm** This most straightforward approach combines cache hit and miss branches in a single pixel shader. They are selectively executed using a single "`if`" statement in the shader (Figure 8(a)). When a group of pixels contains both cache hit and miss and are executed in parallel on the GPU, it takes the time of the longest branch to finish the bundle task.

**Two-pass algorithm** This approach uses the Z-buffer in conjunction with two rendering passes to partially address the performance loss in dynamic flow control (DFC). In the first pass, on a cache miss or forced refresh the shader simply discard the pixel so that the depth buffer is not altered to force the execution in the next pass, otherwise it reuses the cache payload and compute the shading, and write the depth as normal. The second pass executes the original pixel shader only for those pixels that need updates, based on regular depth tests (Figure 8(b)). This method leads to greater data parallelism in the second pass. However, if the shading computation after fetching the payload is non-trivial, the two execution branches are still unbalanced which can limit the speed gain with non-ideal hardware DFC.

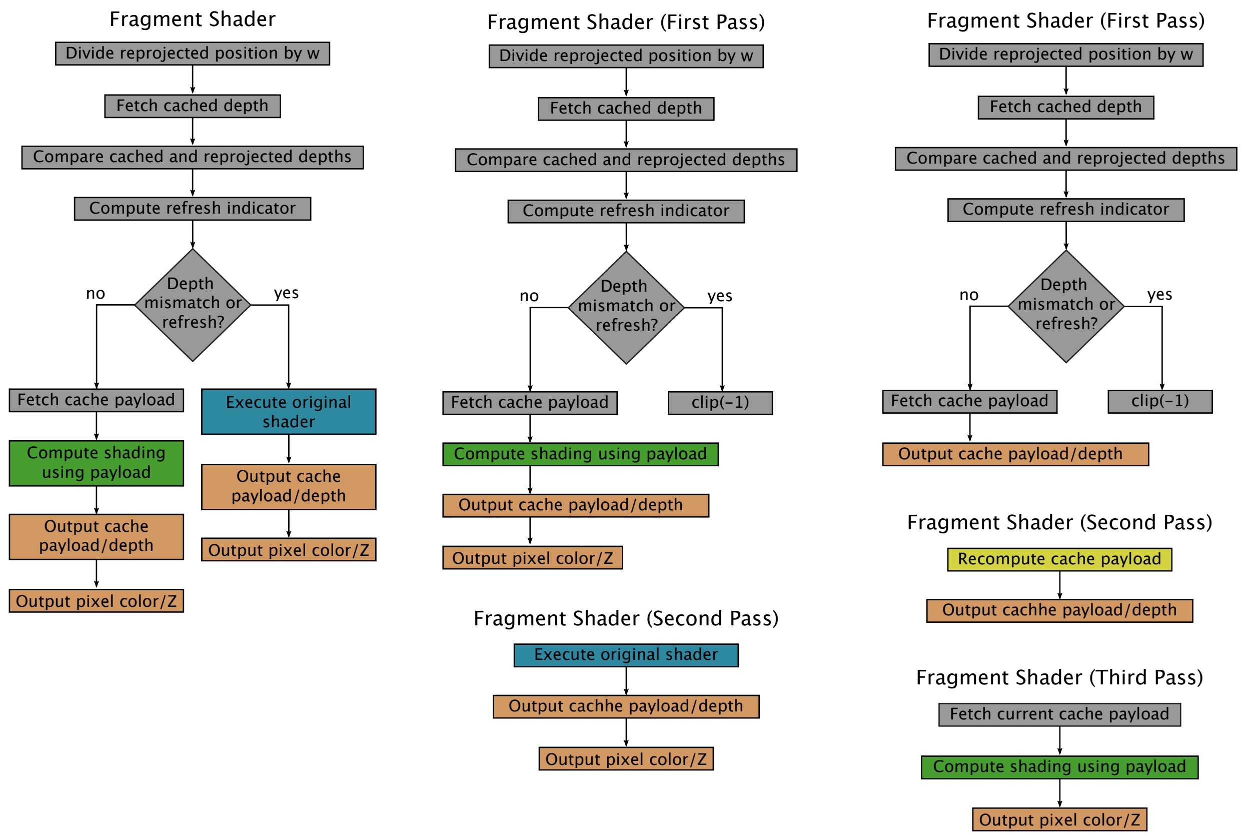
**Three-pass algorithm** By further separating the rest shading computation after obtaining the cache payload into a separate pass, we have this three-pass algorithm. The first and second passes are similar to those in the two-pass algorithm, except that they only output the cache payload to the framebuffer. The third pass fetches this cache value and compute the rest of the shading. This algorithm has all the branches balanced in the process, so that parallelism is mostly exploited. However, there are added costs of more passes for processing geometry.

As Sitthi-amorn et al. (2008a) shows, these three strategies have different advantages and are suited at different occasions. Generally, the two-pass and three-pass algorithms out-perform the one-pass one when the computation of cache payload is expensive, which is almost always true for RRC applications. The three-pass algorithm exhibits better performance if the rest of computation after obtaining the payload is also expensive. It also has the benefit that no multi-render-target (MRT) support is required, since it writes the cache and the final color in different passes. However, if the geometry is complex and additional passes are expensive to process, then the gain of two/three-pass algorithms can be small and may not warrant their use. The quantative analysis can be found in the original paper. Overall it is left to the programmer which algorithm to choose, based on the complexity of the geometry and the shader, as well as which part of the shader is cached.

### 3.2.6 Cache storage

One of the decisions in applying the RRC is the way that the cache is stored. The cache holds the value of the intermediate or final shader output that we want to reuse (i.e. payload) and the screen space depth of every pixel. There are several alternatives of extracting this data out of the previous rendering frame. When RRC is used to cache the final color output of the pixel shader, the most obvious choice is to bind the previous framebuffer as a texture for the cache. Otherwise, there need to be a separate framebuffer for the cache, and the rendering passes (either 1-pass method or 2-pass method) will write both the cache value and the final color into their corresponding buffers using MRT support. Note that as stated before, the 3-pass algorithm does not need MRT for writing to the cache, because the cache payload and the final color are written in different passes.

The pixel depth usually requires more precision than the cache payload. A straightforward implementation to access it is to directly bind the depth buffer from the previous frame. Since both the depth passed down in the shader (`In.PosPrev.z` in Figure 4) and the value retrieved from the depth buffer are expressed in Normalized Device Coordinates (NDC) space (i.e. after division by $w$), they can be directly compared. Alternatively, we can store the depth in a spare channel in the cache (e.g. usually the alpha channel is free). This has the benefit of combining the retrieval of cache value and depth into a single texture fetch, but requires that all the four channels have at least 16-bit precision each.



Fragment Shader
Divide reprojected position by w
Fetch cached depth
Compare cached and reprojected depths
Compute refresh indicator
Depth mismatch or refresh?
no
yes
Fetch cache payload
Compute shading using payload
Output cache payload/depth
Output pixel color/Z
Execute original shader
Output cache payload/depth
Output pixel color/Z
Fragment Shader (First Pass)
Divide reprojected position by w
Fetch cached depth
Compare cached and reprojected depths
Compute refresh indicator
Depth mismatch or refresh?
no
yes
Fetch cache payload
clip(−1)
Compute shading using payload
Output cache payload/depth
Output pixel color/Z
Fragment Shader (Second Pass)
Execute original shader
Output cachhe payload/depth
Output pixel color/Z
Fragment Shader (First Pass)
Divide reprojected position by w
Fetch cached depth
Compare cached and reprojected depths
Compute refresh indicator
Depth mismatch or refresh?
no
yes
Fetch cache payload
clip(−1)
Output cache payload/depth
Fragment Shader (Second Pass)
Recompute cache payload
Output cachhe payload/depth
Fragment Shader (Third Pass)
Fetch current cache payload
Compute shading using payload
Output pixel color/Z

(a) one-pass (b) two-pass (c) three-pass

**Figure 8:** *Three control flow strategies for implementing the RRC (Sitthi-amorn et al., 2008a).*

## 3.3 Determining What to Cache

Note that the value stored in the cache need not be the final pixel color, but can be any intermediate calculation that would benefit from this type of reuse. In fact, the final pixel color usually contains time- and view-dependent components. For example, on a highly polished object the specular highlight often moves quickly along the surface when the light or the viewpoint is changing. Caching such quickly changing signals often results in visual artifacts even when small $n$ is assigned. Therefore, it is important to select the appropriate portion of shading computation for caching. The general guidelines are:

1. The value cached should exhibit only weak view- and light-dependency and changes slowly over time. This guarantees that caching the value will not lead to objectionable temporal lags and refreshing artifacts;
2. The value cached corresponds to a significant portion of computation in the original shader. This ensures that caching the value reduces the cost of shading by a considerable amount.

Generally when selecting a value to cache, the programmer should seek to maximize the ratio of the saved computational effort relative to the magnitude of signal change between the frames (i.e. caching error). This is usually a manual work and requires a solid knowledge of the shader and the relative cost of its each components. Some examples of good candidates are:

Static procedural noise. The standard Perlin noise requires several texture fetches and complex ALU operations to compute. In addition, it is usually computed in multiple bands for increased variety. It is often used as stationary and contains only relatively low frequency components.

Ambient lighting. Diffuse and indirect lighting are difficult to compute in real-time even in an approximate sense. They are view independent and usually change slow over time.

Complex integral. Certain effects, such as soft shadow, SSAO, anti-aliasing, sub-surface scattering and BRDF effects require computing a large integral. The computation of the integral can usually be amortized over time.

Multi pass effects. Motion blur, depth of field and stereoscopic 3D all require generating multiple images for nearby views in each frame.

Although there are many examples shown in various papers demonstrating the effectiveness of RRC, it is still difficult sometimes for programmers to decide whether it should be applied to certain tasks, particularly when the principle of the shader is not totally clear. Sitthi-amorn et al. (2008b) proposed a system to automate this decision. The basic idea is to analyze the shader automatically and obtain the knowledge of tradeoffs in caching different components of the shader. Figure 9 shows the conceptual diagram of the system. The target shader is first analyzed and represented by an abstract syntax tree using compiler techniques. With tree node substitution, various versions of the shader that use RRC to cache different components are generated and stored in a shader pool. For each shader in the pool (correspond to each candidate caching value), an error model and a performance model (will be mentioned in Section 3.4) are fitted based on individual profiling results. Using these model parameters, an interactive profiler can be driven by the user

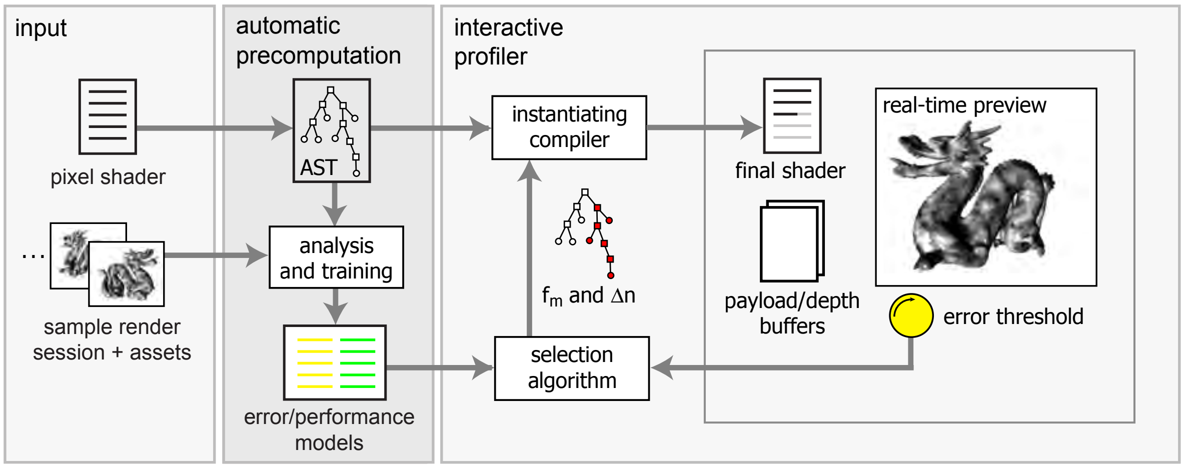


**Figure 9:** *Conceptual model of the automated reprojection-based pixel shader optimization system (Sitthi-amorn et al., 2008b).*

for choosing the best cache candidate. The user simply adjust the desired error threshold and the system will automatically select the the most efficient candidate that satisfies the error tolerance. This system automates both selecting ideal candidate and injecting codes for RRC, thus greatly eases the use of RRC tool for accelerating real-time shaders.

## 3.4 Analysis

In this section we give analysis of the quality and speed tradoff involved in applying the general RRC algorithm. In most cases we limit our discussion to results that are directly relevant to application decisions. For the detailed derivation of these results, please refer to Sitthi-amorn et al. (2008a,b); Yang et al. (2009).

### 3.4.1 Computational Overhead

The speed gain of applying the RRC method is affected by a lot of factors, including the control flow algorithm used (Section 3.2.5), refreshing strategy (Section 3.2.4), workload of the cache-hit and miss shaders, hardware flow control capabilities and granularity of parallel threads. Here we analyze a representative case with using RRC to directly optimize a fixed model/shader scene (Perlin noise albedo with Blinn-Phong specular lighting on a 75K-tris Stanford Dragon model, also described later in Section 3.4.3) with fixed caching component (low-frequency bands of noise computation only – two bands out of five). All the experiments are generated using an animation sequence that shows the object rotating at a moderate rate in front of the camera.

We measure the final average render time of various approaches for comparison. Random distributed refresh regions is used for cache refresh, with different quad sizes ($1\times1$, $2\times2$ and $4\times4$). The refresh period we tested ranges from 1 (always refresh, i.e. the original shader) to 50. We also experimented the 1-pass, 2-pass and 3-pass control flow algorithms described in Section 3.2.5. The render time we measured with these variables are shown in Figure 10.

From the graph we have several observations. The first to notice is that the 1-pass algorithm does not provide much speed gain over all the range of refresh periods. This is because the GPU dynamic flow control (DFC) mechanism only works efficiently for larger continuous regions of pixels. Bulks of concurrently shaded pixels following diverse branches will always take the time of the longest branch to execute, which unfortunately is almost always the case

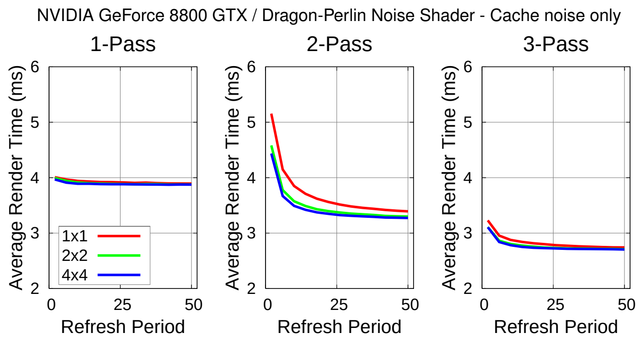


**Figure 10:** *Effect of refresh quad size and refresh period on performance.*

with small random refresh quads. With 2-pass and 3-pass algorithms, different branches are forced to shade in different passes, effectively avoiding this problem. The 3-pass algorithm is further optimized than the 2-pass, because the uncached computation (dark green block in Figure 8) is non-trivial to compute. Therefore the first pass in the 2-pass algorithm still suffers from DFC efficiency issue, as discussed in Section 3.2.5. With a lower refresh period $n$, the 2-pass algorithm can even take more time to execute than the original shader. With the 3-pass algorithm, the DFC inefficiency is mostly avoided, resulting in consistent reduction of shading time.

