Checkerboard Rendering for Real-Time Upscaling on Intel® Integrated Graphics

Trapper Mcferron and Adam Lake

8/9/2018

Abstract

In this white paper, we present checkerboard rendering (CBR) for real-time upsampling on Intel® integrated graphics. We first review some of the previous work on dynamic real-time upscaling techniques, including publicly available references to the Sony PS4* Pro, the Frostbite* engine, and others. Next, we build up the motivation for our use of quarter-resolution 2x multi-sampled anti-aliasing (MSAA) render targets and introduce our temporal reconstruction technique. Finally, we include details of how to modify a renderer to support CBR for both a simple forward and deferred rendering case. In our test scene we show an average speed-up with CBR of 1.14x to 1.2x, and as much as 1.4x in frames where pixel shading cost dominates, with minimal impact on the visual quality of the result. We believe the relative ease of implementation, scalability across multiple resolutions, and compatibility across graphics processing unit (GPU) vendors make CBR an attractive solution for developers looking to get the most out of their GPU. Source code to a reference implementation is made available with this article.

Contents

Introduction

Checkerboard rendering is a technique that produces full-resolution pixels with a significant reduction in shading and minimal impact on visual quality. Checkerboard rendering is fully compatible with modern post-processing approaches to anti-aliasing and it can be implemented in both forward and deferred rendering pipelines.

In this white paper we discuss our motivation and previous work in the field. Next, we provide an overview of our implementation and some of the issues specific to our solution. Then we present details of our implementation and results, discussing both performance and quality. Finally, we provide areas of future work and next steps. The accompanying sample code demonstrates the technique and provides a base implementation for both forward and deferred rendering pipelines. While our focus was on integrated graphics processing units (GPUs), our implementations for both forward and deferred rendering have been tested to run on both AMD and NVIDIA discrete GPUs.

Motivation

Given the recent adoption of checkerboard rendering for upscaling on the Sony PS4* Pro we thought it would be useful to explore the technique on Intel® integrated graphics. In particular, we wanted to develop a solution to the problem of rendering content that was designed for a higher original target resolution and may not perform out of the box on an integrated GPU. Our hypothesis was that checkerboard rendering can be used to provide a higher quality solution than dynamic resolution rendering alone could provide.

A second motivation for this paper was to create an easy-to-follow, non-intrusive, hardware-independent implementation that benefits both integrated and discrete GPUs across a range of resolutions. This sample is meant to be as "drop in" as possible, while also providing the ability to leverage custom solutions that suit the quality versus performance needs of different developers. Game developers can grow their audience by using checkerboard rendering to deliver high-quality visuals while supporting a broader range of platforms with varying GPU rendering performance. At the time of writing the authors know of no other publicly available checkerboard rendering samples that include source code.

Previous Work

Graphics architects are always thinking of creative ways to increase frame rates by imperceptibly reducing the amount of processing required to display a complete frame. Many techniques focus on reducing the geometric level of detail. In our case, however, the goal is to reduce the amount of shading *after* geometric level-of-detail techniques have been applied. To accomplish this, we adapted a technique called *checkerboard rendering* (CBR) that other developers have been using when faced with a similar challenge. A subtle difference between our goal and the goal of the previous work in CBR is that the motivation in previous work on the Sony PS4 Pro was primarily around upscaling from 1080p (1920 x 1080) to 4K (3840 x 2160) displays and working on assets and rendering architectures with mid-range to high-end in mind. Here, we explore techniques that permit us to take content that had an original target of 1080p and instead render at 540p (960 x 540), then use a CBR technique to scale up to 1080p. However, before we get started on the CBR algorithm details, we will discuss some of the earlier work on dynamic resolution rendering (DRR), which can be combined with CBR. In fact, at least two of the CBR solutions reference DRR being used in addition to CBR.

White papers and publicly available source code on CBR algorithms are sparse. In fact, to understand the previous work in the field, we needed to rely on presentations from technical conferences such as GDC or SIGGRAPH, and less on research papers or previously published sample code.

Dynamic Resolution Rendering

Doug Binks published one solution to the problem of upscaling in the article *Dynamic Resolution Rendering*. This technique dynamically adjusts the resolution based on the GPU workload, and works well. However, as pointed out in the article, it is only rendering at the resolution of the render target. All of the usual artifacts of upsampling apply in this scenario. For example, we only have visibility and color information at the render target resolution, which limits the quality of the upscaling. Techniques such as temporal anti-aliasing with jittered sampling help to improve the quality at an incremental cost in performance, and the paper discusses the tradeoffs involved.

Sony PS4* Pro and CBR

Discussion about CBR first became popularized when looking for ways to upscale content that could run on the Sony PS4 Pro, and would take advantage of the ability to display at 4K (3840 x 2160) resolution. This amount of pixel throughput would not be achievable by many engines, and this motivated trying to achieve the highest quality images within the render budget of the new hardware. Mark Cerny points out that 9 of 13 games discussed in the article by Richard Leadbetter used CBR. A few of the titles including *Days Gone**, *Call of Duty*: Infinite Warfare*, *Rise of the Tomb Raider**, and *Horizon Zero Dawn** rendered up to 2160p, while *Watch Dogs* 2*, *Killing Floor* 2*, *InFAMOUS*,* and *Mass Effect*: Andromeda* all used CBR at 1080p. Sony supplied sample code to developers to make it easy to use CBR in their titles. Some titles, such as *Shadow of Mordor**, that did not use CBR, actually used a form of dynamic resolution described above. *Deus Ex: Mankind Divided** used CBR, but also with a frame buffer that varied between 1800p and 2160p, based on scene complexity. One key difference in our implementation is that we do not have support in current Intel integrated graphics hardware for object ID buffers at the same resolution as the depth buffer.

CBR in Rainbow Six* Siege

In 2016, Jalal El Mansouri of Ubisoft described the CBR technique used in *Rainbow Six* Siege*. The original goal was to hit 60 fps with 720p (1280 x 720) on consoles, and 4K on PCs, with decent frame rates, so Ubisoft explored techniques to achieve this. CBR was explored as a way to get around the temporal aliasing issues Ubisoft was seeing with temporal interlaced rendering. For the checkerboard implementation, they rendered to a quarter-size render target with 2x MSAA with color and z-values at each MSAA sample point. They also incorporated temporal anti-aliasing in the resolve shader with an additional resampling to remove some sampling artifacts (referred to as *teething*) that are introduced in the resolve phase. One interesting point about any of the in-engine rendering techniques with high temporal differences—for example, the flickering lights of a police car—is that they are made to take place over at least two frames. The technique Ubisoft presented is most like our implementation.

CBR in the Decima* Engine

At SIGGRAPH 2017, details of the CBR used in Guerilla Games's Decima* engine were presented by Giliam de Carpentier of Guerilla Games and Kohei Ishiyama of Kojima Productions. Rendering and most post processing were done at checkerboard resolution in their implementation. As sample points in the 2x MSAA render target do not match the actual pixel locations expected by the Fast Approximate Anti-Aliasing (FXAA) post process, they cleverly rotate the image 45 degrees, which aligns the sample points to a regular grid. This regular grid now matches actual rasterization locations expected by the post processing anti-aliasing. The results were very high quality in under 2ms on a Sony PS4 Pro.

CBR in the Frostbite* Engine

Frostbite* has used CBR in *Battlefield* 1* and *Mass Effect: Andromeda.* Graham Wihlidal gave a presentation at GDC 2017 on adding support for the technique to the Frostbite engine. The presentation provides an in-depth discussion of a significant portion of the rendering pipeline as well as the challenges faced integrating CBR into their engine. Their implementation incorporates temporal anti-aliasing, dynamic resolution rendering, and enhanced quality anti-aliasing (EQAA).

Technique Overview

Before going through the technical details of our implementation, and our use of a quarter-resolution 2x MSAA render targets, it's important for us to review the basics of traditional upscaling. We do this by first reviewing how pixel coverage and pixel colors are related in a standard, full-resolution render. Next, we step toward our final outcome by building up from a half-resolution render target; then we introduce temporal reconstruction with multiple half-resolution render targets. Describing the problems with half-resolution render targets illustrates the motivation behind the introduction of the quarter-resolution 2x MSAA render targets.

Full-Resolution Render

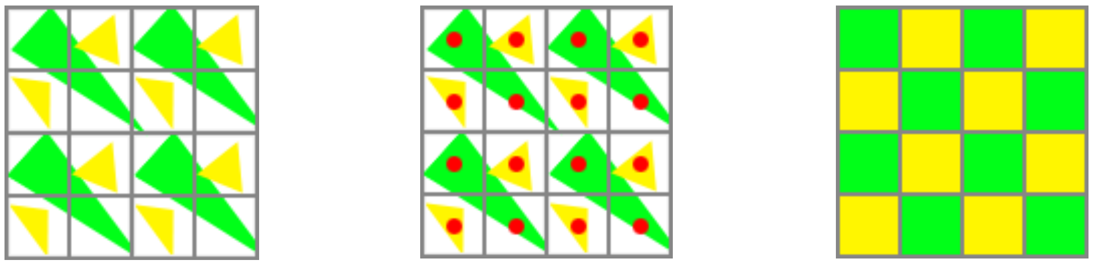


Figure 1: *Pixels are shaded based on triangle coverage; each grid cell represents a render target pixel to be shaded. Left: Triangles to be rasterized to the render target. Center: The coverage test; the red dots represent the sample coverage position for each pixel. Right: Pixel colors are shaded based on the results of the coverage test.*

In Figure 1, a pixel's color is based on the results of the triangle-to-pixel coverage test. The left image demonstrates the triangles to be rasterized, while the center image shows the sample coverage positions for each pixel. The final image, on the right, contains the results of the coverage test; if a rasterized triangle covered the center of a pixel, then that pixel is shaded to the triangle's color.

