Eye-Based Point Rendering for Dynamic Multiview Effects

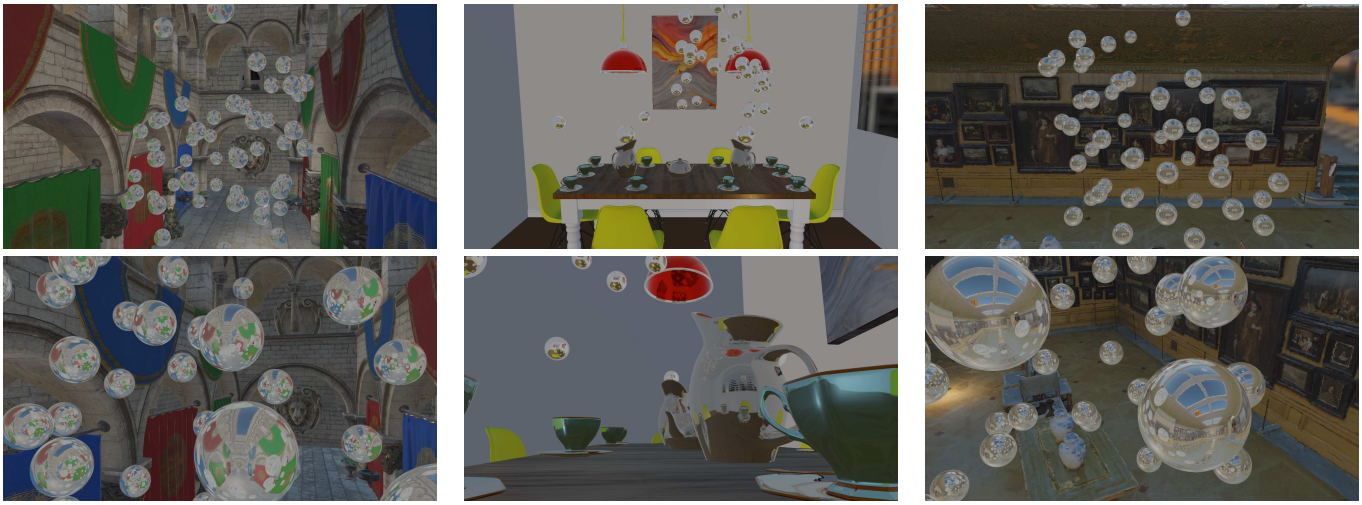


Fig. 1. *300 cube map reflections (50 cubemaps) generated by eye-based point rendering (EPR) for the sponza, breakfast, and gallery scenes [28]; with camera distant showing single reflection (top row) and camera close showing two recursive reflections(bottom row) with a single pass over the geometry. EPR renders reflections up to* 6.7 *times faster than multi-pass rendering, and up to* 7.2 *times faster than improved view independent rendering (iVIR).*

Eye-based point rendering (EPR) can make multiview effects much more practical by adding eye buffer resolution efficiencies to improved view-independent rendering (iVIR). We demonstrate this very successfully by applying EPR to dynamic cube-mapped reflections, sometimes achieving nearly 7× speedups over iVIR and traditional multiview rendering (MVR), with nearly equivalent quality. Our application to omnidirectional soft shadows is less successful, demonstrating that EPR is most effective with larger shader loads and tight eye buffer to off-screen buffer mappings. This is due to EPR's eye buffer resolution constraints limiting points and shading calculations to the sampling rate of the eye's viewport. In a 2.48 million triangle scene with 50 reflective objects (using 300 off-screen views), EPR renders environment maps with a 49.40*ms* average frame time on an NVIDIA 1080 Ti GPU. In doing so, EPR generates up to 5× fewer points than iVIR, and regularly performs 50× fewer shading calculations than MVR.

CCS Concepts: • **Computing methodologies** → **Rendering**; *Graphics processors*; *Point-based models.*

ACM Reference Format:
. 2022. Eye-Based Point Rendering for Dynamic Multiview Effects. 1, 1 (December 2022), 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In computer graphics, indirect effects such as coherent reflections and soft shadows have been difficult to perform on GPU hardware, which often requires rendering multiple views each needing a pass over scene geometry, and each creating significant shading loads.