For all three algorithms, we consistently observed a significant drop in rendering time when going from a refresh quad size of $1 \times 1$ to $2\times2$, for the reason that GPU executes pixel shaders in at least $2\times2$ blocks. The $4 \times 4$ block can provide even larger speed gain for a similar reason. In our experiments, block sizes larger than $4\times4$ can sometimes lead to better performance, but the pattern can be more objectionable in the final rendering.

Finally, as expected, increasing the refresh period leads to a greater number of cache hits within each frame and causes a steady drop in rendering times. This rate of improvement decays exponentially, as the hit-rate $\gamma(n)$ grows geometrically with $n$, modeled with the parametric expression

$$\gamma(n) = \mu(1 - 1/n). \qquad (2)$$

In the equation, $\mu$ is the average percentage of surface area that is mutually visible between consecutive frames in the animation sequence. This number is usually greater than 90% (see Figure 1).

Assume that the shaded pixels on the screen is fixed (e.g. the model is positioned to subtend the entire viewport). We estimate two timing figures for caching a specific node in the shader: the frame time when all the pixels are cache hit ($T_h$) or cache miss ($T_m$). Then the average frame-time is predicted as:

$$T(n) = \gamma(n)T_h + (1 - \gamma(n))T_m. \tag{3}$$

With the 3-pass algorithm, this model is shown to closely agree with the actual frame time measurement for both NVIDIA and ATI hardware (Sitthi-amorn et al., 2008b). From this equation, we conclude that the speed gain is mostly determined by the difference between $T_h$ and $T_m$, as well as the hit rate (Equation 2). The curve in Figure 10 shows that too large a refresh period $n$ does not provide much speed gain over medium values, but leads to linearly slower adaptation to signal changes and allows larger reprojection error to exist. Therefore, this value should be chosen carefully according to properties of the underlying cached component (variation speed and spatial frequency). Trial and error, as well as automatic systems such as the one described by Sitthi-amorn et al. (2008b) can be used for this task.

### 3.4.2 Resampling Error

As described in Section 3.2.3, resampling is required when fetching data from the cache. Since the cached value can be reused over multiple frames, repeated resampling values may lead to undesirable blurring. This blurring has the similar effect of convolving the cache image with a Gaussian filter, of which the size or variance $\sigma$ is growing with increasing number of repeated resampling. This variance is usually used to characterize the "amount/radius of blur" that contaminates the underlying signal.

The amount of blur is related to the fractional pixel velocity after each frame of reprojection. The fractional velocity is defined as $\mathbf{v} = \pi_{t\text{-}1}(\mathbf{p}) - \lfloor \pi_{t\text{-}1}(\mathbf{p}) \rfloor$, i.e. the fractional part of the reprojection vector, which specifies the position of the bilinear kernel center in the pixel grid. Although obtaining a closed-form expression for $\sigma_t^2(\mathbf{p})$ is impractical for arbitrary scene motion (with $\mathbf{v}$ changing every frame), the case of constant panning motion with fixed $\mathbf{v}$ is tractable and can be used to locally approximate other types of motion. Following the derivations from Yang et al. (2009) without applying the exponential smoothing, the variance of blur $\sigma$ after $n$ frames of reprojection is:

$$\sigma_v^2 = n \cdot \frac{v_x(1 - v_x) + v_y(1 - v_y)}{2}, \tag{4}$$

where $v_x$ and $v_y$ are the $x$ and $y$ components of the fractional velocity $\mathbf{v}$. This important result implies that when motion speed is fixed or follows a fixed distribution, then the blur kernel size grows linearly with the increasing number of reprojected frames. This is particularly useful for estimating the worst case amount of blur by simply fixing both $v_x$ and $v_y$ at 0.5. If the user wants to limit the blurring artifact from noticeable, the refresh period $n$ should be set according to Equation 4 such that the blur is less than the inverse of the signal frequency bound.

On the other hand, Sitthi-amorn et al. (2008b) introduce an empirical model that measures the reprojection error by assuming general cases of motion types and correlations of the cached value with the final color. The model is a generalization of the observed fact that in many pixel shaders and representative motion types, the error of the final color decays at an exponential rate with increasing frames of repeated reprojections. Here the error is measured as $L^2-$distance between the color produced by the original shader and one modified to use the RRC. The three factors contributing to this error: the correlation between inputs across consecutive frames, the repeatedly resampling error and the probability of a cache hit (hit-rate), all approximately follow an exponential decreasing rate. Based on these observations, a parametric function $\epsilon$ is proposed for estimating the error:

$$\epsilon(n) = \alpha(1 - e^{-\lambda(n-1)}) \tag{5}$$

Note that $\epsilon(1) = 0$ as required, since this corresponds to refreshing the cache at each frame. This model has two parameters, $\alpha$ for the scale of the error, and $\lambda$ for the exponential decreasing rate of error accumulation. Unfortunately these two parameters are not fixed and require to be fitted using several profiling test for each combination of shader, motion type or cached component (node), as described by Sitthi-amorn et al. (2008b). However, even when uncalibrated, this equation gives intuition of how the rendering quality degrades with caching over more frames – the exponential increase of error. This together with the exponential decrease of render time are taken into account in setting the optimal $n$ for caching.

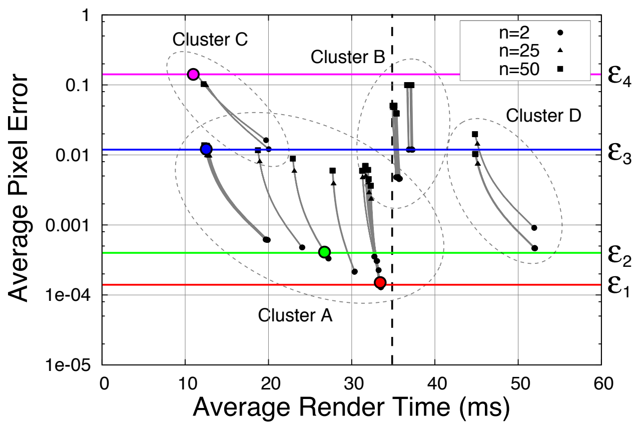
There are occasions when the error does not grow as much (i.e. $\alpha$ and $\lambda$ are small). For example, when the cached signal is of relatively low frequency, the resampling blur will have much lower impact on the error, for reasons we explained before. Similarly, error in the cached component can have only weak effect the final color output for some shaders. It is helpful to keep these cases in mind in selecting the ideal component and shader for acceleration.

### 3.4.3 Quality–Speed Tradeoff

We have shown that for a fixed shader and cached component, the predicted pixel error $\epsilon(n)$ and the frame time $T(n)$ are all functions of the refresh period $n$. It is then possible to analysis the tradeoff between the two by varying $n$ and plotting the trajectory of $\epsilon$ and $T$. The upper graph in Figure 11 shows such tradeoff curves of caching different components in a Perlin noise marble shader, which is also used earlier in Section 3.4.1. The shader combines a marble-like albedo, modeled as five octaves of a 3D Perlin noise function, with a simple Blinn-Phong specular layer. Each curve represents the tradeoff of caching a certain component (intermediate value) in the shader, with varying $n$.

The curves tend to organize into four clusters. Cluster A caches calculations inside the noise function, including its final value. These curves appear near the lower left hand region, corresponding to cached components that leads to greater reduction in render time for less error. Cluster B, on the other hand, represents components that are relatively inexpensive to compute and depend strongly on view and light position. These may include diffuse and specular lighting component and appear near the upper right region, indicating large error and little performance gain. Cluster C represents caching the combination of almost all the components (e.g., the final color). They appear near the upper left region and offers the greatest speed up with large error. Finally, Cluster D represents caching one instance of a value used multiple times in the shader, causing the value to be computed twice in both the cache pass (second pass) and the final color pass (third pass). These nodes introduce wasted computations and are not suitable for caching either.

Within each curve, larger $n$ gives better performance gain but also introduces more shading error. As an example, we set four pixel error thresholds $\epsilon_1$ to $\epsilon_4$ and use the automatic system described by Sitthi-amorn et al. (2008b) to choose the best component for caching and the corresponding $n$. The results visualizing the selected nodes are also shown in Figure 11. Notice that caching too little computation does not lead to satisfying speed gain ($\epsilon_1$, $\epsilon_2$), whereas caching all the computation generates significant visible



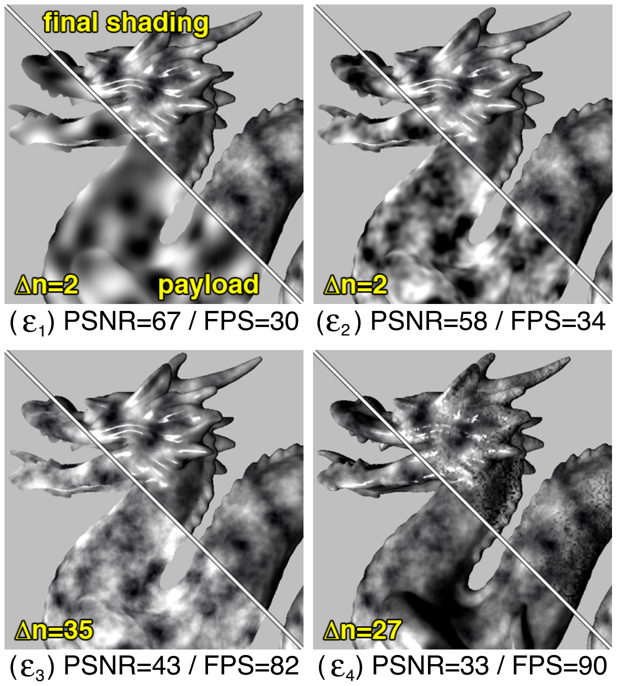


**Figure 11:** *Error/performance paths and four error-performance tradeoff selections for the Perlin Noise-Dragon scene. The graph at top plots the predicted average pixel error vs. render time over a range of refresh periods for every cacheable component in the shader. The images below visualize the best reprojection tradeoff at four different error thresholds indicated in the graph. Each image displays the cache payload and chosen refresh period, along with the final shading and measured frame rate. The original shader runs at 29 FPS as indicated by the dashed line.*

error ($\epsilon_4$). Caching all the noise computation with a relatively large $n$ ($\epsilon_3$) leads to satisfying result in both speed and quality.

### 3.5 Shader Acceleration Examples

In addition to the Perlin-noise dragon shader, we show two more examples here of directly accelerating general pixel shading tasks using the RRC. The first shader is a *Trashcan* environmental reflection shader from ATI's *Toyshop* demo, which combines a simple base geometry with a high-resolution normal map and environment map to reproduce the appearance of a shiny trashcan. It reconstructs the surface color from 25 samples of an environment map combined using a Gaussian kernel. These 25 samples are evaluated along a $5 \times 5$ grid of normal directions computed from the normal map. The result is gamma corrected and finally displayed. In this example, we found that caching the sum of 24 samples of the possible 25 gives the most effective speed up without introducing too much visible artifacts (see Figure 12(left) for a comparison). In other words, the modified shader evaluates 24 samples every fourth frame (on average) and evaluates the single sample with the greatest reconstruction weight at every frame. Indeed, this shader is not particularly suited for using TC to accelerate, because all of the calculations depend strongly on the camera position and cached values quickly become stale. Nevertheless, RRC provides a $2.1\times$ performance improvement at an acceptable level of error.

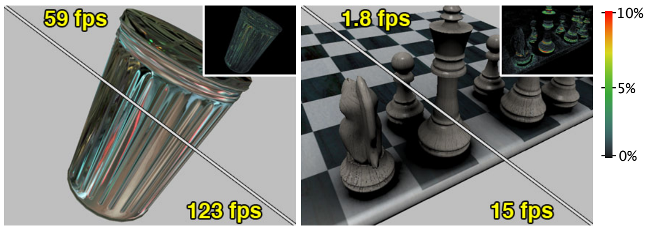


**Figure 12:** *Additional examples of shading acceleration using RRC. Each image compares (top) an input pixel shader to (bottom) a version modified to cache some partial shading computations over consecutive frames. The shading error after applying the cache is illustrated in the inset images.*

The second shader computes an approximate object space ambient occlusion at each pixel for a chessboard scene with the king piece moving and the rest pieces static. The basic idea is to approximate the scene geometry as a collection of discs which are organized in a hierarchical data structure and stored as a texture. As each pixel is shaded, this data structure is traversed to compute the percentage of the hemisphere that is occluded. This calculation is combined with a diffuse texture and a Blinn-Phong specular layer to produce the final color. In this particular scene, the ambient occlusion calculation is carried out by summing the contribution of the king chess piece separately from the other pieces. We found that caching the portion of the ambient occlusion calculation that accounts for only the static pieces gives the best result. In other words, the contribution of the moving king and the remaining shading are recomputed at every frame. This provides a $8\times$ speed-up for a marginal level of error and is demonstrated in Figure 12. Caching more computations such as the entire ambient occlusion calculation will lead to visible error in the result although the speed-up factor is also larger ($15\times$ or more).

## 4 Applications and Extensions of Reverse Reprojection

The reverse reprojection cache is demonstrated to provide impressive acceleration results for a number of common pixel shaders. Next we will show that by simple adaptation and extension, it can also be applied in accelerating a wider range of shading effects, including multi-pass rendering effects, antialiased procedural textures, and global illumination effects that are achieved by Monte-Carlo integration.

### 4.1 Multipass Rendering Effects

There are a number of effects that require rendering a set of images with similar viewpoints. Traditionally the images are rendered separately without considering similarity or correlation within these rendered frames. By applying reprojection based data reuse, we are able to save a significant amount of pixel computation during shad-

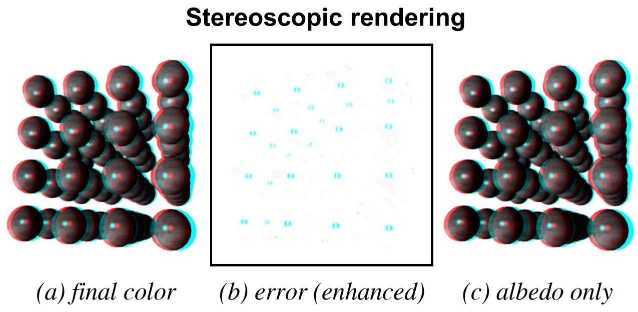


**Figure 13:** *Accelerating stereoscopic rendering using RRC (Courtesy of Diego Nehab). Caching (a) the final color leads to (b) visual errors near specular highlights. These errors can be eliminated by (c) caching only the surface albedo and recomputing the specular contribution at each frame.*

ing mutually visible regions. Technically speaking certain cases of this such as stereoscopic or depth of field cannot be classified as temporal coherence exploitation, since the multiple frames are generated for different view points rather than different time steps. However, the technique behind them are the same.