Half-Resolution Render with Upscaling

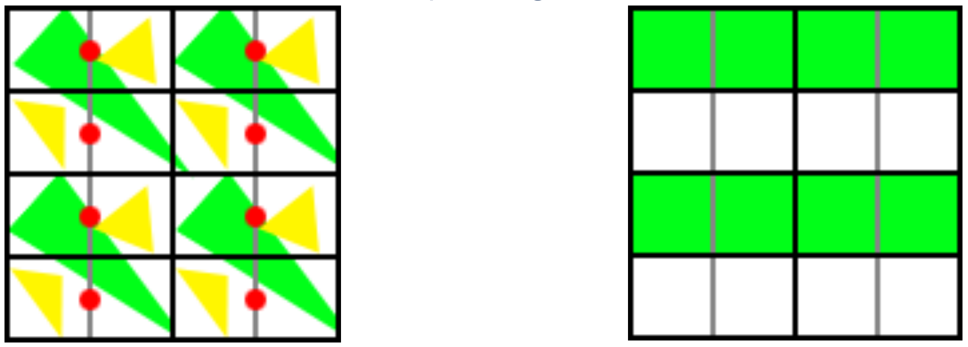


Figure 2: *Shading information is lost due to sample coverage positions; the light-gray outlines represent the pixels of a full-resolution render target, the black outlines represent the pixels of a half-resolution render target. Left: Triangles to be rasterized to the render target, the red dots represent the sample coverage position for each half-resolution pixel; notice only the green triangles pass the coverage test. Right: The shaded pixel colors based on the results of the coverage test; the yellow triangles failed the coverage test and the render target cannot be accurately reconstructed during upscaling.*

Upscaling techniques focus on reducing shading by decreasing the resolution so fewer pixels are shaded. However, as shown in Figure 2, if the triangle does not pass the pixel’s coverage test, the shading information will be lost. In the image on the left, the black outlines represent the pixels for a half-resolution render target. The yellow triangles failed to cover the center of the pixels, thus failing the coverage test; this means the full-resolution render target cannot be accurately reconstructed, as seen in the image on the right.

Half-Resolution Render with Temporal Reconstruction

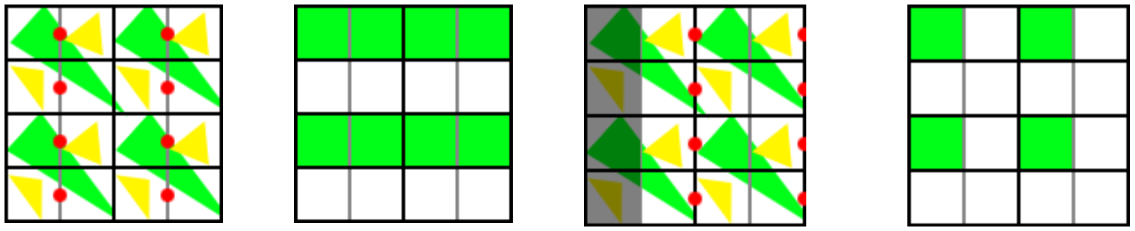


Figure 3: *Temporal reconstruction based on two frames; shading information is lost due to sample coverage positions. The light-gray outlines represent the pixels of a full-resolution render target, the black outlines represent the pixels of a half-resolution render target. From left to right: Triangles to be rasterized to the render target. The red dots represent the sample coverage positions for a half-resolution render target. Frame* N-1*: The shaded pixel colors based on the results of the coverage test. Notice only the green triangles passed the coverage test. Frame* N*: The viewport is jittered to the right. Notice that due to the placement of the sample coverage positions all of the triangles failed the coverage test. Reconstructed frame: The render target cannot be accurately reconstructed during upscaling.*

In an attempt to shade more pixels while keeping the resolutions reduced, temporal techniques utilizing shading data from previous frames were introduced. With these techniques, each frame alternates the render targets and jitters the viewport; the full-resolution render target is then reconstructed from render targets *N-1* and *N* (the previous and current frame render targets). As shown in Figure 3 above, the first two images represent the shading of frame *N-1*. The third image jitters the viewport to the right for frame *N.* Notice that the pixel coverage positions miss both the yellow and the green triangles. The image on the right shows that the full-resolution render target is not accurately reconstructed, similar to the non-temporal technique discussed in the preceding paragraph.

Improved Sample Coverage Positions with 2x MSAA

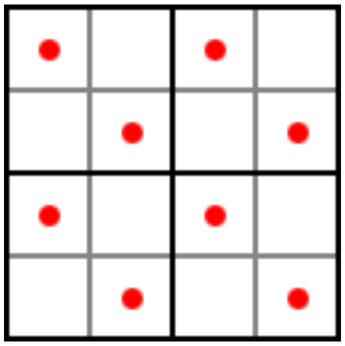


Figure 4: *2x MSAA shading information; the light-gray outlines represent the pixels of a full-resolution render target, while the black outlines represent the pixels of a quarter-resolution render target. The red dots represent the sample coverage positions for 2x MSAA.*

In an ideal scenario, rendering at a reduced resolution and jittering the viewport will place sample coverage positions at the exact locations as their full-resolution counterparts. Fortunately, the 2x MSAA standard sample coverage positions are at these locations; instead of using pixel centers for positions, the samples are positioned in the second and fourth quadrant of each quarter-resolution pixel, as shown in Figure 4.

Sample Coverage Positions with 2x MSAA and Temporal Reconstruction

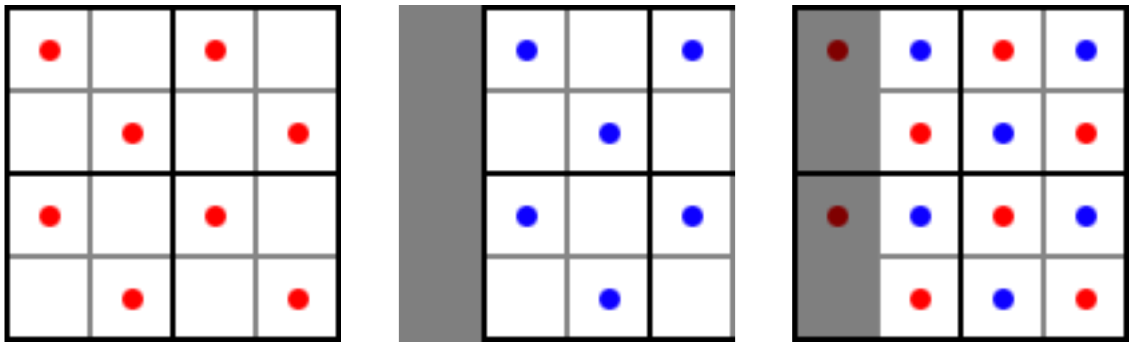
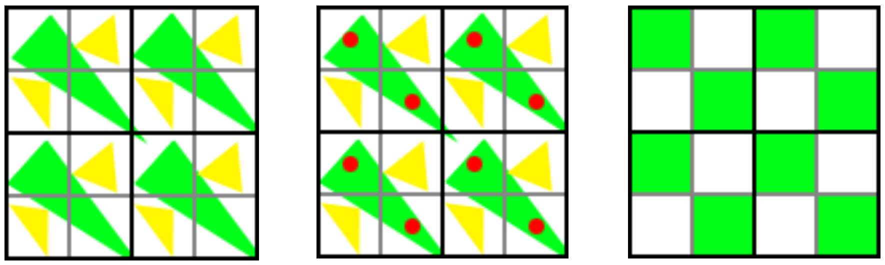


Figure 5: *2x MSAA shading information; the light-gray outlines represent the pixels of a full-resolution render target, the black outlines represent the pixels of a quarter-resolution render target. Left: The red dots represent the sample coverage positions for 2x MSAA during frame* N-1*. Center: The blue dots represent the sample coverage positions for 2x MSAA during frame* N*, which has been jittered one pixel to the right. Right: Temporally combining frames* N-1 *and* N *results in almost the same coverage as a full-resolution render with no MSAA; notice the first column of pixels is not reconstructed due to viewport jitter.*

It is possible to temporally jitter the 2x MSAA sample coverage positions so they overlap every coverage position of a full-resolution render (with the exception of the first and last pixel column). As shown in Figure 5, the interior locations, regions demarcated with grey lines within each black outline, are sample positions of a 2x MSAA surface. The left image shows the two sample positions from the 2x MSAA surface. In the next image, we shifted over to the right half-pixel in the quarter-sized image, which is a full pixel in the final render target. This works because movement by half a pixel in the quarter-resolution target is equivalent to a full pixel in the final target. The right image shows that by using the same fixed 2x MSAA sample pattern, we generate the sample points for a complete full-size render target by combining the results of the two half-resolution frames. Notice the checkerboard pattern that emerges from this technique—the full-resolution render target is almost perfectly reconstructed.

Accurate Full-Resolution Reconstruction using 2x MSAA with Temporal Reconstruction



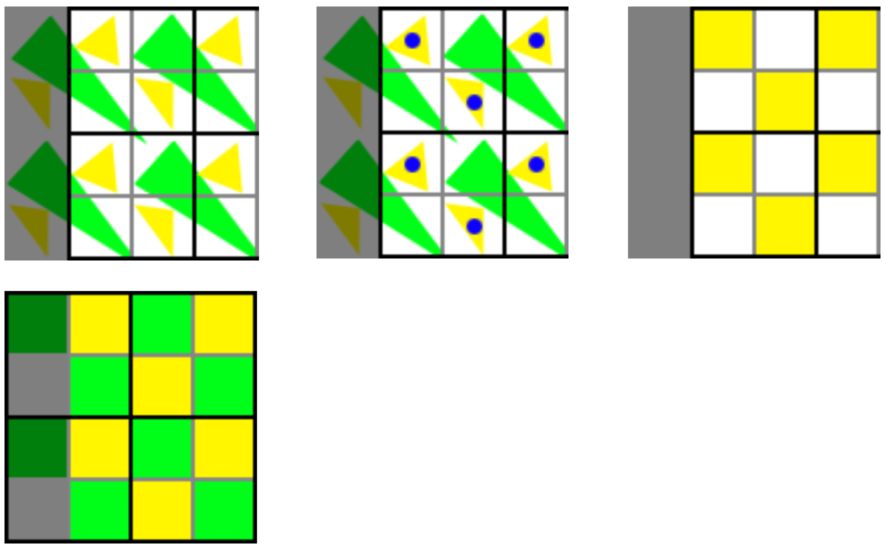


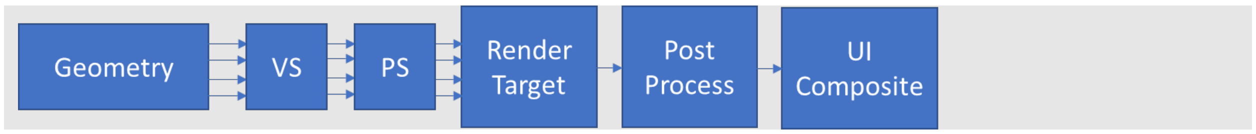
Figure 6: *Full-resolution render target using 2x MSAA with temporal reconstruction. The light-gray outlines represent the pixels of a full-resolution render target, the black outlines represent the pixels of a quarter-resolution render target. Top Left: Triangles to be rasterized to the render target. Top Center: Frame* N-1*, the red dots represent the 2x MSAA sample coverage positions. Top Right: The green triangles passed the coverage test and were shaded accordingly to frame* N-1*. Middle Left: The viewport is jittered one full-resolution pixel to the right. Middle Center: Frame* N*, the blue dots represent the 2x MSAA sample coverage positions. Middle Right: The yellow triangles passed the coverage test and were shaded accordingly to frame* N*. Bottom Left: The full-resolution render target is almost perfectly reconstructed by combining the results of frames* N-1 *and* N*; notice that the first column of pixels is not reconstructed due to viewport jitter.*

If we refer back to our original full-resolution render in Figure 1, by utilizing the 2x MSAA temporal reconstruction concept, it's possible to reconstruct a full-resolution render target from frames *N-1* and *N*, as demonstrated in Figure 6 above. The top row shows the green triangles accurately shaded due to 2x MSAA sampling positions. In the next row, the viewport is jittered one full-resolution pixel to the right, and the yellow triangles are now accurately shaded due to the jittered viewport and 2x MSAA sampling positions. The final image (bottom row) represents the reconstructed render target, which is a near 1:1 match with the full-resolution render target.