Author's address:

View independent rendering (VIR) [27] avoids multiple passes by generating and rendering points in real time. Frame by frame, it converts the model's triangles into a set of points fit to the views the current frame needs. With the points, VIR then renders views in parallel, requiring nearly an order of magnitude fewer passes. However, if off-screen views are not similar, the number of points needed to fill every off-screen buffer pixel rises, increasing the cost of each pass over VIR's point set, and reducing VIR's advantage. Gavane et al. therefore introduced iVIR [12], which lessened this problem by resizing off-screen buffers to match the resolution needed in by the eye-buffer. This enabled rendering of dynamically environment mapped scenes at 2-4× the speed of MVR.

In this paper, we introduce EPR, which innovates by adding the the following improvements to iVIR:

- *EPR limits point cloud size* to eye-buffer resolution, making the point set up to 5× smaller than VIR's. This is enabled by splatting points across several pixels in off-screen buffers.
- *EPR limits shader loads* by deferring lighting of all off-screen pixels until they are reflected into an eye-buffer pixel, regularly making shader loads 50× or more smaller than MVR's. To enable this, each off-screen buffer becomes an indexed G-buffer, referencing the needed shading data in the point set. Moreover, this indirection allows similarly lazy shading of recursive reflections *without additional rendering passes*, magnifying shader speedups further. We call this *eye-based deferred shading*. In principle, this should also be possible in more traditional renderers, as long as they limit memory use by indexing off-screen buffers.

These innovations enable rendering of dynamic cube-mapped reflections nearly 7× faster than iVIR and multiview rendering (MVR), at nearly the same image quality. Speedups of omnidirectional shadows are minor, revealing some of EPR's limitations: significant efficiencies require meaningful shader loads (unlike shadow maps), and fairly tight mappings of pixels in the eye buffer to off-screen buffers. Indeed, in shadow mapping, there is no shading at all, and so no eye-resolution limits to shader loads at all!

2 RELATED WORK

Realistic rendering requires precise modeling of light flow by sampling Kajiya's rendering equation [22]. This can be difficult, especially for effects like soft shadows, depth of field, motion blur and indirect reflections [25]. MVR is often the best way to achieve this with rasterizers: putting multiple views into off-screen buffers and filtering them together when shading. But MVR is slow, with real-time constraints regularly requiring subsampling [38]. Various MVR pipelines have been studied (e.g., [8, 19–21, 39]), but even the best of these slows significantly when generating more than 32 views. Recent ray-tracing hardware has enabled non-MVR solutions, hybrid rasterization-ray tracing pipelines for computing effects including reflections, refractions, and shadows [4, 35].

2.1 Reflections and Environment Mapping

Reflections add insight about a surface's materials and environment, and can be achieved using ray tracing [40]. However, ray-traced reflections can alias, particularly during animation. Avoiding aliasing either requires more samples or hybridized solutions using rasterized reflections [2, 4]. Environment mapping is the best-known rasterized solution. It turns images of the environment around each reflective object into textures that are accessed using view-dependent lookups. There are several techniques, most varying the mapping between textures and the reflective surface [5, 10, 18, 29, 42]. Perhaps the most widely used is cube mapping [14], because it fits well in rectangular perspective buffers, permitting interactive reflections.

Environment mapping produces convincing reflections, but if the scene is dynamic, textures become out of date and must be updated each frame. Also, if an object is close to the reflective surface, different reflective objects will "see" different views of that object, and so they cannot share the same environment map. These shortcomings require extensive multiview rendering in each frame, limiting the use ofenvironment mapping in real time.

2.2 Shadows and Shadow Mapping

Shadows help viewers understand the spatial relationships between objects. Rasterizers render the "hard" shadows cast by point lights using shadow mapping [41], which compares the eye's view and the light's view to determine what is in shadow; and shadow volumes [7], which geometrically models shadow boundaries and finds the objects intersecting them. More natural "soft shadows" are cast by realistic light sources with area, instead of imaginary point lights. Both shadow mapping [9, 17] and shadow volume [1] soft shadow solutions exist, but both introduce unfortunate tradeoffs between speed and accuracy. Fast approximations [3, 16] compromise shadow quality to maintain real-time speed, such as percentage-closer soft shadows (PCSS), which renders single-view hard shadows, and then blurs their boundaries [11, 31].

Most real-world lights emit in many directions. This omnidirectional lighting poses a challenge for current shadow algorithms, which remain efficient by restricting light (shadow) flow to a limited range of directions. One rasterized solution creates six shadow buffers on a cube surrounding each point sample of the the light source [13, 24].