### 4.1.1 Stereoscopic Rendering

Rendering stereo scenes requires generating two images for two different views. These two images are then encoded and sent to a display. For example, when rendering anaglyph stereo images, we render the left eye view into the red channel, and the right eye view into the green and blue channels. Using glasses with color filters, each eye is exposed to the appropriate view, and the images appear to have depths. Similar idea applies to other stereoscopic rendering techniques. In order to combine the two different views into a single image, two rendering passes are required for each of the view, and a final pass directs these two images into different channels of the result image.

The proximity of theses views allows us to reuse shading information computed for one view at mutually visible points in the opposing view. This idea was simultaneously introduced by Hasselgren and Akenine-Möller (2006) targeting a specially augmented multi-view rasterization architecture, and Nehab et al. (2006) in a simplified form for common hardware. As in normal stereo rendering, we proceed in two passes. On the first pass, we render the right eye view, caching the results. On the second pass, we render the scene using the left eye camera parameters, and perform one cache lookup per pixel. The hit shader simply copies the value read from the right eye. The miss shader computes the pixel color from scratch. Finally, we composite the results of the first and second passes. The early-z culling technique can also be used to improve branching efficiency in this case.

The data reuse between different views can lead to substantial performance gains if the pixel shading cost dominates the rendering process. It should be noted that although the two views may not be exactly the same due to view-dependent effects, artifacts are rarely distracting. Furthermore, if high-precision results are required, it is usually possible to cache only the expensive view-independent information, and add view-dependent components after cache lookup. This is similar to the normal workflow described in Section 3.3.

Figure 13a shows the result of caching the final color in rendering an anaglyph stereo image, containing a scene with 2k triangles and procedurally generated noise texture. The scene also contains a specular highlight view-dependent effect, which is common among many materials. The result is generally acceptable but there are some errors visible when comparing against the groundtruth (Figure 13b). If we do not cache the highlight, i.e. recompute it after cache lookup, the error is completely eliminated. We observe a 57% improvement in frame rate with caching the final color, and 40% with caching the view-independent component.

### 4.1.2 Motion Blur

The motion blur effect can be simulated using temporal supersampling either in fixed time step or using stochastic sampling. The most general technique on graphics hardware is to divide the frame time into multiple smaller time steps, and render the scene at each time step into an accumulation buffer (Haeberli and Akeley, 1990). However, when directly implemented, this approach can be overly slow. Fortunately, spatio-temporal coherence within time samples allows us to use reprojection to speed up the rendering process. Similar ideas has been explored by Chen and Williams (1993) and Havran et al. (2003b) respectively for image based rendering and ray-tracing applications.

We accelerate the accumulation buffer approach using the RRC as follows. For all the intermediate frames within a output frame time interval, we only fully render the first intermediate frame (both into the accumulation buffer and the cache). For the remaining intermediate frames, we perform regular cache look up using reprojection and attempt to reuse the value. Only when cache miss do we compute the pixel color from scratch. Since all the value are directly retrieved from the first frame, there is no repeated interpolation error as in the general task. The shading is also expected not to change significantly because of the relatively small time step. Therefore, no refresh is needed and the cache hit patterns are likely to be coherent across the image. This makes it efficient to use the 1-pass algorithm, which also reduces the overhead of geometry processing when using the RRC.

For this application, the quality loss is usually trivial and not perceptible, given that the final frame is an average of many time samples. On the contrary, since the rendering process becomes much faster, more time samples can be used. Figure 14 shows equal-time comparisons of the results for the brute-force and the RRC accelerated results on a synthetic scene with motion blur effects. The model has 2.5K triangles and uses Perlin noise for albedo. RRC allows us to double the number of time samples and thus achieve a much smoother and convincing result.

### 4.1.3 Depth of Field Effects

The depth of field effect can be simulated using the method very similar to motion blur techniques described before (Also refer to (Haeberli and Akeley, 1990)). The only difference is that individual lens samples, instead of time samples, are accumulated. Sharp images are rendered for slightly different camera positions that sample the area of the aperture. The sample pattern can be random (such as a Poisson disk pattern) or stratified. If enough samples are generated, the averaged image produces an appropriate depth of field effect. Similar to motion blur, the lens samples are usually very close to each other. Therefore TC can be exploited for acceleration using the same technique as described in Section 4.1.2.

Again in Figure 14 we show equal-time comparisons for this application using the same scene. Since the depth of field effect requires a 2D sampling, the undersampling artifacts exhibited in the brute-force method are worse than the case of motion blur and can be a ghosting distraction to the viewer. With the cached method, the number of samples is more than doubled, producing acceptable and convincing results at reasonable framerates.

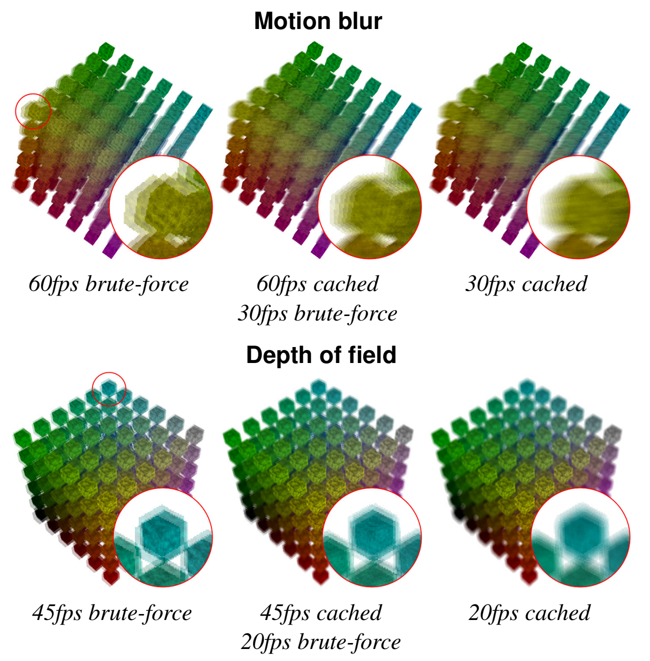


**Figure 14:** *Equal time/quality comparisons between bruteforced methods for rendering motion blur and depth of field effects and accelerated techniques using the RRC (Courtesy of Diego Nehab). Left: At high frame rates, brute-force methods may undersample camera locations and lead to unconvincing results. Middle: Our caching technique lowers the cost of a single pass, allowing the accumulation of more samples and thus smoother effects at comparable frame rates. Right: Results obtained with cache-based methods at equal frame rates.*

## 4.2 Amortized Supersampling

A number of shading techniques require each pixel to compute a weighted average of a number of spatial samples so that an Monte-Carlo integral is obtained. This is often expensive to compute but is crucial for high-quality rendering effects, such as spatial anti-aliasing, soft shadow and global illumination approximations. We describe a method for reprojecting and reusing previously computed samples to help reducing the average computational cost and improving quality. This section also provides a theory background for some of the later sections.

### 4.2.1 Theory

To amortize the cost of sampling over multiple frames, we need to reuse shading results that are computed in previous frames. For instance, we can use a moving average over the past $n$ estimates for a given surface point. The degree of variance reduction in the Monte-Carlo integral is directly related Unfortunately, storing the past estimates in separate cache buffers leads to a serious disadvantage of requiring keeping and reprojecting $n$ cache entries for each pixel. To make this scheme practical, we need to combine the multiple cached values of the same surface point.

A direct approach for achieving this is to introduce a recursive exponential smoothing filter for combining the results from different frames. Let us first assume that in each frame $t$ we only compute one sample $s_t(\mathbf{p})$ for each pixel $\mathbf{p}$. The sample position is determined randomly or quasi-randomly depending on the integration scheme we are using. We keep a running estimate of the integral value in the cache $f_t$. The running estimate is updated in every frame after the new sample is obtained, according to the following equation.

$$f_t(\mathbf{p}) \leftarrow (\alpha)s_t(\mathbf{p}) + (1-\alpha)f_{t\text{-}1}\big(\pi_{t\text{-}1}(\mathbf{p})\big). \tag{6}$$

Note that by expanding this recursive formulation, the running estimate becomes a weighted sum of all the previous samples for the same surface point. The weight of a single sample decreases exponentially in time, and the smoothing factor $\alpha$ regulates the tradeoff between variance reduction and responsiveness to changes in the scene. For example, a small value of $\alpha$ reduces the variance in the estimate and therefore produces a smoother result, but introduces more lag if the shading changes between frames. If reprojection fails at any pixel (i.e. $\pi_{t\text{-}1}(\mathbf{p}) = \varnothing$), then $\alpha$ is locally reset to 1 to give full weight to the current sample.

In a precise form, the degree of variance reduction is $\frac{\alpha}{2-\alpha}$. For example, choosing a value of $\alpha = 2/5$ reduces the variance to $1/4$ the original. This is roughly equal to combining 4 previous samples with equal weights, although in reality 10 past frames affects the result where earlier samples are lost in 8-bits of framebuffer precision. It is noted that the result obtained using the recursive filter is biased by favoring the later samples. This will not result in a shifted mean of the estimate if the sampling sequence is uniformly random, but instead is important for achieving better quality in the presence of signal changes and resampling error.

### 4.2.2 Achieving Subpixel Accuracy: Shader Antialiasing

Due to resampling error, simple amortized supersampling described above cannot be directly used in applications that require subpixel accuracy, such as pixel antialiasing. Yang et al. (2009) propose a scheme to extend the technique to approximate subpixel accuracy. Here we give a brief description of the underlying ideas. The theoretical details can be found in the original paper.

The major application of the described technique is to antialias procedural shaders. This includes procedural materials and complex shading functions. Unlike prefiltered textures, procedurally defined signals are not usually bandlimited (Ebert et al., 2003), and producing a bandlimited version of a procedural shader is a difficult and ad-hoc process (Apodaca and Gritz, 2000). Generally supersampling is often used for this task. However it can be prohibitively expensive to evaluate the shader multiple times per pixel. Fortunately, in many cases the underlying signals are constant or vary slowly over time, so that previously computed samples can be reprojected and reused, as described in the general case above. The major issue to solve here is to reduce the resampling error to prevent the accumulated cached samples from being distorted.

From the reprojected error estimator Equation 4 we get that the bilinear reconstruction reaches the largest error when $v_x$ or $v_y$ is close to 0.5. An effective method to reduce $v_x$ and $v_y$ is to apply a higher resolution cache buffer, for example, $2 \times 2$ of the screen size. However, maintaining such a buffer and reproject it every frame introduces a significant overhead to the acceleration process. We can instead store the $2 \times 2$ quadrant samples of each pixel into four subpixel buffers $\{b_k\}$, $k \in \{0, 1, 2, 3\}$ in an interleaved way. Each subpixel buffer are screen sized and manages one quadrant of a pixel. These subpixel buffers are updated in a round-robin fashion, i.e. only one per frame.

Reconstructing a subpixel value from the four subpixel buffers involves more work. Note that in the absence of scene motion, these four subpixel buffers effectively form a higher-resolution framebuffer. However, under scene motion, the subpixel samples computed in earlier frames reproject to offset locations. Conceptually, we forward reproject all the previous samples into the current

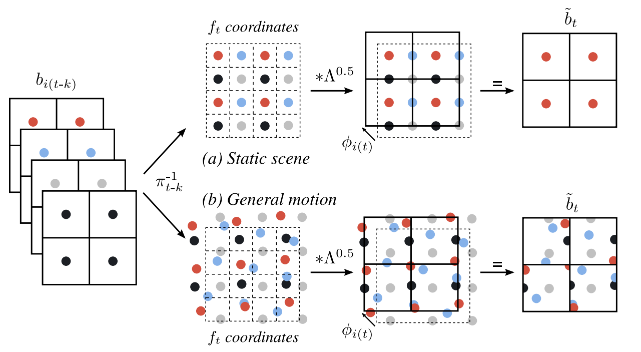


**Figure 15:** *Sampling from multiple subpixel buffers. To properly reconstruct the quadrant value, Yang et al. (2009) use nonuniform blending weights defined by a tent function centered on the quadrant being updated. (a) In the absence of local motion, only the correct pixel has non-zero weight in the tent, so no resampling blur is introduced; (b) For a moving scene, the samples are weighted using the tent function, and higher weights are given to samples closer to the desired quadrant center to limit the amount of blur.*

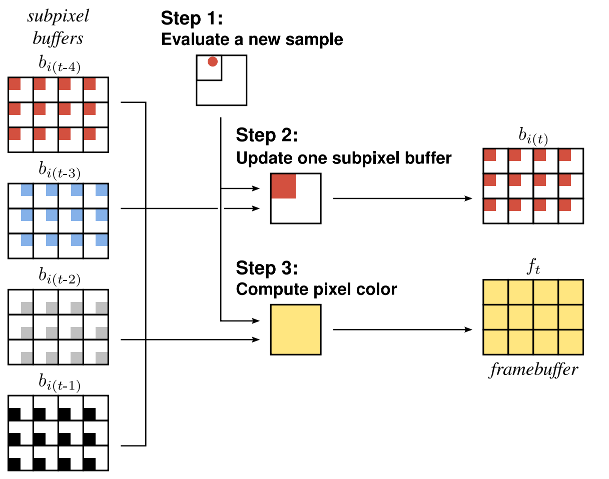


**Figure 16:** *The main steps of the fragment shader algorithm of amortized supersampling using subpixel buffers. All steps are performed together in the main rendering pass.*

frame, as indicated in Figure 15. Since the reprojected samples may form an arbitrary pattern, to reconstruct the quadrant value, we apply a tent weighting function that encompasses the current quadrant and compute the weighted sum of all the samples that fall under the support of this function. This effectively helps to reduce the contribution of distant samples and limit the amount of blur introduced.

At each frame, we update one of the subpixel buffers by computing one new sample per pixel, and combine it with the history value using the exponential smoothing filter (Equation 6). We then compute the final color of every pixel by reconstructing from the subpixel buffers. This uses the similar technique for reconstructing subpixels, except that the tent now spans the entire pixel. This process is illustrated in Figure 16. Note that in the actual implementation, we do not use forward reprojection. Instead, we apply the RRC method and reproject the tent back to the previous subpixel buffers. This has proven to be an reasonable approximation that makes the reconstruction step more efficient on graphics hardware.

Even when using the subpixel buffers, the reconstruction error may still exceed the user's tolerance, especially when small $\alpha$ is applied.