How to Modify the Renderer to Support CBR

Now that we have reviewed the motivation for the modifications to the render targets, in this section we describe the modifications to two simple canonical pipelines, one forward and one deferred, and show how CBR can be used in both cases. The forward renderer is the simpler case so we start with that, but deferred renderers are common in the industry so we wanted to demonstrate this as well.

Modifications to Forward Rendering to Support CBR



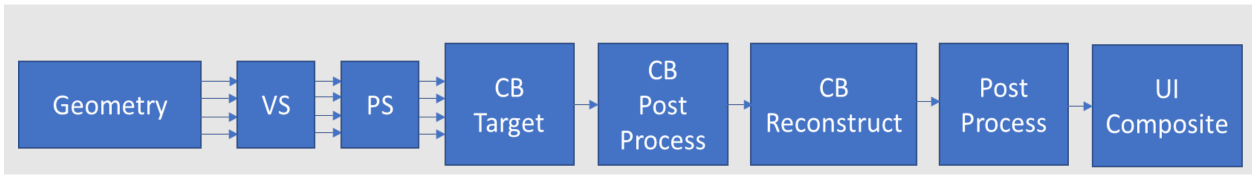


Figure 7: *Top, an overview of a forward shading pipeline. Bottom, a forward shading pipeline with a CBR implementation.*

The modifications to a forward rendering pipeline are straightforward, and we provide full sample code. The traditional forward pipeline is modified to alternate between two checkerboard quarter-resolution render targets. There is optional post processing on the checkerboard result as well as on the final stage of the CBR phase, where the final full-size render target is reconstructed. Figure 7 shows a high-level diagram of these changes. The results can be seen in Figure 8, below.

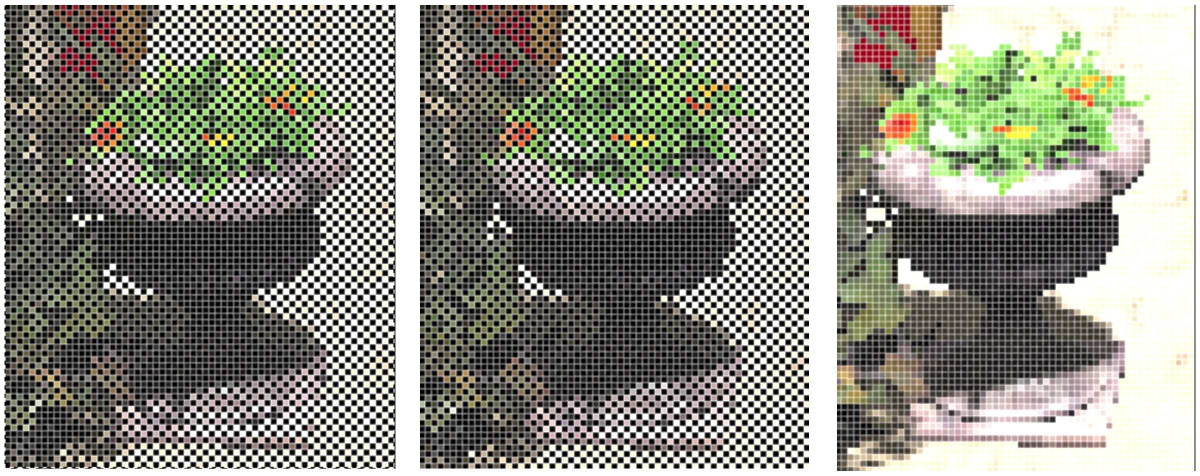


Figure 8: *Frame* N-1 *and frame* N *in a 2x MSAA buffer; shading is done in a checkerboard fashion. From left to right: Frame* N-1, *Frame* N *with viewport jitter, reconstructed render target.*

Forward Rendering Algorithm

We outline the steps for a modified forward rendering pipeline here:

1. Create two color and depth buffers (for frames *N-1* and *N*) that are one quarter of the desired resolution (width /2, height /2). These buffers should be set to use 2x MSAA with a shade per sample (as opposed to per pixel).
    a. In order to get increased texture resolution, a MIP LOD bias needs to be applied to textures. (MIP is acronym for the Latin phrase *multum in parvo*, meaning "much in little"; LOD stands for Level Of Detail). In Direct3D* 12, use a D3D12_SAMPLER_DESC MipLODBias of -0.5f during the 3D scene pass.
2. Render frame *N-1*.
3. Render frame *N* with the viewport jittered one full-resolution pixel in the quarter-resolution buffers to the right.

4. Run a checkerboard reconstruction shader that uses both frames to reconstruct a full-resolution render target.

Modifications to the Deferred Rendering Pipeline to Support CBR

CBR for a deferred pipeline is more complex than the forward rendering case. In a forward pipeline, both opaque and transparent objects are sorted and shaded at the time they are rendered. As demonstrated in the previous section, shading them to a checkerboard buffer requires only changing their render target. Here, we summarize a deferred rendering pipeline, and then describe the modifications to support CBR.

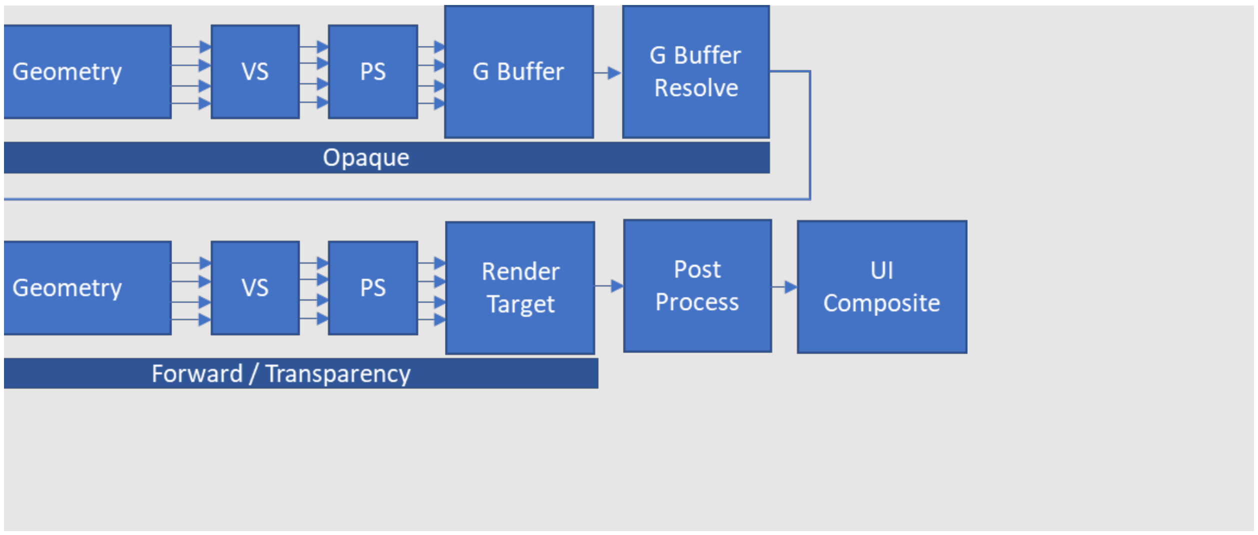


Figure 9: *A simplified overview of a deferred shading pipeline.*

With a deferred pipeline, the pixel color is determined through the use of three steps outlined in Figure 9, and detailed below:

1. Material information is rendered to separate render targets commonly referred to as the G-Buffer. In our example we have a G-Buffer that consists of three render targets: *albedo*, *normal*, and *specular*.
1. The G-Buffer is then used in a resolve step. The G-buffer resolve uses the render targets (albedo, normal, specular) to determine the pixel's shaded color, and writes the result to a render target.
2. Finally, objects that are better suited for forward rendering (for example, transparent objects) are shaded directly onto the render target.

Modifications to Support CBR

For our checkerboard deferred pipeline, the first modification we make is to Phase 1. The G-Buffer's render targets are created as checkerboard buffers: one quarter final resolution, with 2x MSAA enabled, as shown in Figure 10.

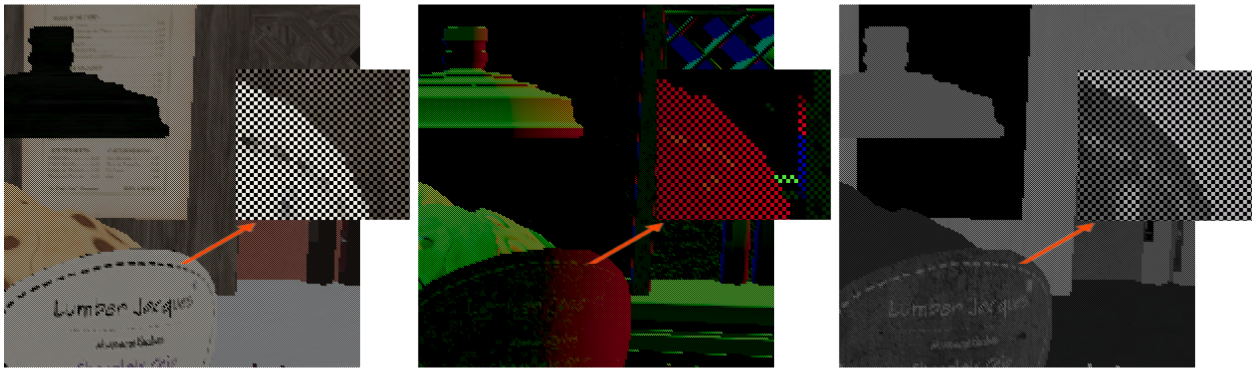


Figure 10: *Each G-Buffer render target is written as a 2x MSAA checkerboard buffer.*

In phase 1, the checkerboard buffers are rendered. In Phase 2, the G-Buffer resolve shader must be 2x MSAA aware and must output to a render target suitable for holding two shades per pixel. In our technique, we create a non-MSAA render target that is the same height as the checkerboard target, but twice the width. This allows us to shade and store each sample position at a unique texel. We refer to this texture as the Shade Resolve Target (SRT) and show an example in Figure 11.