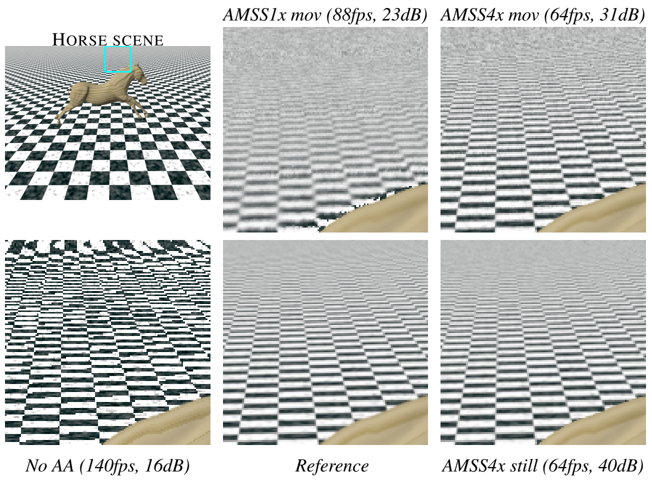


**Figure 17:** *Comparison between no antialiasing, amortized supersampling (with screen-size cache, and with $2 \times 2$ subpixel buffers), and the ground-truth reference result for a horse-checkerboard scene. The 4x still image approaches the quality of the reference result, whereas the motion result provides an acceptable approximation without overblurring.*

In addition, the antialiased signal may occasionally vary temporally due to, for example, light and view-dependent effects. Yang et al. (2009) propose methods to estimate the reconstruction error, as well as the signal change in real-time, and limit $\alpha$ accordingly such that a minimum amount of refresh is guaranteed. The reconstruction error is estimated by deriving an empirical relationship between the fractional pixel velocity $\mathbf{v}$, $\alpha$ and the error, similar to the purpose of Equation 4. Signal change, on the other hand, is estimated by a smoothed residual between the aliased sample and the history value. The user set thresholds for both errors, and the bounds for $\alpha$ are computed based on the error values.

Figure 17 shows the result of applying amortized supersampling with $2 \times 2$ subpixel buffer on a horse-checkboard scene, which includes an animated wooden horse galloping over a marble checkered floor. The result shows significant improvement over the screen-sized amortized supersampling with only a minor drop of speed. In fact, the PSNR shows that this technique offers better quality when compared to the conventional $4 \times 4$ stratified supersampling, which runs at a six times lower framerate.

## 4.3 Discrete LOD Blending

The idea behind discrete *level-of-detail* (LOD) techniques is to use a set of representations with differing complexities (level-of-detail) for one model and select the most appropriate representation for rendering at runtime. Complexity can for instance vary in the employed materials or shaders or in the amount of triangles used. Due to memory constrains and the effort in creating them only a small number of LODs is being used and therefore switching from one representation to another can lead to noticeable popping artifacts. A theoretical solution would be to switch only when the respective pixel output of two representations is indistinguishable. This so called *late switching* has practical problems. First, it is hard to guarantee equality in pixel output for a given view scenario and lighting without rendering both representations first, which of course defeats the purpose. Second, the idea of switching as late as possible counteracts the potential gain of employing LODs in the first place. In practice switching is done as soon as ”acceptable”.

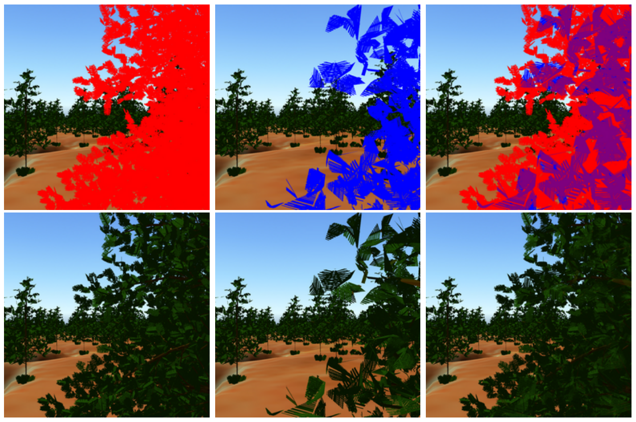


**Figure 18:** *LOD interpolation combines two buffers containing the discrete LODs to create smooth LOD transitions.* First and second column*: buffers;* last column*: combination. The top row shows the two LODs in red and blue respectively.*

A more practical solution to this problem proposed by Giegl and Wimmer (2006) is to replace the hard switch by a transition phase, in which the two representations are alpha blended in screen space.

Apart from other problems, this approach requires that the geometry (and the shaders) of both representations have to be rendered in this transition phase, thereby generating a higher rendering cost than the higher quality level alone would incur. To circumvent this Scherzer and Wimmer (2008) have introduced LOD interpolation (see Figure 18). The idea is that by using TC the two LODs required during an LOD transition can be rendered in *subsequent frames*. Two separate render passes are used to achieve the transition phase between adjacent LOD representations: Pass one renders the scene into an off-screen buffer (called *LOD buffer*). For objects in transition one of the two LOD representations is used and only a certain amount of its fragments are rendered (see Figure 19), depending on were in the transition (i.e., how visible) this object currently is. This is later repeated in the next frame using the other LOD representation and rendering into a second *LOD buffer*. The second pass combines these two *LOD buffers* (one from the current and one from the previous frame) to create the desired smooth transition effect.

To determine the number of fragments to render for a given representation, so-called *visibility textures* are used. Each encodes a visibility threshold function $visTex(\mathbf{p}) \rightarrow [0..1]$ that maps the object-space coordinate (before any animation is applied) of a given fragment $\mathbf{p}$ to the fragments visibility threshold.

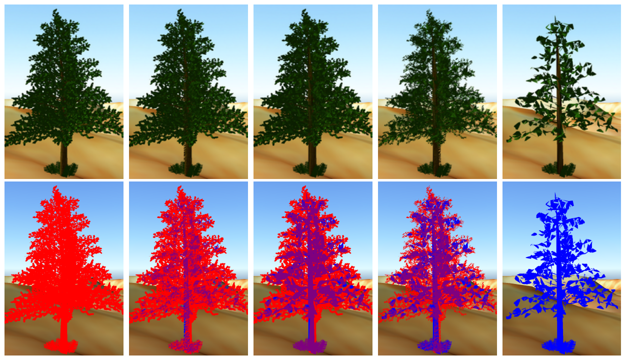
This allows individual fragments to be discarded by comparing the output of this function to an objects visibility threshold $\iota$. $\iota$ encodes were in the transition phase a representation currently is and is given by the transition function depicted in Figure 19

Writing this process as a function gives $discard : R^3 \times [0..1] \rightarrow \{true, false\}$

$$discard : (\mathbf{p}, \iota) \mapsto visTex(\mathbf{p}) < \iota. \qquad (7)$$

Note that even though the visibility function may be continuous, the thresholding operation gives a binary result and therefore no semi-transparent pixels appear, which avoids blending and the costly ordering of fragments.

By using different visibility textures, one can control in which way the individual fragments of a given object become visible. Examples include a uniform noise pattern, a function that decreases from



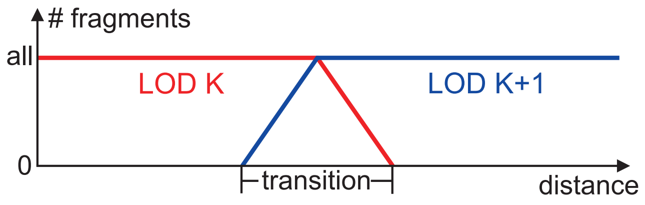


**Figure 19:** *Transition phase from $LOD_k$ to $LOD_{k+1}$:* left:*$LOD_k$;* middle: *midway in the transition all fragments of both LODs are drawn;* right: *$LOD_{k+1}$;* Below: *First $LOD_{k+1}$ is gradually introduced till all its fragments are drawn. Then $LOD_k$ is gradually removed by rendering fewer and fewer fragments. The top two rows show the result of our method and a false color illustration.*

the center outward, or any other function best suited to a given object. This has the effect that the amount and distribution of the visible fragments of an object can be controlled (see Figure 20). Also note that although $visTex$ is given as a 3D function it is often not necessary to store it in a 3D texture, as can be seen by the noise texture example.

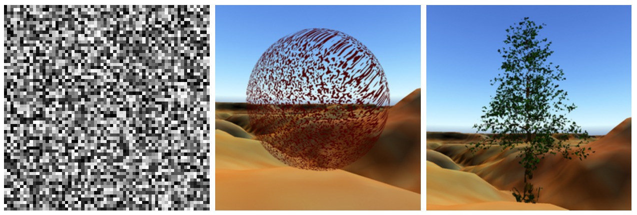


**Figure 20:** *A uniform noise visibility texture (*left*) applied to two different models with visibility $\iota = 0.5$).*

## 4.4 Casting Shadows

Shadows are widely acknowledged to be one of the global lighting effects with the most impact on scene perception. They are perceived as a natural part of a scene and give important cues about the spatial relationship of objects.

Due to its speed and versatility, shadow mapping is one of the most used real-time shadowing approaches. The idea is to first create a depth image of the scene from the point of view of the light source (shadow map). This image encodes the front between lit and unlit parts of the scene. On rendering the scene from the point of view of the camera each fragment is transformed into this space. Here the depth of each transformed camera fragment is compared to the respective depth in the shadow map. If the depth of the camera fragment is nearer it is lit otherwise it is in shadow (See Figure 21).

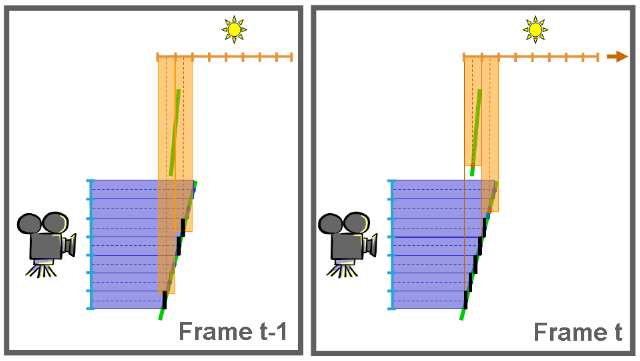


**Figure 21:** *If the rasterization of the shadow map changes (here represented by a right shift), the shadowing results may also change. On the* left *three fragments are in shadow, while on the* right *five fragments are in shadow. This results in flickering or swimming artifacts in animations.*

### 4.4.1 Pixel Correct Shadows

The most concerning visual artifacts of this method originate from aliasing due to undersampling. The cause for undersampling is in turn closely related to rasterization that is used to create the shadow map itself. Rasterization uses regular grid sampling for rasterization of its primitives. Each fragment is centered on one of these samples, but is only correct exactly at its center. If the viewpoint changes from one frame to the next, the regular grid sampling of the new frame is likely to be completely different than the previous one. This frequently results in artifacts, especially noticeable for thin geometry and the undersampled portions of the scene called *temporal aliasing*.

This is especially true for shadow maps. Due to shadow map focusing, a change in the viewpoint from one frame to the next also changes the regular grid sampling of the shadow map. Additionally the rasterized information is not accessed in the original light-space where it was created, but in eye-space, which worsens these artifacts. This frequently results in temporal aliasing artifacts, mainly flickering (See Figure 21).

The main idea in Scherzer et al. (2007) is to jitter the view port of the shadow map differently in each frame and to combine the results over several frames, leading to a higher effective resolution.

Exponential smoothing as described above is employed here on the shadow map tests $s_t[\mathbf{p}]$. This serves a dual purpose. On the one hand temporal aliasing can be reduced by using a big smoothing factor. On the other hand, the shadow quality can actually be made to converge to a pixel-perfect result by optimizing the choice of the smoothing factor.

The smoothing factor allows balancing fast adaption on changing input parameters against temporal noise. With a bigger smoothing factor, the result depends more on the shadow results result from the current frame and less on older frames and vice versa. The smoothing factor is now determined according to the *confidence* of the shadow lookup. The confidence is defined to be higher if the lookup falls near the center of a shadow map texel, since only near the center of shadow map texels it is very likely that the sample actually represents the scene geometry (see Figure 23 and 22). In

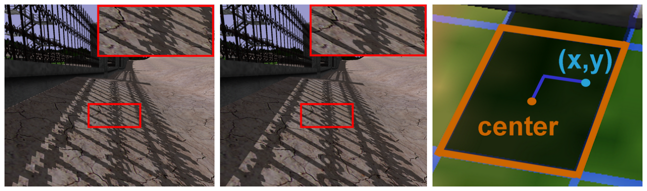


**Figure 22:** *LiSPSM* (left) *gives good results for a shadow map resolution of* $1024^2$ *and a view port of* $1680 \times 1050$, *but temporal reprojection* (middle) *can still give superior results because it uses shadow test confidence, defined by the maximum norm of shadow map texel center and current pixel* (right).

the paper the maximum norm of the current pixel $\mathbf{p}$ and the shadow map texel center $\mathbf{c}$ is used to account for this

$$\mathsf{conf} = \left(1 - \max\left(\left|\mathbf{p}_x - \mathbf{c}_x\right|, \left|\mathbf{p}_y - \mathbf{c}_y\right|\right) \cdot 2\right)^m, \qquad (8)$$

but other norms could be used as well. The parameter $m$ defines how strict this confidence is applied. $m < 4$ results in fast updates were most shadow map lookups of the current frame have a big weight and the resulting shadow has noisy edges. $m > 12$ results in accurate but slow updates were most lookups from the current frame have small weight. The authors found out that $m$ should be balanced with camera movement. When the camera moves fast $m$ can be small because noise at the shadow borders is not noticed (the human eye integrates the motion anyway). Only for a slowly moving camera or a still image are higher values of $m$ necessary.

This confidence can now be directly used in the exponential smoothing formula

$$f_t[\mathbf{p}] \leftarrow (\mathsf{conf})s_t[\mathbf{p}] + (1 - \mathsf{conf})f_{t\text{-}1}\big(\pi_{t\text{-}1}(\mathbf{p})\big). \qquad (9)$$

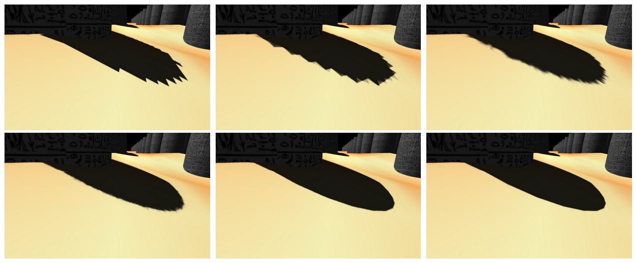


**Figure 23:** *Shadow adaption over time of an undersampled uniform shadow map after 0* (top-left), *1* (top-middle), *10* (top-right), *20* (bottom-left), *30* (bottom-middle) *and 60* (bottom-right) *frames.*

### 4.4.2 Soft Shadows

In reality most light sources are area light sources and hence most shadows exhibit soft borders. *Light source sampling* introduced by Heckbert and Herf (1997) creates a shadow map for every sample (each on a different position on the light source) and calculates the average (= soft shadow) of the shadow map test results $s_i$ for each pixel (see Figure 24). Therefore, the soft shadow result from $n$ shadow maps for a given pixel $\mathbf{p}$ can be calculated by

$$\psi_n(\mathbf{p}) = \frac{1}{n}\sum_{i=1}^{n} s_i(\mathbf{p}). \qquad (10)$$

The primary problem here is that the number of samples (and therefore shadow maps) to produce smooth penumbrae is huge and therefore this approach is slow. Typical methods for real-time applications approximate an area light by a point light located at its center and use heuristics to estimate penumbrae, which leads to soft shadows that are not physically correct. Here overlapping occluders can lead to unnatural looking shadow edges, or large penumbrae can cause single sample soft shadow approaches to either break down or become very slow

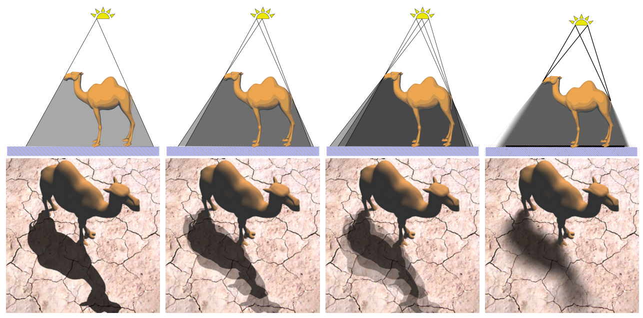


**Figure 24:** *Light sampling with 1, 2, 3 and 256 shadow maps* (left *to* right).

The main idea of Scherzer et al. (2009) is to formulate light source area sampling in an iterative manner, evaluating only a single shadow map per frame. Reformulating Equation 10 gives

$$\psi(\mathbf{p}) = \frac{s(\mathbf{p}) + \Sigma(\mathbf{p})}{n(\mathbf{p}) + 1} \qquad \Sigma(\mathbf{p}) = \sum_{i=1}^{n(\mathbf{p})} s_i(\mathbf{p}). \tag{11}$$

were $s(\mathbf{p})$ is the hard shadow map result for the current frame and pixel and $n(\mathbf{p})$ is the number of shadow maps evaluated until the last frame for this pixel. Note that now $n$ depends on the current pixel because dependent on how long this pixel has been visible, a different number of shadow maps may have been evaluated for this pixel. Calculation of this formula is straight forward if $n(\mathbf{p})$ and $\Sigma(\mathbf{p})$ are stored in a buffer (another instance of the RRC). With this approach, the soft shadow improves from frame to frame and converges to the true soft shadow result if pixels stay visible ”long enough” (see Figure 25, upper row).

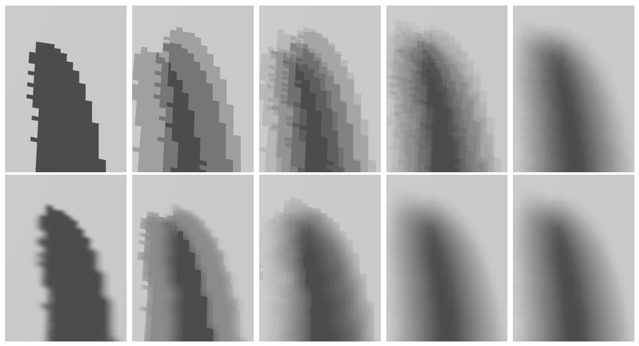
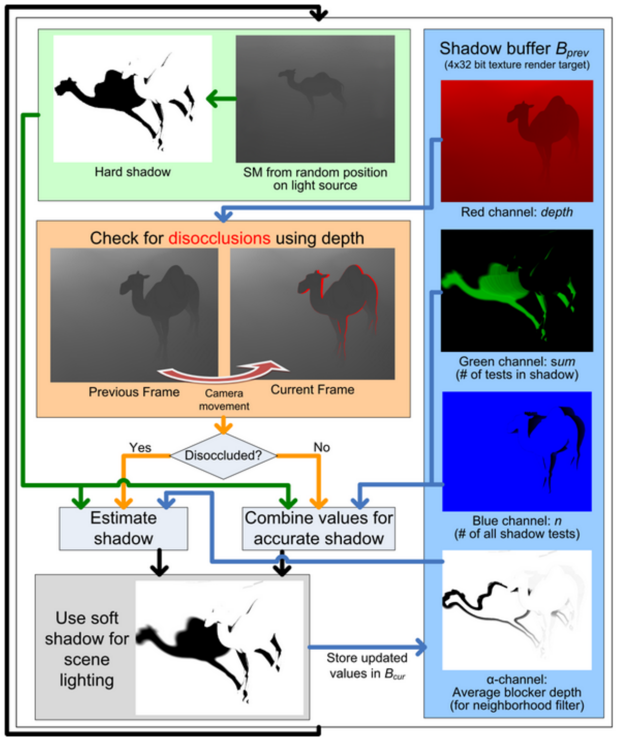


**Figure 25:** *Convergence after 1,3,7,20 and 256 frames;* upper row*: sampling of the light source one sample per frame;* lower row*: soft shadows with TC.*

In praxis this can result in temporal aliasing for small $n$. Care has to be taken how to manage those cases. When a pixel becomes newly visible and therefore no previous information is available in the RRC, a fast single sample approach (PCSS with a fixed 4x4 kernel) is employed to generate an initial soft shadow estimation for this pixel. For all other $n$ the expected standard error is calculated and if it is above a certain threshold (expected fluctuation in the soft shadow result in consecutive frames) a depth-aware spatial filter is



Hard shadow
SM from random position on light source
Shadow buffer $B_{prev}$
(4x32 bit texture render target)
Red channel: *depth*
Check for disocclusions using depth
Previous Frame
Camera movement
Current Frame
Green channel: *sum* (# of tests in shadow)
Yes
Disoccluded?
No
Estimate shadow
Combine values for accurate shadow
Blue channel: *n* (# of all shadow tests)
Use soft shadow for scene lighting
Store updated values in $B_{cur}$
α-channel: Average blocker depth (for neighborhood filter)

**Figure 26:** *Structure of the soft shadows with TC algorithm.*

employed to take information from the neighborhood in the RRC into account (see Figure 26). This approach largely avoids temporal aliasing and can be nearly as fast as hard shadow mapping if all pixels have been visible for some time and the expected standard error is small enough (see Figures 25 and 27).

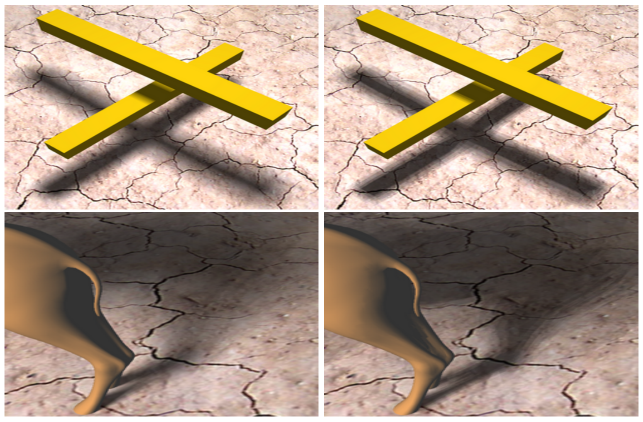


**Figure 27:** Left side*: soft shadows with TC;* right side*: PCSS 16/16; Overlapping occluders* (upper row) *and bands in big penumbras* (lower row) *are known problem cases for single sample approaches.*

## 4.5 Global Illumination

### 4.5.1 Screen-Space Ambient Occlusion

Ambient occlusion (Cook and Torrance, 1981) is a cheap but effective approximation of global illumination which shades a pixel with the percentage of the hemisphere that is blocked. It can be seen as the diffuse illumination due to the sky (Landis, 2002). Ambient occlusion of a surface point $\mathbf{p}$ with normal $\mathbf{n_p}$ is computed as

$$\mathrm{AO}(\mathbf{p}, \mathbf{n_p}) = \frac{1}{\pi} \int_{\mathbf{\Omega}} (\mathbf{n_p} \cdot \boldsymbol{\omega}) \, V(\mathbf{p}, \boldsymbol{\omega}) \, d\boldsymbol{\omega}, \tag{12}$$

where $\mathbf{\Omega}$ denotes all directions on the hemisphere and $V$ is the (inverse) binary visibility function, with $V(\mathbf{p}, \boldsymbol{\omega}) = 1$ if the visibility in this direction is blocked by an obstacle, 0 otherwise.

Screen-space ambient occlusion (SSAO) methods, as first introduced by Mittring (2007) sample the frame buffer as a discretization of the scene geometry. We assume that any SSAO method can be written as an average over contributions $C$ which depend on a series of samples $\mathbf{s_i}$:

$$SSAO_n(\mathbf{p}) = \frac{1}{n} \sum_{i=1}^{n} C(\mathbf{p}, \mathbf{s_i}) \tag{13}$$

In order to approximate equation 12 using Monte Carlo integration, the contribution function for SSAO is usually

$$C(\mathbf{p}, \mathbf{s_i}) = V(\mathbf{p}, \mathbf{s_i}) \, max(\, \cos(\mathbf{s_i} - \mathbf{p}, \mathbf{n_p}), 0). \tag{14}$$

In contrast to equation 12, directions have been substituted by actual sample points around $\mathbf{p}$, and thus $V(\mathbf{p}, \mathbf{s_i})$ is now a binary visibility function that gives 0 if $\mathbf{s_i}$ is visible from $\mathbf{p}$ and 1 otherwise. A depth test of the current sample $\mathbf{s_i}$ determines its visibility $V$. Several variants of SSAO have been proposed since (Fox and Compton, 2008; Bavoil et al., 2008; Szirmay-Kalos et al., 2010).

The quality of SSAO can be significantly improved with TC, as was already shown in commercial games (Smedberg and Wright, 2009). This is due to the beneficial properties of SSAO, i.e., its invariance from light source and view direction and the local support of the SSAO kernel. Previously computed SSAO samples can be cached and reused with reverse reprojection. In the following we will discuss the method of Mattausch et al. (2010), who focus on improving the SSAO quality, and optionally allow for some performance optimization by using less samples in sufficiently converged regions.

**Refining the SSAO solution over time** First we discuss the SSAO accumulation scheme. In frame $f$, a new contribution $C_t$ is calculated from $k$ new samples:

$$C_t(\mathbf{p}) = \frac{1}{k} \sum_{i=j_t(\mathbf{p})+1}^{j_t(\mathbf{p})+k} C(\mathbf{p}, \mathbf{s_i}), \tag{15}$$

where $j_f(\mathbf{p})$ counts the number of unique samples that have already been used in this solution. We combine the new contribution with the previously computed solution:

$$\begin{aligned} SSAO_t(\mathbf{p}) &= \frac{w_{t-1}(\mathbf{p}_{t-1}) SSAO_{t-1}(\mathbf{p}_{t-1}) + kC_t(\mathbf{p})}{w_{t-1}(\mathbf{p}) + k} && (16) \\ w_t(\mathbf{p}) &= min(w_{t-1}(\mathbf{p}_{t-1}) + k, w_{max}). && (17) \end{aligned}$$

The weight $w_{t-1}$ represents the number of samples that have already been accumulated in the solution, until $w_{max}$ has been reached. This predefined maximum ensures that the influence of older contributions decay over time (note that the solution converges very quickly) and controls the refresh rate.

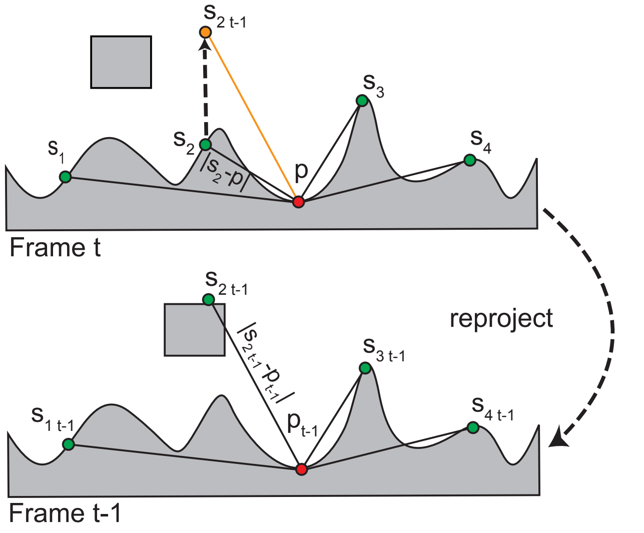


**Figure 28:** *The distance of* $\mathbf{p}$ *to sample point* $\mathbf{s}_2$ *in the current frame differs significantly from the distance of* $\mathbf{p}_{t-1}$ *to* $\mathbf{s}_{2_{t-1}}$ *in the previous frame, hence we assume that a local change of geometry occurred, which affects the shading of* $\mathbf{p}$.

**Detecting changes** Special attention must be paid to the detection of cache misses (i.e., pixels with an invalid SSAO solution) due to dynamic changes. In order to benefit from TC in fully dynamic scenes, we have to efficiently detect and handle such cache misses. A cached value of a pixel is invalid if either one of the following three conditions has occurred: 1) a disocclusion of the current pixel, 2) the pixel was previously outside the frame buffer, 3) a change in the *sample neighborhood* of the pixel. Case 1) and case 2) can be handled like conventional cache misses as described previously in Section 3.2.2. However, we additionally have to check for case 3), because the shading of the current pixel can be affected by nearby moving objects within the sampling radius. Consider for example a scenario where a box is lifted from the floor. The SSAO values of pixels in the contact shadow area surrounding the box change, even if there is no disocclusion of the pixel itself.

Checking the complete neighborhood of a pixel would be prohibitively expensive, and therefore we use sampling. Fortunately we already have a set of samples, namely the ones used for AO generation. That means that we can effectively use our AO sampling kernel for two purposes: for computing the current contribution $C_t(\mathbf{p})$ and to test for validity. A measure for validity of a sample $\mathbf{s}_i$ for shading a pixel $\mathbf{p}$ can be estimated by computing the change in relative positions of a sample and pixel (as illustrated in Figure 28):

$$\delta(\mathbf{s}_i) = ||\mathbf{s}_i - \mathbf{p}| - |\mathbf{s}_{i_{t-1}} - \mathbf{p}_{t-1}||. \tag{18}$$

We only have to use those samples for the neighborhood test that lie in front of the tangent plane of $\mathbf{p}$, since only those samples actually modify the shadow term. Note that we could additionally check if the angle between surface normal and vector to the sample point has changed by a significant amount from one frame to the next.

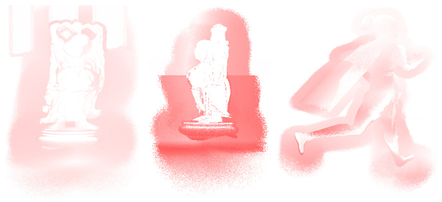


**Figure 29:** *Confidence values computed by our smooth invalidation technique for a rotating object (left), a translating object (middle), and an animated character (right). We use a continuous scale from red (confidence=0) to white (confidence=1).*

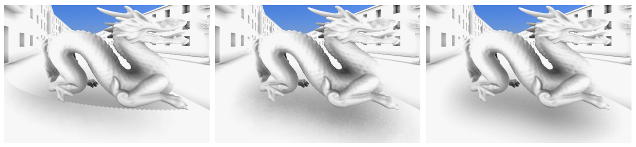


**Figure 30:** *Moving dragon model using no invalidation (left, causing artifacts in the shadow), a high invalidation factor (middle, causing noise), an invalidation factor set to a proper value (right, no artifacts).*

However, this would require to store the surface normal of every pixel in the previous frame.