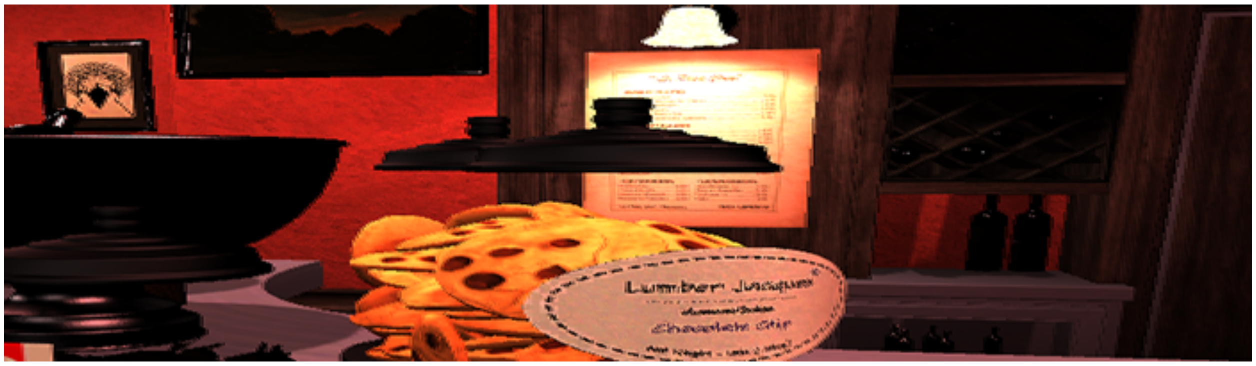


Figure 11: *The SRT is twice the width of our 2x MSAA targets and holds the shaded result of each 2x MSAA sample.*

Phase 3 must also be modified, as traditionally the forward shading step can blend directly to the resolved G-Buffer target. However, two issues prevent us from doing this:

1. Our SRT is twice the desired width.
2. We want to shade the transparency utilizing CBR, and the SRT is not an MSAA target.

We solve this issue by rendering the forward objects to a separate 2x MSAA buffer, which we call the Checkerboard Forward Buffer (CFB), as shown in Figure 12.

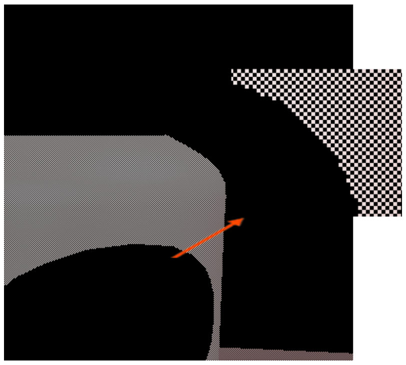


Figure 12: *The CFB stores all forward rendered objects along with their combined alpha values.*

We require a final checkerboard reconstruction step that reads the SRT and CFB buffers from frame *N-1* and frame *N*, and uses their shading to reconstruct a full-resolution render target (as seen in Figure 13).

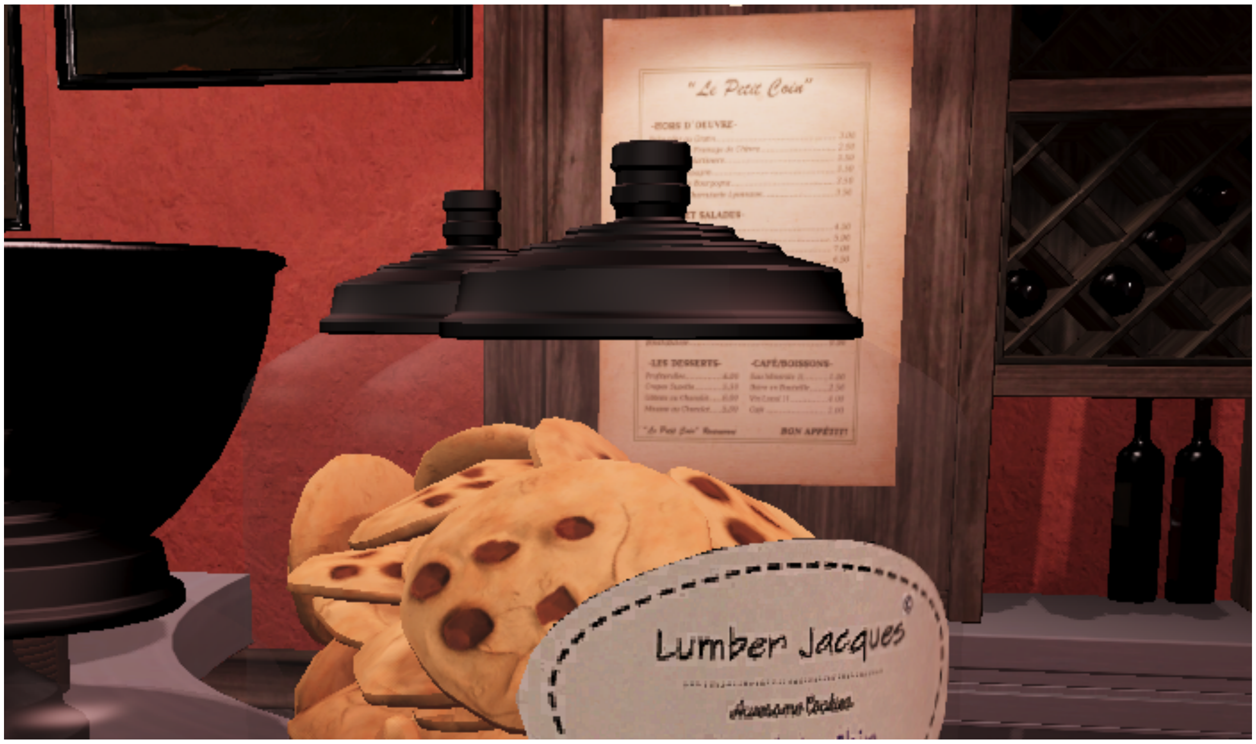


Figure 13: *Checkerboard reconstructed render target from our deferred shading implementation.*

Finally, we present a visual overview of the pipeline modifications in Figure 14. We include both the opaque and transparent rendering phase and the checkerboard reconstruction.

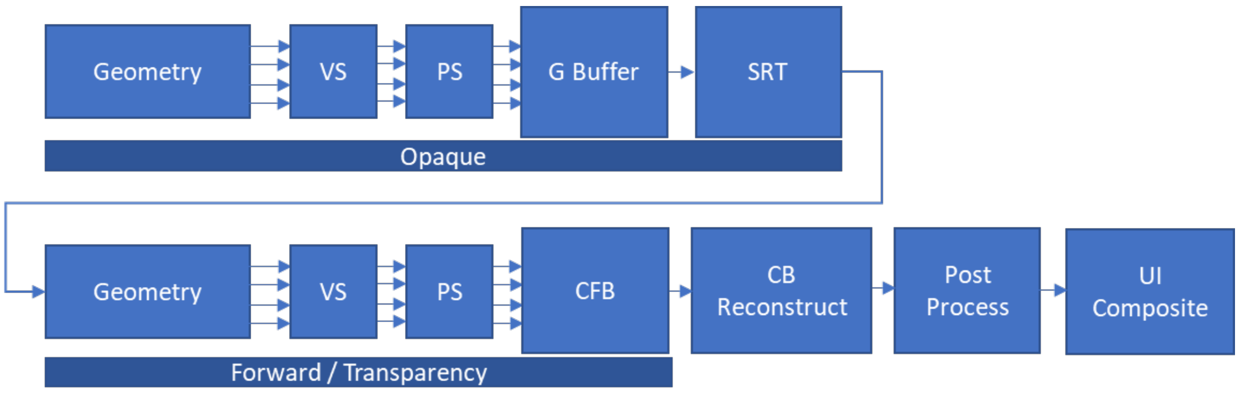


Figure 14: *A modified deferred rendering pipeline with support for CBR.*

Shading in Motion

In practice, reconstructing the full-resolution render target is more difficult. Games are not static, and each frame can be full of motion. Our checkerboard reconstruction step must account for a shaded element changing pixel position. Similarly, the required shading information may be missing due to it being occluded in the previous frame.

Adding the use of motion vectors to the checkerboard reconstruction pass is the first step in solving this problem. Motion vectors are used to track the movement between frames and adjust the pixel lookup of frame *N-1* during the reconstruction step; an example of this is provided in Figure 15. In our sample code, motion vectors are derived from the depth buffer, but this only works for static objects. In a real-world scenario, motion vectors for dynamic objects should be provided. We show an image of per pixel motion vectors mapped to the green color channel in Figure 16.

However, there is a complication to be aware of when providing motion vectors for dynamic objects. The motion vectors required for each pixel in frame N-1 will not have been rendered in the current frame *N*. We propose a couple of options to acquire this motion vector; it is left to the reader to choose a solution that best fits their scenario:

- The missing motion vector could be extrapolated based on the current motion vectors in frame *N* that surround it. This would most likely be acceptable in the majority of cases but could cause artifacts for slow-moving alpha-tested objects (such as slightly swaying vegetation).
- A *motion vector* pass, which sets the viewport to frame *N-1*'s jitter and renders the current motion vectors. This solution would update frame *N-1*'s motion vector buffer with the most recent data. Additionally, if the engine has a notion of *object IDs*, only the dynamic objects would need to be submitted, and the checkerboard reconstruction

could sample the motion vector when the object ID matched; otherwise the CBR could derive it from the depth buffer.