**Smooth invalidation** Consider for example a slowly deforming surface, where the SSAO will also change slowly. In such a case it is not necessary to fully discard the previous solution. Instead we introduce a new continuous definition of invalidation that takes a measure of change into account. This measure of change is given by $\delta(\mathbf{s}_i)$ at validation sample position $\mathbf{s}_i$, as defined in Equation 18. In particular, we compute a *confidence* value $conf(\mathbf{s}_i)$ between 0 and 1. It expresses the degree to which the previous SSAO solution is still valid:

$$conf(\mathbf{s}_i) = 1 - \frac{1}{1 + S\delta(\mathbf{s}_i)}. \tag{19}$$

The invalidation factor $S$ is a parameter which controls the smoothness of the invalidation. The overall confidence in the previous SSAO solution is given by

$$conf(\mathbf{p}) = \min(conf(\mathbf{s}_0), .., conf(\mathbf{s}_k)). \tag{20}$$

This value is used to attenuate the weight $w_t$ given to the previous solution (refer to Equation 17). Figure 29 depicts a visualization of $conf$ for different types of movements. Figure 30 shows the effect of the smooth invalidation factor on a scene with a translational movement. Setting the invalidation factor $S$ to $(15 \leq S \leq 30)$ usually gives good results.

A comparison of conventional SSAO with TSSAO is shown in Figure 31. The noisy appearance of a coarse SSAO solution that uses only a few samples (image a) can be improved with a screen-space spatial discontinuity filter. However, the result of this operation can be quite blurry (image b). As long as there is a sufficient history for a pixel, temporal SSAO (TSSAO) produces smooth but crisp SSAO without depending on heavy post-processing (image c).

Note that for TSSAO, spatial filtering only has to be applied in regions of change. This is done scaling the filter radius with a measure of the inverse convergence $1 - w_n/w_{max}$. The results of the filtering can be further improved by making it convergence-aware, i.e., assigning higher weights to sufficiently converged filter samples.

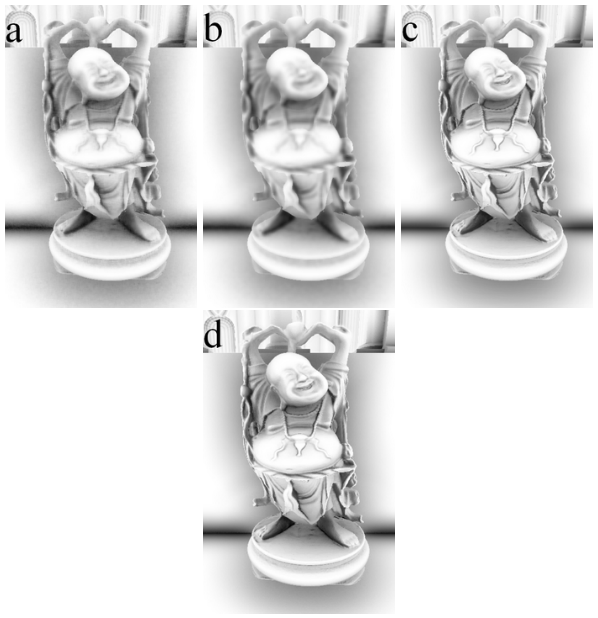


**Figure 31:** *SSAO without TC using 32 samples per pixel with (a) a weak blur, (b) a strong blur (both 23 FPS), (c) temporal SSAO using 8–32 samples (initially 32, 8 in a converged state) (45 FPS). (d) Reference solution using 480 samples (2.5 FPS). The scene has 7M vertices and runs at 62 FPS without SSAO.*

### 4.5.2 Imperfect Shadow Maps

Instant radiosity (Keller, 1997) is a hardware-friendly global illumination method that computes so called virtual point lights (VPLs) along the intersections of a light path with a surface, and uses them for indirect scene illumination. The visibility is resolved by computing a shadow map for each VPL. The shadow map computation is also the main bottle neck of the algorithm, as it requires to sample the scenes many times for a reasonable number of VPLs. This drawback prevents real-time frame rates even when restricted to first-bounce global illumination.

Based on the observation that exact visibility is not required for low-frequency global illumination, Ritschel et al. (2008) proposed to use a coarse point-based scene representation instead. These imperfect shadow maps allow hundreds of shadow map-based visibility queries per frame in interactive time. However, even such a large number of queries are insufficient to avoid typical undersampling artifacts, e.g., resulting in flickering between frames if the VPLs are recomputed.

Knecht et al. (2010) combines the imperfect shadow mapping approach with temporal reprojection. He manages to improve the quality and reduce the before mentioned artifacts (refer to Figure 32). The main problem of using TC for global illumination is the global nature of changes of the lighting conditions and the scene configuration. Knecht chose to use a non-binary threshold similar to the smooth invalidation technique for temporal SSAO (refer to Section 4.5.1). In particular, the confidence into a previous solution is guided by the amount of change of a pixel between the previous and current frame. The following measures are used in order to evaluate a confidence value $conf$:

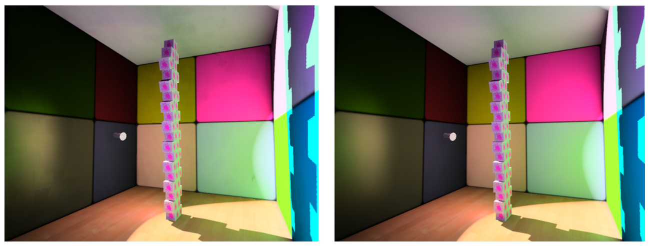


**Figure 32:** *Imperfect shadow maps still show some artifacts with 256 VPLS, which can be smoothed out using TC (Knecht, 2009).*

$$\begin{aligned}\epsilon_{pos} &= ||(x_t - x_{t-1}; y_t - y_{t-1}; d_t - d_{t-1})\mathbf{w_p}|| \\ \epsilon_{norm} &= (1 - n \cdot n_{prev})w_n \\ \epsilon_{ill} &= saturate(||I_t - I_{t-1}||^3)w_i \\ conf &= saturate(1 - \max(\epsilon_{pos}; \epsilon_{norm}; \epsilon_{ill}))c_B \quad (21)\end{aligned}$$

The first equation computes a distance value using screen-space position and depth, the second equation takes the differences in the normals into account, the third the difference in the illumination values. The weights $\mathbf{w_p}$, $w_n$, and $w_i$ are highly scene dependent. The final confidence is computed as the maximum of these measures multiplied with some base confidence $c_B$, and then used as the weight of a standard exponential smoothing operation. Due to the low frequency nature of indirect illumination, the motion blur like artifacts caused by moving light sources and animated objects are not very distracting in the general case.

### 4.6 Spatio-Temporal Upsampling

In many rendering applications, spatial coherence also exists within the shading signal, especially with low-frequency diffuse shading. Herzog et al. (2010) propose a spatio-temporal upsampling technique that exploits both spatial and temporal coherence for shading acceleration. The spatial coherence part follows the geometry-aware upsampling (Yang et al., 2008) method, which computes the entire screen in lower resolution, and then uses a joint-bilateral filter to upsample the shading result to full resolution. The joint-bilateral filter takes normal and depth difference between the low-resolution samples and the desired pixel as the "range" weight, which gives preference to the samples from the same geometry piece during upsampling. This helps to avoid blurring artifacts.

In the technique proposed by Herzog et al. (2010), shading is always computed in lower density than the screen resolution (e.g. $1/4 \times 1/4$). In every frame, $1/16$ samples on the screen are shaded in an interleaved way (Keller and Heidrich, 2001; Segovia et al., 2006). This effectively reduces the shading cost in a spatially consistent manner that facilitates upsampling. The newly shaded samples are upsampled as in Yang et al. (2008), keeping the bilateral weights. Then the previously stored payload (with full resolution) is reprojected to the current frame, and blend with the upsampled new contribution using

$$f_t(\mathbf{p}) = \frac{\tilde{h}(\mathbf{p})\tilde{w}(\mathbf{p}) + w_f w_{t\text{-}1}(\pi_{t\text{-}1}(\mathbf{p}))f_{t\text{-}1}(\pi_{t\text{-}1}(\mathbf{p}))}{\tilde{w}(\mathbf{p}) + w_f w_{t\text{-}1}(\pi_{t\text{-}1}(\mathbf{p}))}, \qquad (22)$$

where $f_t(\mathbf{p})$ is the payload value of pixel $\mathbf{p}$ at frame $t$, $\tilde{h}$ is the upsampled new shading result with summed bilateral weights $\tilde{w}(\mathbf{p})$, $f_{t\text{-}1}(\pi_{t\text{-}1}(\mathbf{p}))$ is the reprojected cache payload value, $w_f$ is the temporal fading factor, and $w_{t\text{-}1}(\pi_{t\text{-}1}(\mathbf{p}))$ is the spatio-temporal weight, storing a confidence value of $f_{t\text{-}1}(\pi_{t\text{-}1}(\mathbf{p}))$.

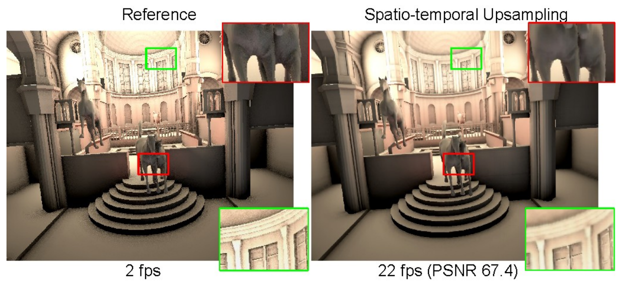


**Figure 33:** *Spatio-temporal upsampling applied to a fully dynamic scene with complex indirect shading and ambient occlusion computed in real time (Herzog et al., 2010).*

This equation is essentially similar to the exponential smoothing equation (Equation 6) that we see previously in many other applications. However, there are multiple factors that adjust the weights. We first notice that with the bilateral weight $\tilde{w}(\mathbf{p})$, the pixels that are close to the recomputed pixels receive more weights than its neighbors. This leads to a smooth update of the recomputed pixels. The spatio-temporal weight $w_{t\text{-}1}(\pi_{t\text{-}1}(\mathbf{p}))$ on the other hand represents how much information has been accumulated in the cache. It can be viewed as a confidence of the cache payload, and is recomputed every frame after the payload refresh according to the blending weights in the equation. Finally, the temporal fading factor controls the exponential smoothing rate and is dynamically adjusted according to estimated temporal gradient. This helps to achieve quick response to fast shading signal changes.

The technique improves over spatial upsampling in that undersampled information in the low-resolution shading result can be recovered from previous frames. It was implemented on a deferred shading renderer and tested using several computationally expensive scenes (Herzog et al., 2010). Figure 33 demonstrates the method achieving significant improvement in speed ($11\times$) without much sacrifice in quality. It combines the benefits of both spatial upsampling and temporal reprojection and only introduces a small performance overhead.

## 5 Temporal Coherence in Object Space

Besides algorithms for pixel reprojection, TC can also be used to speed up algorithms that operate in object space. For example, TC is often utilized to speed up culling techniques like view frustum culling and online occlusion culling. TC can also be exploited for other purposes, e.g., to achieve real-time global illumination based on the instant radiosity algorithm proposed by Keller (1997). In this section, we discuss some of these uses of TC.

### 5.1 Temporal Coherence in Culling Techniques

View-frustum culling and visibility culling are important acceleration techniques for real-time rendering. View-frustum culling simply prunes all objects in the scene which don't intersect the current view frustum. Visibility culling aims to prune all objects that are invisible because of occlusion due to other objects as early as possible in the pipeline. Visibility culling enables us to achieve the important property of *output-sensitivity*, i.e., the render time depends only on the complexity of the actual output, not the scene complexity (refer to Figure 34). Traditionally many rendering engines use preprocessed visibility and compute potentially visible sets (PVSs) for a set of view cells in a lengthy offline step. However, recent advances arguably make online occlusion culling (computing visibility on the fly for the current view point) a more attractive choice. Online oc-

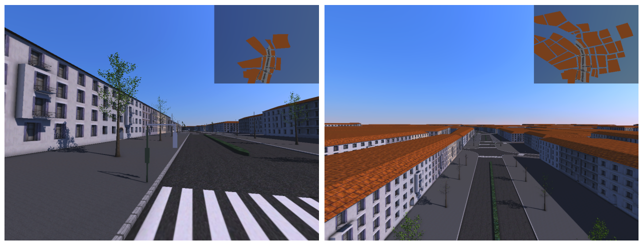


**Figure 34:** *(left) Using visibility culling, we send only the visible buildings bordering to the street (shown as red roofs in the small overview window) to the GPU for a low view point. (right) This is only a fraction of the buildings rendered for a high view point.*

clusion culling does not require to preprocess and store visibility information, and naturally allows for dynamic scenes. The major challenge is to reduce the overhead produced by the online visibility calculations.

To accelerate culling techniques like view frustum culling and online occlusion culling, TC is extensively used. It's importance, in particular for online occlusion culling, cannot be stressed enough - in fact utilizing TC is one of the key concepts to make online occlusion culling feasible in practice. The other key concept are spatial hierarchies, in order to reduce the complexity of the traversal algorithm from $O(n)$ to $O(log(n))$ in the number of visible objects $n$.

As proposed by Assarsson and Möller (2000), it is beneficial to use temporal coherence to speed up the plane tests for view frustum culling. A fast rejection of a bounding box is possible because it is very likely that a bounding box that was outside a plane in the previous frame is still outside in the current frame.

Due to TC, most objects that where visible in the previous frame are also visible in the current frame. Querying the visibility of these objects will result in many wasted tests and potentially cause significant overhead, which is not acceptable in practical applications like games. For example, consider an extreme scenario where everything is visible and nothing can be culled. Furthermore, culling can be only effective if there is something that we can cull against. An approximate front-to-back ordering of objects helps in this regard, but in many cases will fail to capture the visibility relations.

This observation leads to the following general strategy, which is implemented in different forms by all state-of-the-art occlusion culling algorithms: We first establish a *visible front* of those objects that were visible in the previous frame without expensive testing, assuming that they *stay visible*. Afterwards, we query the visibility the remaining objects against this visible front, assuming that these *stay invisible*. To keep the overdraw low, we also have to update the visibility classifications of the object from the visible front. This can be done in a lazy manner, assuming that there is coherence over several frames.

The clever hierarchical z-buffer algorithm proposed by Greene et al. (1993) uses both spatial hierarchies and TC for maximal efficiency. To accelerate visibility queries, it maintains a two-fold hierarchy - an image pyramid over the z-buffer and an octree hierarchy over the objects, as well as TC to establish the visible front. The feasibility of this algorithm suffered from the drawback that only parts of it have been supported by the hardware until now. The conceptually related algorithm of Zhang et al. (1997) aimed to speed up the queries by utilizing fast texture hardware.

*Coherent Hierarchical Culling*

```
CHC begin
DistanceQueue.push(Root); // initialize traversal

while !DistanceQueue.Empty() ||
      !QueryQueue.Empty() do
   //-- first part: handle queries
   while !QueryQueue.Empty() &&
         FirstQueryFinished()) do
            N = QueryQueue.Dequeue();
      if Q.visible then PullUpVisibility(N);

  //-- second part: handle traversal
  if !DistanceQueue.Empty() then
     N = DistanceQueue.DeQueue();

     if InsideViewFrustum(N) then
        wasVisible = N.visible;
        N.visible = false; // reset classification
        // query prev. invisible nodes and leaves
        if !wasVisible && IsLeaf(N) then
           IssueOcclusionQuery(N);
           QueryQueue.Enqueue(N);
        // always traverse prev. visible nodes
        if (wasVisible) then TraverseNode(N);
end CHC

TraverseNode(N) begin
if IsLeaf(N) then
   Render(N);
else
   DistanceQueue.PushChildren(N);
end TraverseNode

PullUpVisibility(N) begin
while !N.IsVisible do
   N.IsVisible = true; N = N.Parent;
end PullUpVisibility
```

**Figure 35:** *Listing of the coherent hierarchical culling (CHC) algorithm.*

### 5.1.1 Hardware Occlusion Queries

Despite these interesting research efforts, online occlusion culling was mostly considered too costly for practical usage before dedicated hardware support for occlusion queries existed. A turning point came when hardware acceleration for occlusion queries were finally available for consumer graphics hardware, which simply returns the number of visible pixels of the queried geometry. For rendering acceleration they are used conservatively, by querying the visibility of a simple *proxy geometry* (e.g., the axis aligned bounding box of an object). After the introduction of hardware accelerated occlusion queries, the potential of online occlusion culling gained the attention of both the research community and the industry, and opened the field for a variety of algorithms (Klosowski and Silva., 2001; Hillesland et al., 2002; Govindaraju et al., 2003; Staneker et al., 2004). However, the queries still come with a non-negligible cost, and algorithms have to find a way to fill the latency until the result returns in a meaningful way. A naive hierarchical implementation which waits for the query result at each node before further traversing a hierarchy can actually *slow down* rendering significantly due to the idle time of GPU and CPU – this is where TC comes into play.

### 5.1.2 Coherent Hierarchical Culling (CHC)

The *coherent hierarchical culling* (CHC) algorithm by Bittner et al. (2004) utilizes TC to avoid the *idle time* caused by the synchronisation between the CPU and the GPU based occlusion queries. This algorithm is rather simple and intuitive and provides good performance in most cases. A listing of the (slightly simplified) CHC algorithm is shown in Figure 35. The algorithm works for any underlying type of spatial hierarchy (e.g., bounding volume hierarchy (BVH), kD-tree, octree), only assuming that the objects are stored in the leaves. In our experiments BVH (using the surface-area heuristics (Havran, 2000) for construction) turned out to be a good choice.

The CHC algorithm traverses nodes in a front-to-back order. In order to exploit TC, nodes that were visible in the last frame are still assumed visible in the current frame and vice-versa. Hence we always wait for the query result of a previously invisible node. Likewise, previously visible nodes are always traversed and rendered in the current frame (line 24 in the listing). In the original CHC algorithm a query is always issued for previously visible nodes (line 21), but we do not wait for the result. For this purpose the pending queries are managed in a dedicated *query queue* (lines 7–10). Fortunately, graphics hardware provides a cheap way to test if a query result is available (the FirstQueryFinished() function in line 8).Once the query result is available we use it for the visibility classification in the *next frame*. This way, both CPU and GPU are kept busy while waiting for query results. The algorithm further exploits (spatial) coherence by identifying invisible subtrees. This is handled by automatically setting a node to invisible (line 18) and then pulling up the classification of visible nodes (line 10). This way, the visibility of many previously invisible leaf nodes can be decided with a single query.

A problem of the original CHC algorithm is the high number of wasted occlusion queries (i.e., queries that report an object to be visible or queries that are more costly than rendering the node itself), which induce an overhead and make CHC noticeable slower than view-frustum culling for view points where most of the scene is visible. To reduce the number of queries for previously visible nodes, the authors suggested to test previously visible nodes only every number of frames.

### 5.1.3 Near Optimal Hierarchical Culling (NOHC)

The *near optimal hierarchical culling* (NOHC) algorithm (Guthe et al., 2006) computes the probability that a node will stay visible based on a statistical model of the estimated screen coverage due to objects rendered previously in the current frame. The expected value for accumulated screen coverage $c_{scr}$ after $i+1$ rendered objects is computed as

$$c_{scr}(O_{i+1}) = c_{scr}(O_i) + (1 - c_{scr}(O_i))c(O_i), \qquad (23)$$

where $C(O_i)$ is the estimated screen coverage of object $O_i$. The feasibility of a query is then decided using a sophisticated cost model that takes the probability that a node is still visible into account, and weights it with the cost of a query versus the cost of rendering an object. These quantities are measured during an offline hardware calibration step that samples different parameters like time required for rasterization or time required for transformation.

Note that, this calibration model still does not take the complex interaction of queries and objects during rendering into account, and the fact that much of the overhead of a query is caused by *GPU render state changes* that happen during this interaction. I.e., when

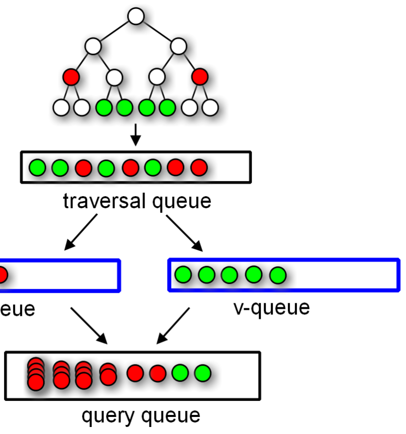


**Figure 36:** *Different queues used by the CHC++ algorithm. The queues which were not used by the CHC algorithm are highlighted in blue. Previously invisible / visible nodes are depicted in red / green. The multiple nodes in the query queue indicate multiqueries.*

changing between render mode and query mode, at least *depth write* must be turned on and off on the GPU, which can accumulate to significant overhead for many of queries, depending on the cost of these state changes on the target hardware.

### 5.1.4 Coherent Hierarchical Culling Revisited (CHC++)

The CHC++ algorithm (Mattausch et al., 2008) addresses the before mentioned drawbacks while keeping the structure of the original CHC algorithm. It was especially designed to fit the demands of modern rendering engines like Ogre3D. Unlike NOHC, it does not rely on hardware calibration. CHC++ aims to a) reduce the overall number of queries and b) reduce the query overhead (e.g., due to the induced state changes) by making better use of temporal and spatial coherence of visibility. It extends the CHC algorithm with a couple of simple optimizations like adaptive visibility prediction and query batching. As a result of the new optimizations, the number of issued occlusion queries and the number of rendering state changes are significantly reduced, leading to a speedup of 2-3 times compared to the previous state-of-the-art. The most important of them are:

**Queues for batching of queries.** Before a node is queried, it is appended to a queue. Separate queues are used for accumulating previously visible and previously invisible nodes. We use the queues to issue *batches of queries* instead of individual queries. This reduces state changes by one to two orders of magnitude.

**Multiqueries.** CHC++ compiles *multiqueries*, which are able to cover more nodes by a single occlusion query. This method is able to reduce the number of queries for previously invisible nodes up to an order of magnitude by making better use of TC. The decision of including a node in a multiquery is based on its history. I.e., nodes that were invisible for a long time are likely to stay invisible, hence they can be handled by a single query. Note that the nodes can be spatially unrelated.

**Randomized sampling pattern for visible nodes.** The algorithm applies a temporally jittered sampling pattern for scheduling queries for previously visible nodes. This reduces the number of queries for visible nodes and while spreading them evenly over the frames of the walkthrough. This way frame rate drops can be avoided that happen because of many queries being issued in the same frame, caused by aligned visibility events.

An overview of the queues used in CHC++ can be seen in Fig-

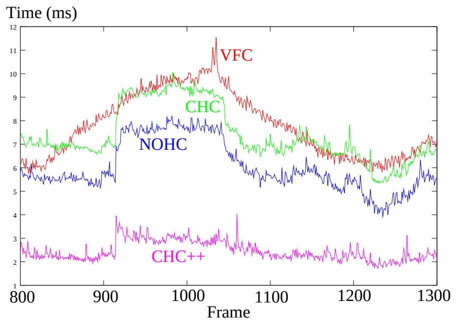


**Figure 37:** *Comparison of view frustum culling (VFC), coherent hierarchical culling (CHC), near optimal hierarchical culling (NOHC), and CHC++, showing a problematic view point for CHC.*

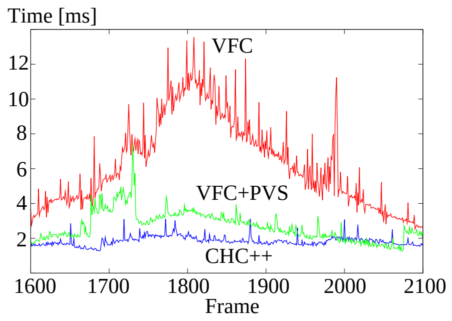


**Figure 38:** *Comparison of view frustum culling (VFC), view frustum culling and potentially visible sets (VFC+PVS), and CHC++ (Bittner et al., 2009).*

ure 36. As it keeps the original structure of the CHC algorithm (refer to listing 35), an existing CHC implementation can be extended to CHC++ in a straightforward fashion. Like CHC, it uses a query queue to manage the pending queries and a traversal queue for the front-to-back traversal of the spatial hierarchy. CHC++ allocates two additional queues for batching previously visible and invisible nodes. Furthermore, CHC++ has a mechanism for multi-queries which allows queries over multiple nodes.

Note that the batching is a very simple optimization which, according to our experiments, improves the performance the most. Hence programmers are advised to take the CHC algorithm, and start by extending it with the batching optimization. The separation into two queues makes sense because previously visible node queries have no dependencies as we do not suspend rendering until the query result is available. Hence previously visible nodes can be queried at any time, for example to fill up wait time, or the queries can even be issued after the main algorithm and the query results fetched in the beginning of the next frame.

A performance comparison of view frustum culling with the occlusion culling algorithms CHC, NOHC, and CHC++ in the Powerplant scene (12M triangles) is shown in Figure 37 (using a NVIDIA GeForce 8800 GTX). This plot shows a difficult view point where CHC has noticeable overhead. The hardware calibration process of NOHC brings some improvements, while the enhancements of CHC++ bring even more.

Figure 38 shows timings in the Powerplant scene (using a NVIDIA GeForce 280 GTX). Interestingly, online occlusion culling with CHC++ is faster than rendering based on potentially visible sets (PVSs) in this walkthrough. In moderately to highly occluded scenario, the overhead of the online occlusion culling algorithm typically is less than the overhead caused by the greater conservativity of preprocessed visibility (note that view cells under a certain size are not feasible).

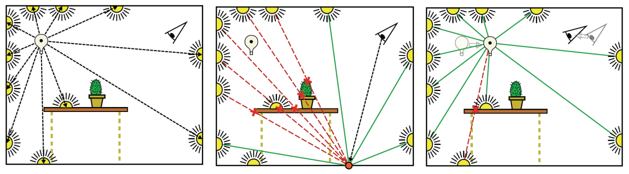


**Figure 39:** *(Left) Instant radiosity shoots paths from the light source, and creates virtual point lights at the intersection with geometry. (Middle) Using shadow maps, the visibility of each VPL and their contribution to the current image is determined. (Right) Temporal coherence: When the view point or light source moves, one of the VPLs becomes invisible from the light source, all the others are reused (Laine et al., 2007).*

As a conclusion, we can state that online occlusion culling is at least comparable to PVS-based rendering for common scenarios and should be considered as a viable alternative to time-consuming preprocessing by developers. Still it has to be mentioned that the overhead due to idle time cannot be completely avoided by these conservative occlusion culling algorithm. I.e., it can happen that all rendering commands for the previously visible nodes have been issued, but rendering of the visible front is not yet finished. Then we have to wait for the query result of the next previously invisible node. To achieve online occlusion culling algorithm with neglectable overhead, we would have to give up conservativity and accept a latency of a single frame.

## 5.2 Incremental Instant Radiosity

Laine et al. (2007) exploit TC to reach real-time frame rates in another variation of the instant radiosity algorithm (described in Section 4.5.2). For the sake of performance, the authors restrict their method to compute first bounce indirect illumination. Still hundreds virtual point lights are needed for convincing global illumination, and each VPL requires a shadow maps of for visibility computation. While Ritschel et al. (2008) addresses this problem by using simplified shadow map representations, this algorithm heavily utilizes TC to achieve real-time frame rates. In particular, they reuse the valid VPLS from the last frame and recomputing only a budget of invalid shadow maps in a frame. The validity of each VPL is tested with a ray caster. The algorithm uses a 2D Delaunay triangulation to manage the VPL distribution. The main task is to keep a good distribution of VPLs in every frame. When choosing the location of new VPLs, the method minimizes *dispersion*, which is computed as the radius of the largest empty circle that contains no sample points. The algorithm is visualized in Figure 39.

Note that the method is conceptually similar to the occlusion culling algorithms discussed in the previous section. The visibility is assumed to change slowly, and hence a lazy update strategy is chosen (in fact, the method could be seen as a special kind of visibility algorithm itself). While the parameters settings affect only the rendering speed in case of the discussed occlusion culling algorithms, both performance and rendering quality are affected here. The algorithm captures changes in the scene with a certain latency, and shadows cast from dynamic objects are not supported. The authors report a speedup from 1.4–6.8 for different scenes and resolutions. In their tests, they fixed the number of VPLS to 256 and the recomputation budget to 4–8 VPLs.

## 6 Conclusions

In this course notes we have described the use of TC for real-time rendering. Due to the rigorous time constrains in real-time rendering we want to reuse resources and previous calculations as much as possible to compute the best possible quality within the small time slot between consecutive frames. We have shown that in general the coherence between frames is very high – hence a high potential for rendering acceleration and quality improvement exists. This potential was recognised almost from the beginning of computer graphics research and many clever methods have been proposed. However, most of the earlier works are designed for offline and CPU based rendering systems. Hence they are either too complicated for real-time use or fail to exploit the massive parallelism of modern graphics hardware.

The way to employ TC in real-time rendering was introduced with real-time reverse reprojection, a screen-space approach that allows to cache and reuse shading results from previous frames. This method is efficient on modern GPUS, simple to implement, and general in the sense that it can be used for a variety of different shading effects. It opens many possibilities to accelerate existing algorithms, and to implement costly shading effects in real-time by spreading out computations over time. We have also discussed limitations and the errors that are introduced due to repeated resampling. A number of practical applications using this scheme are discussed in detail, including multi-pass effects, amortized supersampling, LOD blending, shadows and global illumination acceleration. Overall, real-time reprojection can be of great value to game developers and it pays off to examine existing algorithms for their suitability for conversion into versions that use TC.

Besides from screen-space reprojection we have described methods that use TC in object space, mainly for the purpose of visibility culling. These methods are mainly concerned with avoiding wasting queries and algorithm time by predicting the visibility classification based on previous frames. We have shown that those methods can significantly improve performance without any preprocessing or restrictions to the scene configurations. Additionally, a practical method was presented that uses object-space TC to achieve full-fledged global illumination in real time.

We believe that the possibilities of using TC for real-time rendering have not yet been exploited. The research potential mainly lies in 2 directions: 1) to make the current TC operations more robust, avoid artifacts, and make better use of the available TC, and 2) to find new algorithms and fields that can benefit from TC. A big potential may also lie in better exploitation of combined spatial and temporal coherence techniques. We think this is a fast emerging direction in the real-time rendering field and we hope this course will help conveying the idea and inspire new research in the area.

## References

ADELSON, S. J. and HODGES, L. F. 1992. Visible surface ray-tracing of stereoscopic images. In *ACM-SE 30: Proceedings of the 30th annual Southeast regional conference*, New York, NY, USA. ACM, pages 148–156. ISBN 0-89791-506-2.