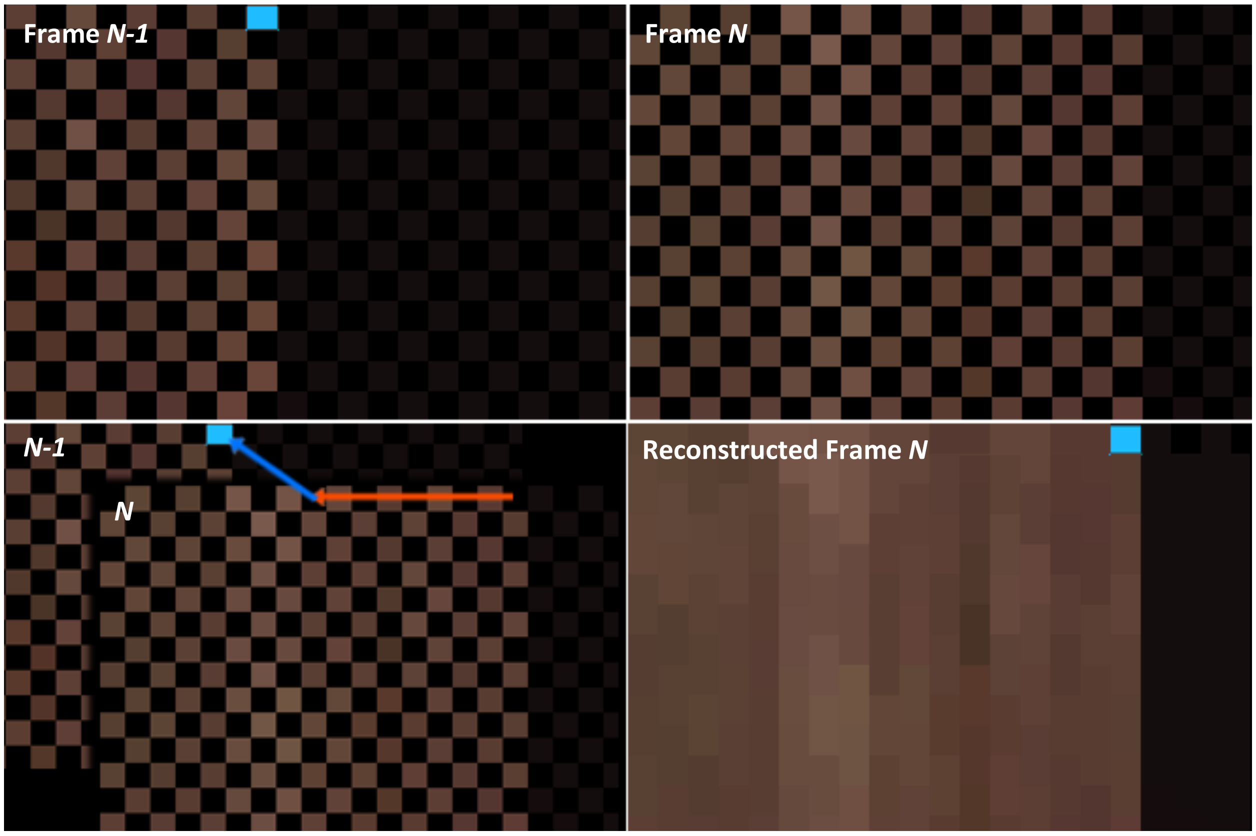


Figure 15: *How motion vectors are used during the reconstruction step, a blue square has been used to highlight an individual pixel for clarity. Top left: Frame* N-1*. Top right: Frame* N*, the camera moved a few pixels to the left. Bottom left: Frame* N *is shown overlaying Frame* N-1*, the reconstruction step is processing the image at the location of the blue pixel. The red arrow shows the motion between frame* N-1 *and frame* N*; this is used as an offset for the texel in frame* N-1 *(as shown by the blue arrow). Bottom right: The reconstructed render target based on frame* N-1 *and frame* N*.*

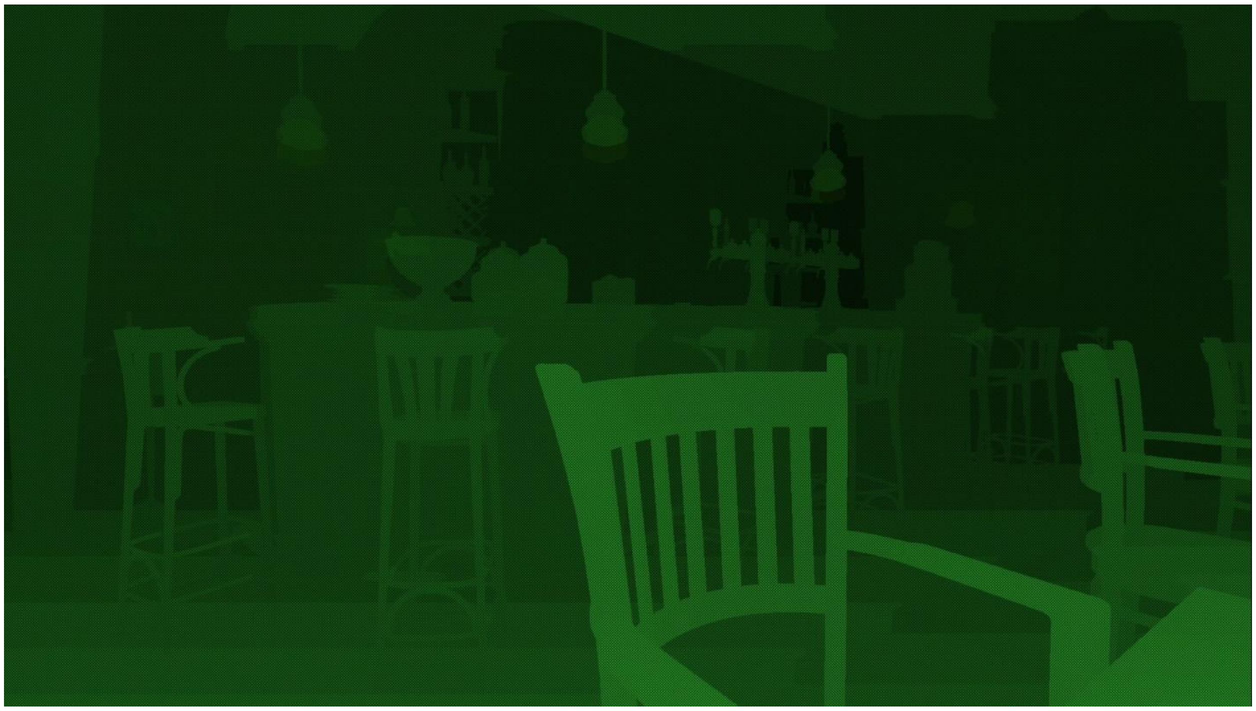


Figure 16: *Per pixel motion vectors derived from the depth buffer. The camera is translating up (positive Y), the magnitude of the Y motion per pixel is reflected in the green channel of the render target. In this scenario, foreground objects have larger motion vectors between frames.*

Missing Shading Information

A more complicated problem arises when the motion vectors point the reconstruction shader to shading information that doesn't exist in the previous frame. Consider the example in Figure 17:

- In frame *N-1*, a wine bottle is covering a barstool
- In frame *N*, the camera has moved to the left, revealing the barstool

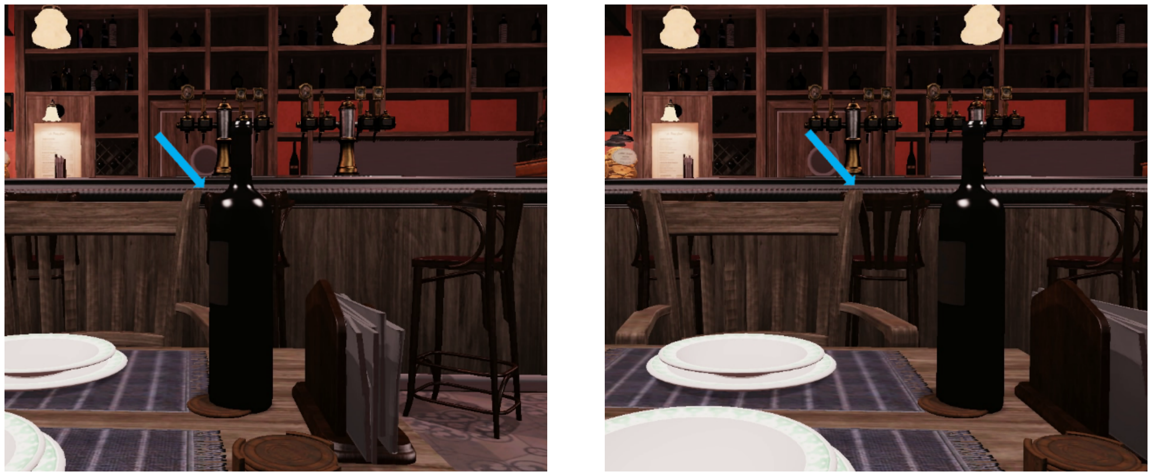


Figure 17: *Left: Frame* N-1, *a wine bottle covers a barstool. Right: Frame* N, *the camera moves to the left, revealing the barstool.*

When resolving the barstool's shading, the reconstruction shader reads the motion vectors that point to the right, in essence saying, "The barstool's shading in frame *N-1* is X pixels to the right." In reality, however, the wine bottle was covering the barstool, and attempting to blend those pixels during the reconstruction step results in *ghosted* artifacts, as shown in Figure 18. We need to detect this scenario and adjust the checkerboard reconstruction computation accordingly.

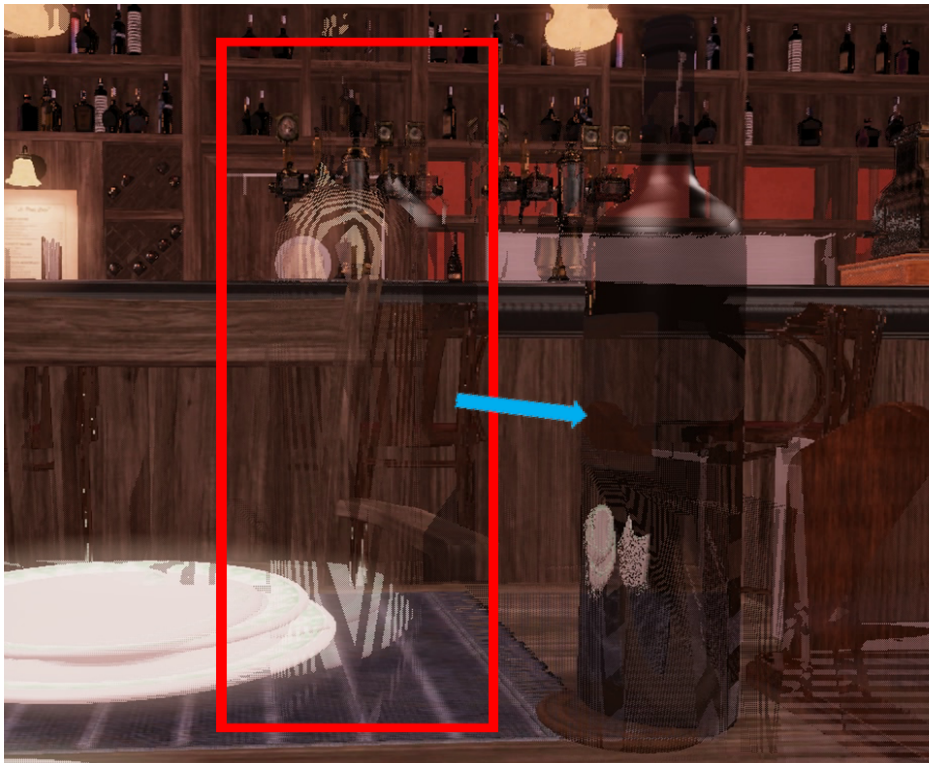


Figure 18: *The red outline highlights the ghosted artifacts that appear when resolving the barstool. This is due to the required shading information being occluded by the wine bottle in frame* N-1.

We provide two independent and straightforward solutions for detecting and solving missing shading information. The first approach compares linear depth values between frames *N-1* and *N* as shown in the code sample below (Extract 1). If the difference in depth values passes a minimum threshold, then it is assumed that the shading information is occluded as shown in Figure 19; for the purposes of this document we refer to it as the Check Shading Occlusion step (CSO).

The second solution takes a more basic approach and assumes occlusion for any shading information that moved more than a quarter-resolution texel; we refer to it as the Assume Shading Occluded step (ASO). While the latter approach is not as accurate as the former, in practice it requires fewer samples. The visual results are identical for shades that did not move and diverge minimally for shades that moved more than a quarter-resolution texel. Figure 20 shows an example of the divergence between CSO and ASO; notice the former retrieved the shading information, while the latter extrapolated an incorrect pixel color. More complicated solutions using object ID buffers, color comparison, and so on could be implemented. Readers can employ a more complex solution tailored to their pipeline, if required.

```
// If there is pixel motion between frames
if ( qtr_res_pixel_delta.x || qtr_res_pixel_delta.y )
{
   float4 current_depth;

   // Fetch the interpolated depth at this location in Frame N
   current_depth.x = readDepthFromQuadrant( qtr_res_pixel +
                     cardinal_offsets[ Left ], cardinal_quadrants[ 1 ] );
   current_depth.y = readDepthFromQuadrant( qtr_res_pixel +
                     cardinal_offsets[ Right ], cardinal_quadrants[ 1 ] );
   current_depth.z = readDepthFromQuadrant( qtr_res_pixel +
                     cardinal_offsets[ Down ], cardinal_quadrants[ 0 ] );
   current_depth.w = readDepthFromQuadrant( qtr_res_pixel +
                     cardinal_offsets[ Up ], cardinal_quadrants[ 0 ] );

   float current_depth_avg = (projectedDepthToLinear( current_depth.x ) +
                              projectedDepthToLinear( current_depth.y ) +
                              projectedDepthToLinear( current_depth.z ) +
                              projectedDepthToLinear( current_depth.w )) * .25f;

   // reach across the frame N-1 and grab the depth of the pixel we want
   // then compare it to Frame N's depth at this pixel to see if it's within range
   float prev_depth = readDepthFromQuadrant( prev_qtr_res_pixel, quadrant_needed );
   prev_depth = projectedDepthToLinear( prev_depth );

   // if the discrepancy is too large assume the pixel we need to
   // fetch from the previous buffer is missing
   float diff = prev_depth - current_depth_avg;
   missing_shading = abs(diff) >= tolerance;
}
```

Extract 1: *The CSO code step to determine if shading information in frame* N-1 *is missing. Compare the extrapolated depth value at the pixel in frame* N *with the actual depth value at the pixel in frame* N-1*; If it's greater than an empirically determined tolerance then assume the shading is occluded.*

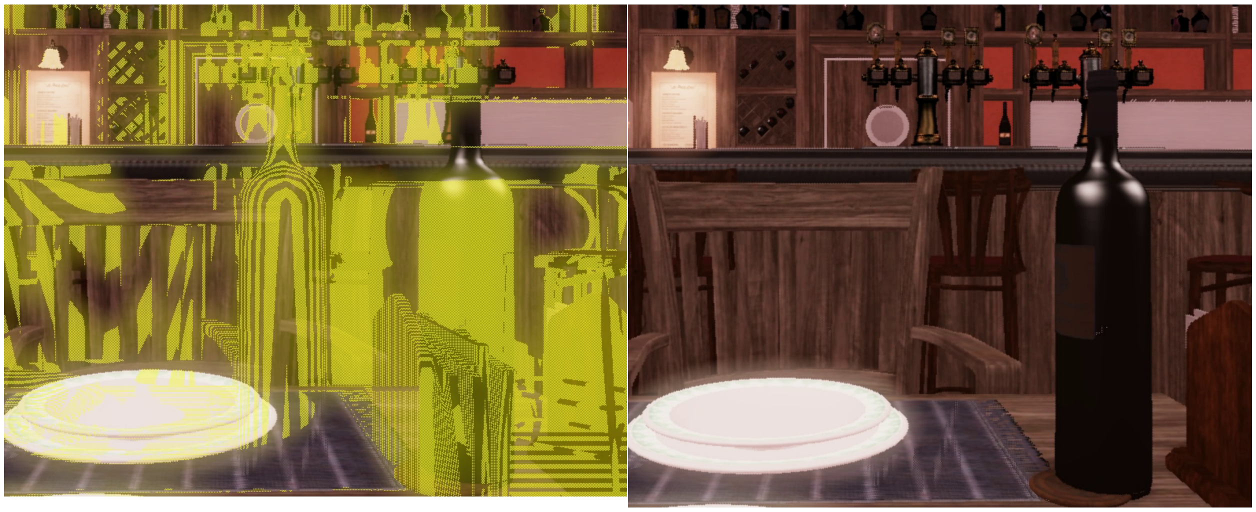


Figure 19: *Left: The camera jumped one quarter of the screen width between frame* N-1 *and frame* N. *Pixels in yellow highlight shading information that our CSO has determined to be missing. Right: The shades that were occluded are reconstructed using a blend of the current frame's surrounding pixels.*

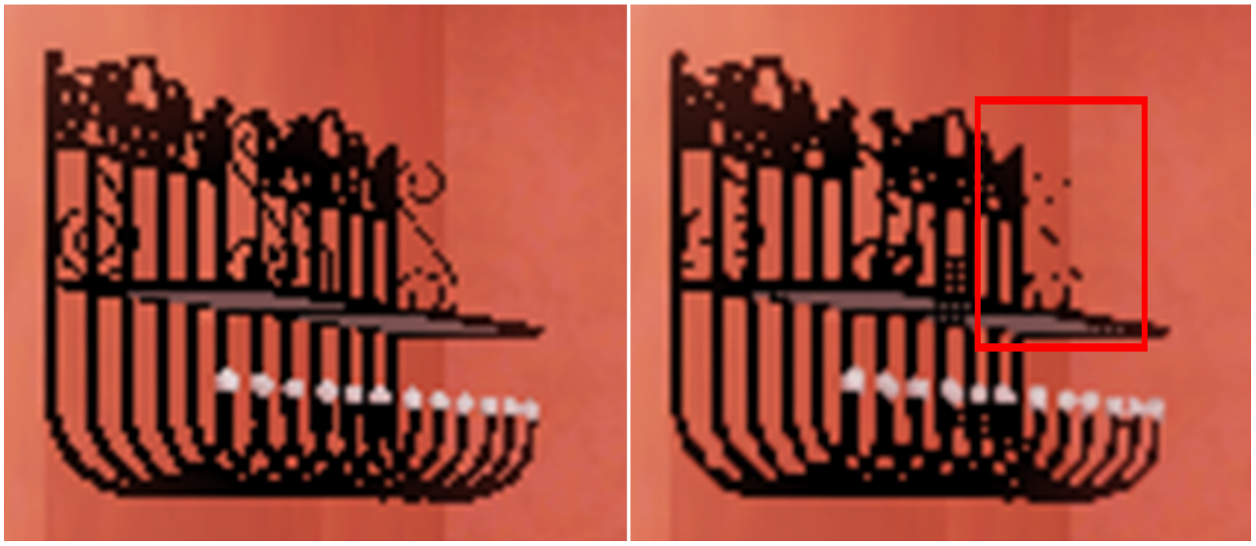


Figure 20: *Artifacts when using CSO versus ASO. Image magnified by 500 percent with no anti-aliasing for demonstration purposes. Left: CSO does not assume occlusion and attempts to retrieve the coat rack's shading using motion vectors. Right: ASO assumes occlusion and uses extrapolation to fill in the coat rack's shading. Notice it incorrectly uses shades from the wall and some of the coat rack shades are lost (as highlighted by the red outline).*

Checkerboard Reconstruction Shader Pseudo Code

The pseudo code for our checkerboard reconstruction shader is as follows (and is also shown as abbreviated HLSL (High-Level Shading Language) in Extract 2):

For each pixel:

1. Was this pixel rendered with frame *N* (that is, the most recent frame)?
    a. If so, sample from frame *N* and exit.

    b. Otherwise, we need the pixel from frame *N-1*, so proceed with the algorithm.
2. Check the motion.
    a. If there is no motion, sample from frame *N-1* and exit.
    b. Otherwise, we need to determine the sub-pixel location for the shading in frame *N-1.*
3. Apply motion vectors to determine the sub-pixel location in Frame *N-1.*
    a. If the camera moved in such a way that the sub-pixel location in frame *N* overlaps frame *N-1* (that is, it cancelled the jitter effect) then the shading information is not available. Perform a blend using frame *N*'s pixels in the cardinal directions and exit.
    b. Otherwise, proceed with the algorithm.
4. If the user wants CSO.
    a. Sample the depth in frame *N-1* (using the motion vectors as offsets).
    b. Average the surrounding depths in frame *N* (cardinal directions).
    c. Compare the two depths.
    d. If the difference in depths passes a minimum threshold, assume that the required shading is occluded.
5. If the shading is occluded or the user chose ASO, perform a blend using frame *N*'s pixels in the cardinal direction.
6. Otherwise, sample the sub pixel sample from frame *N-1* (using the motion vectors as offsets) and exit.

```
// if the pixel we are writing to is in a MSAA
// quadrant which matches our latest CB frame
// then read it directly and we're done
if ( frame_quadrants[ 0 ] == quadrant || frame_quadrants[ 1 ] == quadrant )
   return readFromQuadrant( qtr_res_pixel, quadrant );
else
{
   // We need to read from Frame N-1

   ...

   // Get the screen space position this pixel was rendered in Frame N-1
   uint2 prev_pixel_pos = ...

   // Which MSAA quadrant was this pixel in when it was shaded in Frame N-1
   uint quadrant_needed = ...

   ...

   // if it falls on this frame (Frame N's) quadrant
   // then the shading information is missing
   // so extrapolate the color from the texels around us
   if ( frame_quadrants[ 0 ] == quadrant_needed ||
        frame_quadrants[ 1 ] == quadrant_needed )
      missing_shading = true;
   else if ( qtr_res_pixel_delta.x || qtr_res_pixel_delta.y )
   {
      // Otherwise we might have the shading information,
      // Now we check to see if it's occluded

      // If the user doesn't want to check
      // for occlusion we just assume it's occluded
      // and this pixel will be an extrapolation of Frame N's pixels around it
      // This generally saves on perf and isn't noticeable
      // because the shading will be in motion
      if ( false == check_shading_occlusion )
         missing_shading = true;
      else
      {
         ...

         // if the discrepancy is too large assume the pixel we need to
         // fetch from frame N-1 is missing
         float diff = prev_depth - current_depth_avg;
         missing_shading = abs(diff) >= tolerance;
       }
     }

     // If we've determined the pixel (i.e. shading information) is missing,
     // then extrapolate the missing color by blending the
     // current frame's up, down, left, right pixels
     if ( missing_shading == true )
        return colorFromCardinalOffsets( qtr_res_pixel,
                                         cardinal_offsets,
                                         cardinal_quadrants );
     else
        return readFromQuadrant( prev_qtr_res_pixel, quadrant_needed );
}
```