ADELSON, S. J. and HODGES, L. F. 1995. Generating exact ray-traced animation frames by reprojection. *IEEE Comput. Graph. Appl.*, 15(3):43–52. ISSN 0272-1716.

AKELEY, K. and SU, J. 2006. Minimum triangle separation for correct z-buffer occlusion. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, New York, NY, USA. ACM, pages 27–30. ISBN 3-905673-37-1.

APODACA, A. and GRITZ, L. 2000. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

ASSARSSON, U. and MÖLLER, T. 2000. Optimized view frustum culling algorithms for bounding boxes. *Journal of graphics, GPU, and game tools*, 5(1):9–22.

BADT JR., S. 1988. Two algorithms for taking advantage of temporal coherence in ray tracing. *VC*, 4:123–132.

BAVOIL, L., SAINZ, M., and DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *SIGGRAPH '08: ACM SIGGRAPH 2008 talks.*

BISHOP, G., FUCHS, H., MCMILLAN, L., and ZAGIER, E. J. S. 1994. Frameless rendering: double buffering considered harmful. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, New York, NY, USA. ACM, pages 175–176. ISBN 0-89791-667-0.

BITTNER, J., MATTAUSCH, O., WONKA, P., HAVRAN, V., and WIMMER, M. 2009. Adaptive global visibility sampling. In *SIGGRAPH '09: ACM SIGGRAPH 2009 Papers*, New York, NY, USA. ACM.

BITTNER, J., WIMMER, M., PIRINGER, H., and PURGATHOFER, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3):615–624. ISSN 0167-7055. Proceedings EUROGRAPHICS 2004.

CHEN, B., EDWARD, J., and II, S. 1999. Lod-sprite technique for accelerated terrain rendering. In *ISBN 0-7803-5897-X. Held in*, pages 291–298.

CHEN, S. E. and WILLIAMS, L. 1993. View interpolation for image synthesis. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, New York, NY, USA. ACM, pages 279–288. ISBN 0-89791-601-8.

COOK, R. L. and TORRANCE, K. E. 1981. A reflectance model for computer graphics. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, New York, NY, USA. ACM, pages 307–316. ISBN 0-89791-045-1.

COORG, S. and TELLER, S. 1996. Temporally coherent conservative visibility (extended abstract). In *SCG '96: Proceedings of the twelfth annual symposium on Computational geometry*, New York, NY, USA. ACM, pages 78–87. ISBN 0-89791-804-5.

DAVIS, T. A. 1998. Generating computer animations with frame coherence in a distributed computing environment. In *ACM-SE 36: Proceedings of the 36th annual Southeast regional conference*, New York, NY, USA. ACM, pages 1–7. ISBN 1-58113-030-9.

DAVIS, T. A. and DAVIS, E. W. 1999. Exploiting frame coherence with the temporal depth buffer in a distributed computing environment. In *PVGS '99: Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics*, Washington, DC, USA. IEEE Computer Society, pages 29–38. ISBN 1-58113-237-9.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., and WORLEY, S. 2003. *Texturing and Modeling: A Procedural Approach*. Morgan Kaufmann, 3rd edition.

FOX, M. and COMPTON, S. 2008. Ambient occlusive crease shading. *Game Developer Magazine (March 2008).*

GAUTRON, P. 2008. Temporal radiance caching. In *SIGGRAPH '08: ACM SIGGRAPH 2008 classes*, New York, NY, USA. ACM, pages 1–49.

GIEGL, M. and WIMMER, M. 2006. Unpopping: Solving the image-space blend problem for smooth discrete lod transitions. *Computer Graphics Forum*, 26(1):46–49. ISSN 0167-7055.

GOVINDARAJU, N. K., SUD, A., YOON, S.-E., and MANOCHA, D. 2003. Interactive visibility culling in complex environments using occlusion-switches. In *SI3D*, pages 103–112.

GREENE, N., KASS, M., and MILLER, G. 1993. Hierarchical Z-buffer visibility. In *Computer Graphics (Proceedings of SIGGRAPH '93)*, pages 231–238.

GRÖLLER, M. E. 1992. *Coherence in Computer Graphics*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria.

GUTHE, M., BALÁZS, Á., and KLEIN, R. 2006. Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In T. Akenine-Möller and W. Heidrich, editors, *Eurographics Symposium on Rendering 2006*, The Eurographics Association.

HAEBERLI, P. and AKELEY, K. 1990. The accumulation buffer: hardware support for high-quality rendering. In *Proc. SIGGRAPH '90*, ACM, pages 309–318.

HASSELGREN, J. and AKENINE-MÖLLER, T. 2006. An efficient multi-view rasterization architecture. In *Eurographics Symposium on Rendering*, pages 61–72.

HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

HAVRAN, V., BITTNER, J., and SEIDEL, H.-P. 2003. Exploiting temporal coherence in ray casted walkthroughs. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, New York, NY, USA. ACM Press, pages 149–155. ISBN 1-58113-861-X.

HAVRAN, V., DAMEZ, C., MYSZKOWSKI, K., and SEIDEL, H.-P. 2003. An efficient spatio-temporal architecture for animation rendering. In *EGRW '03: Proceedings of the 14th Eurographics workshop on Rendering*, Springer, pages 106–117.

HECKBERT, P. S. and HERF, M. 1997. Simulating soft shadows with graphics hardware. Technical Report CMU-CS-97-104, CS Dept., Carnegie Mellon U. CMU-CS-97-104, http://www.cs.cmu.edu/ ph.

HERZOG, R., EISEMANN, E., MYSZKOWSKI, K., and SEIDEL, H.-P. 2010. Spatio-temporal upsampling on the GPU. In *Symposium on Interactive 3D Graphics and Games*, ACM.

HILLESLAND, K., SALOMON, B., LASTRA, A., and MANOCHA, D. 2002. Fast and simple occlusion culling using hardware-based depth queries. Technical Report TR02-039, Department of Computer Science, University of North Carolina - Chapel Hill.

HUBSCHMAN, H. and ZUCKER, S. W. 1981. Frame-to-frame coherence and the hidden surface computation: Constraints for a convex world. In *SIGGRAPH '81: Proceedings of the 8th annual conference on Computer graphics and interactive techniques*, New York, NY, USA. ACM, pages 45–54. ISBN 0-89791-045-1.

JEVANS, D. A. 1992. Object space temporal coherence for ray tracing. In *Graphics Interface 92*, pages 176–183.

KELLER, A. 1997. Instant radiosity. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 49–56.

KELLER, A. and HEIDRICH, W. 2001. Interleaved sampling. In *Proceedings of the 12th Eurographics Workshop on Rendering Techniques*, London, UK. Springer-Verlag, pages 269–276. ISBN 3-211-83709-4.

KLOSOWSKI, J. T. and SILVA., C. T. 2001. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379. ISSN 1077-2626.

KNECHT, M. 2009. Real-time global illumination using temporal coherence. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria.

KNECHT, M., TRAXLER, C., MATTAUSCH, O., PURGATHOFER, W., and WIMMER, M. 2010. Differential instant radiosity for mixed reality. In *Proc. Ninth IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR'10)*, Seoul, Korea.

LAINE, S., SARANSAARI, H., KONTKANEN, J., LEHTINEN, J., and AILA, T. 2007. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of Eurographics Symposium on Rendering 2007*, Eurographics Association, pages 277–286.

LANDIS, H. 2002. Production-ready global illumination. In *Proceedings of the conference on SIGGRAPH 2002 course notes 16*.

LENGYEL, J. and SNYDER, J. 1997. Rendering with coherent layers. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co., pages 233–242. ISBN 0-89791-896-7.

MARK, W. R., MCMILLAN, L., and BISHOP, G. 1997. Post-rendering 3d warping. In *SI3D '97: Proceedings of the 1997 symposium on Interactive 3D graphics*, New York, NY, USA. ACM, pages 7–ff. ISBN 0-89791-884-3.

MATTAUSCH, O., BITTNER, J., and WIMMER, M. 2008. Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum (Proceedings of Eurographics 2008)*, 27(3):221–230. ISSN 0167-7055.

MATTAUSCH, O., SCHERZER, D., and WIMMER, M. 2010. High-quality screen-space ambient occlusion using temporal coherence. *Computer Graphics Forum (to appear)*.

MCMILLAN, L. and BISHOP, G. 1995. Head-tracked stereoscopic display using image warping. In *Proceedings SPIE, volume 2409*, pages 21–30.

MITTRING, M. 2007. Finding next gen - cryengine 2. In *Proceedings of the conference on SIGGRAPH 2007 course notes, course 28, Advanced Real-Time Rendering in 3D Graphics and Games*, ACM Press, pages 97–121.

NEHAB, D., SANDER, P. V., and ISIDORO, J. R. 2006. The real-time reprojection cache. In *ACM SIGGRAPH Sketch*, page 185.

NEHAB, D., SANDER, P. V., LAWRENCE, J., TATARCHUK, N., and ISIDORO, J. R. 2007. Accelerating real-time shading with reverse reprojection caching. In *Graphics Hardware*, pages 25–35.

QU, H., WAN, M., QIN, J., and KAUFMAN, A. 2000. Image based rendering with stable frame rates. In *VISUALIZATION '00: Proceedings of the 11th IEEE Visualization 2000 Conference (VIS 2000)*, Washington, DC, USA. IEEE Computer Society. ISBN 0-7803-6478-3.

REGAN, M. and POSE, R. 1994. Priority rendering with a virtual reality address recalculation pipeline. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, New York, NY, USA. ACM, pages 155–162. ISBN 0-89791-667-0.

RITSCHEL, T., GROSCH, T., KIM, M. H., SEIDEL, H.-P., DACHSBACHER, C., and KAUTZ, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Transactions on Graphics (Proc. SIGGRAPH ASIA 2008)*, 27(5): 129.

SCHAUFLER, G. 1996. Exploiting frame to frame coherence in a virtual reality system. In *VRAIS '96: Proceedings of the 1996 Virtual Reality Annual International Symposium (VRAIS 96)*, Washington, DC, USA. IEEE Computer Society, page 95. ISBN 0-8186-7295-1.

SCHERZER, D., JESCHKE, S., and WIMMER, M. 2007. Pixel-correct shadow maps with temporal reprojection and shadow test confidence. In *Eurographics Symposium on Rendering*, pages 45–50.

SCHERZER, D., SCHWÄRZLER, M., MATTAUSCH, O., and WIMMER, M. 2009. Real-time soft shadows using temporal coherence. *Lecture Notes in Computer Science (LNCS)*.

SCHERZER, D. and WIMMER, M. 2008. Frame sequential interpolation for discrete level-of-detail rendering. *Computer Graphics Forum (Proceedings EGSR 2008)*, 27(4):1175–1181. ISSN 0167-7055.

SEGOVIA, B., IEHL, J.-C., MITANCHEY, R., and PÉROCHE, B. 2006. Non-interleaved deferred shading of interleaved sample patterns. In *Proceedings of Graphics Hardware*, pages 53–60.

SHADE, J., GORTLER, S., HE, L.-W., and SZELISKI, R. 1998. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, New York, NY, USA. ACM, pages 231–242. ISBN 0-89791-999-8.

SIMMONS, M. and SEQUIN, C. H. 2000. Tapestry: A dynamic mesh-based display representation for interactive rendering. In *Proceedings of the 11th Eurographics Workshop on Rendering*, pages 329–340.

SITTHI-AMORN, P., LAWRENCE, J., YANG, L., SANDER, P. V., and NEHAB, D. 2008. An improved shading cache for modern GPUs. In *Proc. of Graphics Hardware*, pages 95–101.

SITTHI-AMORN, P., LAWRENCE, J., YANG, L., SANDER, P. V., NEHAB, D., and XI, J. 2008. Automated reprojection-based pixel shader optimization. *ACM Trans. Graph.*, 27(5):127.

SMEDBERG, N. and WRIGHT, D. 2009. Rendering techniques in gears of war 2.

SMKY, M., KINUWAKI, S.-I., DURIKOVIC, R., and MYSZKOWSKI, K. 2005. Temporally coherent irradiance caching for high quality animation rendering. In *The European Association for Computer Graphics 26th Annual Conference EUROGRAPHICS 2005*, volume 24 of *Computer Graphics Forum*, Dublin, Ireland. Blackwell, pages xx–xx.

STAMMINGER, M., HABER, J., SCHIRMACHER, H., and SEIDEL, H.-P. 2000. Walkthroughs with corrective texturing. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, London, UK. Springer-Verlag, pages 377–388. ISBN 3-211-83535-0.

STANEKER, D., BARTZ, D., and STRASSER, W. 2004. Occlusion culling in OpenSG PLUS. *Computers and Graphics*, 28(1):87–92.

SUTHERLAND, I. E., SPROULL, R. F., and SCHUMACKER, R. A. 1974. A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55. ISSN 0360-0300.

SZIRMAY-KALOS, L., UMENHOFFER, T., TOTH, B., SZECSI, L., and SBERT, M. 2010. Volumetric ambient occlusion for real-time rendering and games. *IEEE Computer Graphics and Applications*, 30:70–79.

TOMASI, C. and MANDUCHI, R. 1998. Bilateral filtering for gray and color images. In *Proc. ICCV*, page 839.

TORBORG, J. and KAJIYA, J. T. 1996. Talisman: commodity realtime 3d graphics for the pc. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, New York, NY, USA. ACM, pages 353–363. ISBN 0-89791-746-4.

VELÁZQUEZ-ARMENDÁRIZ, E., LEE, E., BALA, K., and WALTER, B. 2006. Implementing the render cache and the edge-and-point image on graphics hardware. In *GI '06: Proceedings of Graphics Interface 2006*, Toronto, Ont., Canada, Canada. Canadian Information Processing Society, pages 211–217. ISBN 1-56881-308-2.

WALTER, B., DRETTAKIS, G., and GREENBERG, D. P. 2002. Enhancing and optimizing the render cache. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association, pages 37–42. ISBN 1-58113-534-3.

WALTER, B., DRETTAKIS, G., and PARKER, S. 1999. Interactive rendering using the render cache. In D. Lischinski and G. Larson, editors, *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, New York, NY. Springer-Verlag/Wien, pages 235–246.

WIMMER, M., GIEGL, M., and SCHMALSTIEG, D. 1999. Fast walkthroughs with image caches and ray casting. In M. Gervautz, D. Schmalstieg, and A. Hildebrand, editors, *Virtual Environments '99. Proceedings of the 5th Eurographics Workshop on Virtual Environments*, Eurographics, Springer-Verlag Wien, pages 73–84. ISBN 3-211-83347-1.

YANG, L., NEHAB, D., SANDER, P. V., SITTHI-AMORN, P., LAWRENCE, J., and HOPPE, H. 2009. Amortized supersampling. *ACM Trans. Graph.*, 28(5):135.

YANG, L., SANDER, P. V., and LAWRENCE, J. 2008. Geometry-aware framebuffer level of detail. *Computer Graphics Forum*, 27 (4):1183–1188.

ZHANG, H., MANOCHA, D., HUDSON, T., and III, K. E. H. 1997. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH*, pages 77–88.

ZHU, T., WANG, R., and LUEBKE, D. 2005. A gpu-accelerated render cache. *Pacific Graphics, (Short Paper Session)*.