Extract 2: *Abbreviated HLSL code demonstrating the pseudo-code routine listed above. Quadrant refers to the MSAA sample location within a quarter-resolution pixel.*

Visual Results

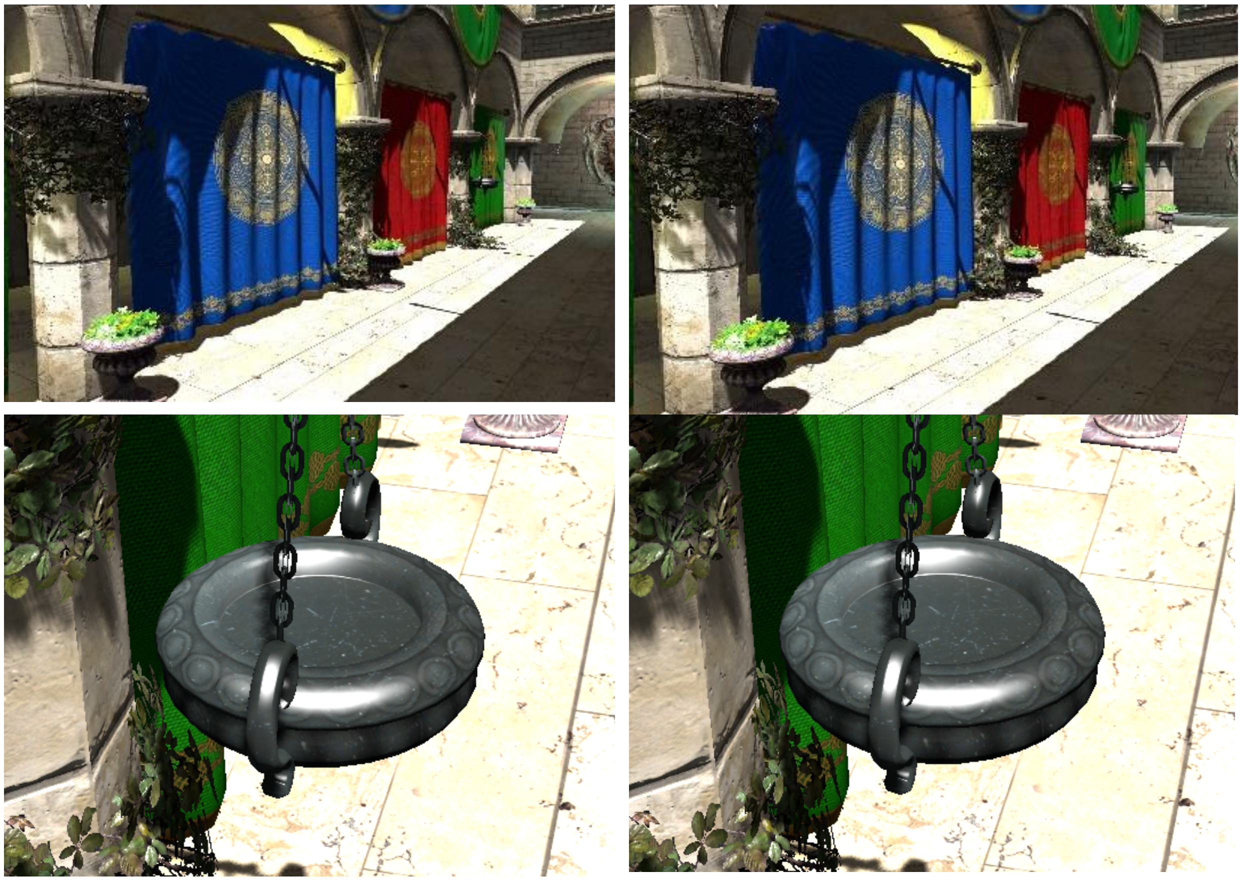


Figure 21: *Top left, full-resolution rendering of the Atrium Sponza Palace* model hallway. Top right, quarter-resolution rendering of the hallway with checkerboard reconstruction. Bottom left, full-resolution rendering of a pot. Bottom right, quarter-resolution rendering of the pot with checkerboard reconstruction.*

As shown in Figure 21, for static scenes or scenes with movement crossing multiple pixels, the results subjectively rival full resolution. The former, because there is no temporal shading information missing during render target reconstruction; the latter, because dramatic motion results in a lack of frame-to-frame coherence, and the eye does not perceive extrapolation errors during the reconstruction. However, with subtle movement, some visual artifacts are apparent on primitive edges with high color contrast, similar to the *edge crawling* seen in traditional aliasing (Figure 22). Popular anti-aliasing solutions, including CMAA, FXAA, and TAA, remove or greatly alleviate these artifacts.

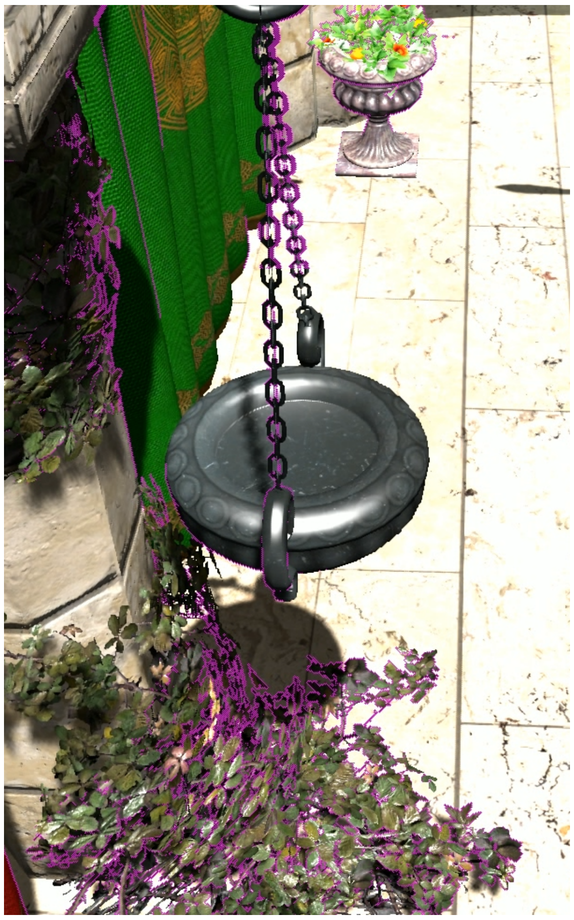
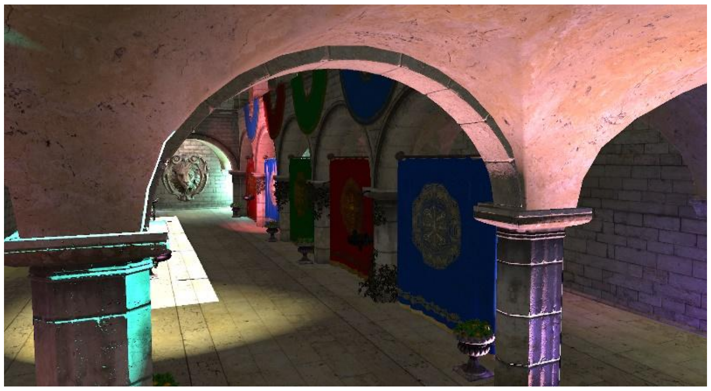
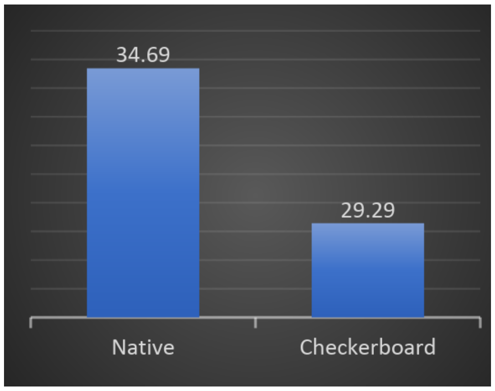
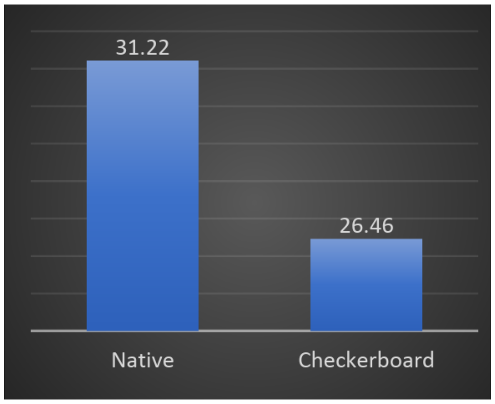


Figure 22: *Subtle camera movement. Pixels in purple represent shading information that was occluded in frame* N-1 *and require extrapolation; this can lead to additional edge crawling.*

Performance Results

We compared the performance of CBR to full resolution with all images rendered at 1080p. As seen in Figure 23 below, CBR reduced frame time by approximately 5ms in scenes with motion below a quarter-resolution texel. For scenes with greater motion, we tested both reconstruction solutions—ASO and CSO—over time with a camera fly-through of the Atrium Sponza Palace* model scene. The former approach averaged a 15 percent performance increase over full resolution as shown in Figure 24; the latter averaged a 12 percent performance increase as shown in Figure 25. The CSO's slight decrease in performance gains are due to the cost of the occlusion checks in frames, which do not benefit from the reduced shading rate (that is, frames that are not bottlenecked by the pixel shading stage).



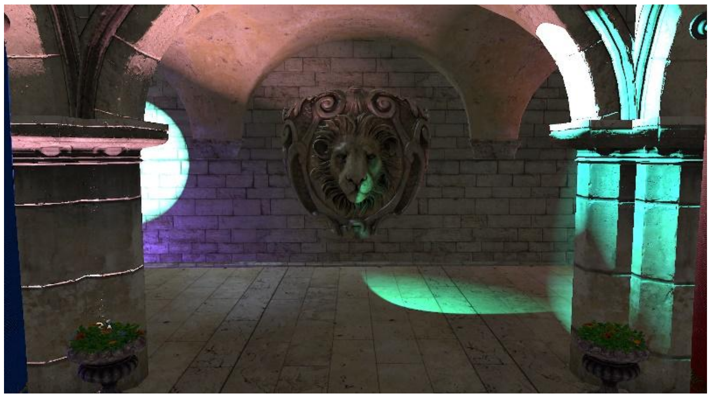
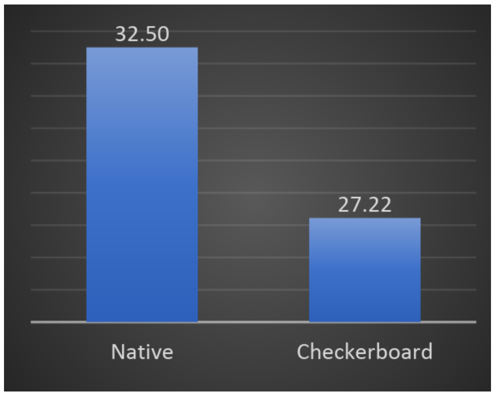



Figure 23: *Graphics shown are GPU frame time in milliseconds; frames are rendered at 1920 x 1080 with no post effects applied (AA, SSAO, and so on).*

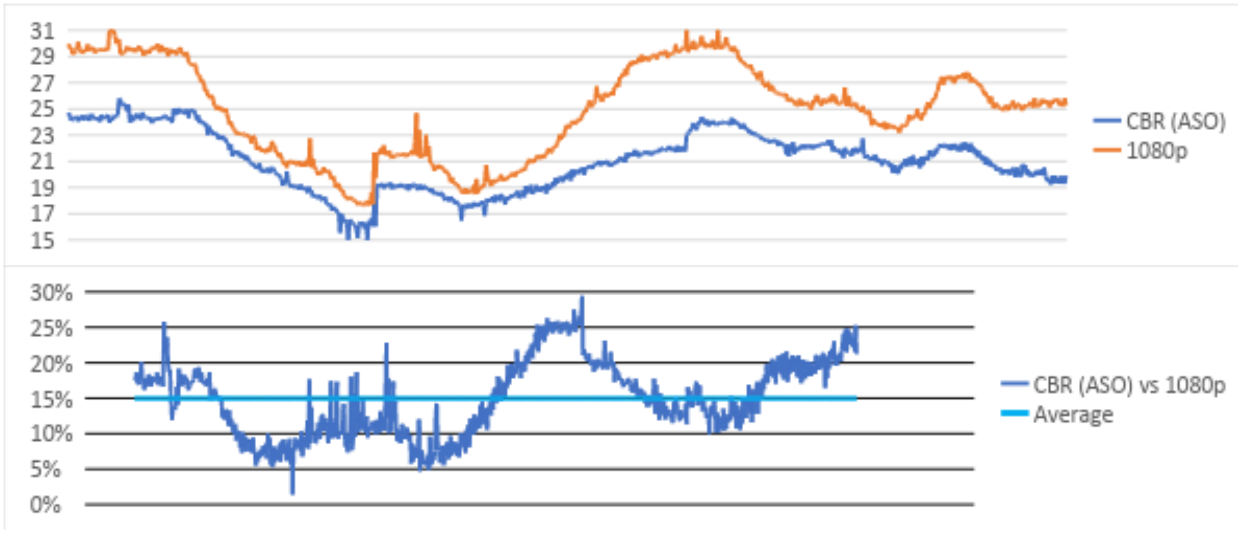


Figure 24: *Performance comparisons of CBR with ASO versus 1080p. Top: GPU frame time in milliseconds (lower is better). Bottom: The percentage performance increase of CBR versus 1080p (higher is better).*

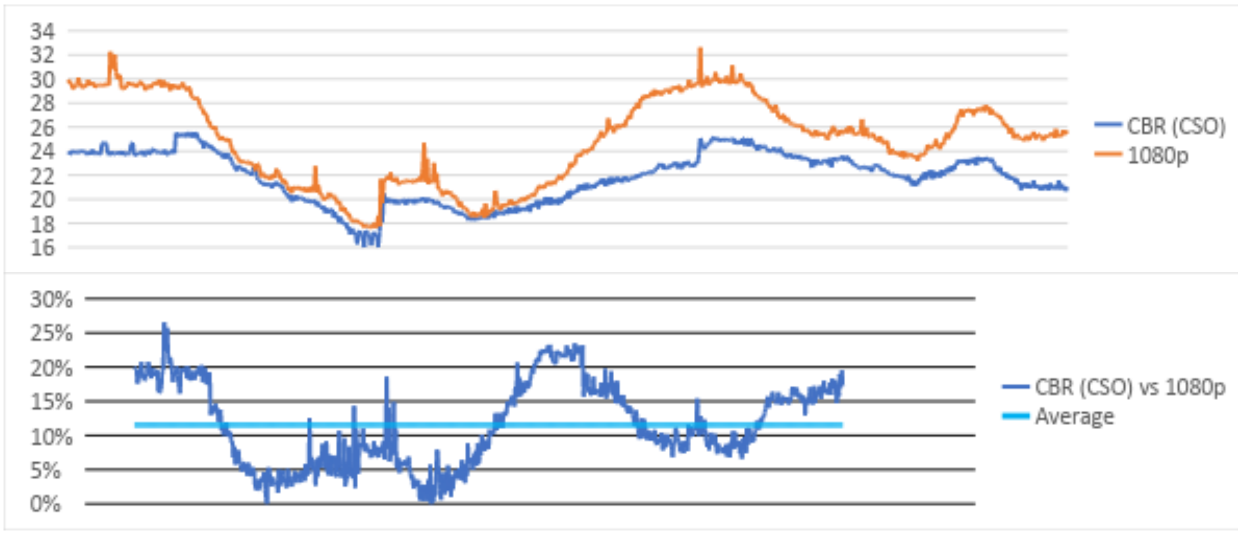


Figure 25: *Performance comparisons of CBR with CSO versus 1080p. Top: GPU frame time in milliseconds (lower is better). Bottom: The percentage performance increase of CBR versus 1080p (higher is better). Notice the areas with minor performance increases are geometry bound and the increased cost of the CSO occlusion checks are apparent.*

Future Work

We have a few ideas we'd like to explore in a future implementation, and we'll mention two of them here.

First, we'd like to combine dynamic resolution rendering and CBR in a single implementation. The Frostbite engine and the Sony PS4 Pro game *Deus Ex: Mankind Divided* (both mentioned earlier in the article) have done this, and we find it useful to combine the techniques.

Second, we think it would be a natural fit to combine a post-process anti-aliasing solution within the checkerboard reconstruction shader. Both techniques rely on combinations of color or depth buffer sampling; theoretically they could be integrated within the same shader and reduce the cost of a separate anti-aliasing pass.

Conclusion

The technique outlined in this white paper presents a straightforward solution for integrating CBR into existing forward or deferred shading pipelines. For workloads that are heavily bound by the pixel shader stage, we've seen CBR reduce frame times by up to 30 per cent, versus full resolution. We believe the relative ease of implementation, scalability across multiple resolutions, and compatibility across GPU vendors make it an attractive solution for developers looking to get the most out of their GPU.

Acknowledgements

The authors want to thank numerous people who contributed to the technical work or content reviews of this paper and the accompanying sample code. Kai Xiao and the rest of the Intel ART team were helpful in discussions and sharing the initial checkerboarding source code to get us started; Stephen Junkins provided early reviews and championed the work, and Marissa du Bois was very helpful with handling internal overhead to get sample code out the door. We want to thank Amazon for the Lumberyard Bistro* model, NVIDIA for hosting the Open Research Content Archive (ORCA), and Crytek for the updated Atrium Sponza Palace model.

Sample Code

We have implemented the techniques discussed in this paper into a fork of the DirectX12 Mini-Engine using Shader Model 5. The fork includes instructions to build and run a working sample and is available at https://github.com/GameTechDev/DynamicCheckerboardRendering.

References

Amazon Lumberyard Bistro, Open Research Content Archive (ORCA). https://developer.nvidia.com/orca/amazon-lumberyard-bistro.

Original Intel Developer Zone Article by Doug Binks. https://software.intel.com/en-us/articles/dynamic-resolution-rendering-article, July 13, 2011.

Giliam de Carpentier and Kohei Ishiyama. *Decima: Advances in Lighting and AA.* http://advances.realtimerendering.com/s2017/DecimaSiggraph2017-final.pptx. SIGGRAPH 2017 Advances in Real-Time Rendering.

Crytek Sponza Model. http://www.crytek.com/cryengine/cryengine3/downloads/.

Jalal El Mansouri. Rendering *Rainbow Six Siege*. GDC 2016. https://www.gdcvault.com/play/1022990/Rendering-Rainbow-Six-Siege.

High-Quality Temporal Supersampling, by Brian Karis. http://advances.realtimerendering.com/s2014/. SIGGRAPH 2014 Advances in Real-Time Rendering in Games.

Inside PlayStation Pro 4 Pro: How Sony made the first 4K games console, by Richard Leadbetter. Interview with Mark Cerny, PS4 Pro Architect. https://www.eurogamer.net/articles/digitalfoundry-2016-inside-playstation-4-pro-how-sony-made-a-4k-games-machine. October 20, 2016.

Timothy Lottes. *FXAA.* February 2009. http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf

Filip Strugar. *Conservative Morphological Anti-Aliasing (CMAA) – March 2014 Update.* https://software.intel.com/en-us/articles/conservative-morphological-anti-aliasing-cmaa-update. March 18, 2014.

Graham Wihlidal. *4K Checkerboard in Battlefield 1 and Mass Effect: Andromeda.* https://www.ea.com/frostbite/news/4k-checkerboard-in-battlefield-1-and-mass-effect-andromeda. . GDC 2017.

Notices

§ Configurations: System Manufacturer and Model: Skull Canyon: Windows 10 Pro 64-bit (10.0, Build 17134) (17134.rs4_release.180410-1804), Intel(R) Core(TM) i7-6770HQ CPU @ 2.60GHz, 16384MB RAM, Intel Iris Pro Graphics 580, 8264 MB Display Memory, Driver Version: 24.20.100.6136.

§ For more information go to http://www.intel.com/performance.