2.3 Points and iVIR

EPR avoids many rasterization constraints by using points as a rendering primitive [15, 26]. Unfortunately when views change, points can reveal incorrectly modeled gaps in surfaces. Preventing such artifacts often requires a prohibitive number of points, or similarly prohibitive filtering of sparser point sets (e.g., [34]). However, newer algorithms can avoid such painful tradeoffs. Ritschel et al. [32–34] adaptively sample according to brightness and view. Schutz et al. [37] apply compute shaders to render points more quickly.

Marrs et al's VIR [27] innovates by using the GPU rasterizer to create a new set of points for every frame, customized to that frame's off-screen views. It then renders views in parallel with nearly an order of magnitude fewer geometry traversals than MVR. VIR ensures that at least one point reaches every pixel in off-screen buffers. When views captured in these buffers are self-similar (as in directional soft shadows), resulting point clouds remain tractably small. But when views are heterogeneous (as with environment maps or omnidirectional shadows), point clouds grow, making real-time performance challenging, particularly when shader loads are large. To mitigate this problem, Gavane and Watson's iVIR [12] more efficiently samples triangles and shrinks off-screen buffers to match eye buffer resolution, reducing the number of points and shader loads. This enables application of iVIR to environment mapping with $2-4\times$ speedups over MVR.

3 THE EPR RENDERING PIPELINE

iVIR did improve VIR, but speedups were modest. EPR improves speeds further with *eye-resolution* reductions in point cloud size and shader loads. This section focuses on reductions in point cloud size, since *all* shading (including shading of off-screen buffers) is deferred until the final deferred eye pass.

Figure 2 shows the EPR pipeline. Briefly, the vertex shader determines whether a triangle is visible and therefore sampled at all. The geometry shader sets the density with which a triangle will be sampled by points, based on eye-resolution constraints. The rasterizer produces those points, with fragments treated as points moving through the fragment shader into storage buffers. The compute shader splats those points into off-screen view buffers, which are in fact index G-buffers referencing a point's shading data in the point buffers.

We next discuss each pipeline stage in more detail, highlighting *eye-resolution* computational constraints.

3.1 The vertex shader computes visibility

Geometry should only be rendered if it might be visible to the eye. We begin this visibility determination in the vertex shader. After

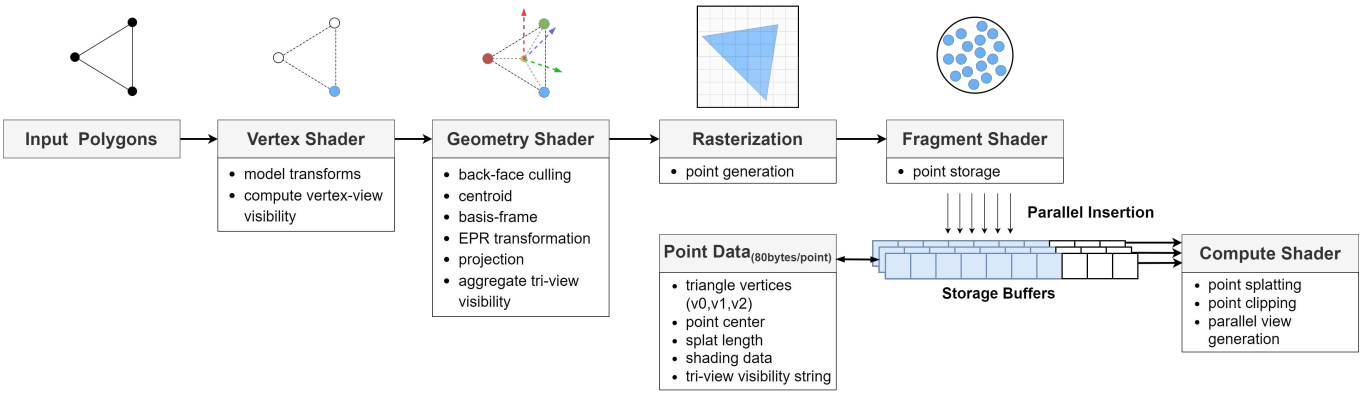


Fig. 2. *Eye-based point rendering (EPR) pipeline as implemented in the GPU*

applying the model transformation, we conservatively compute the visibility of the vertex in each of the views to be rendered, creating a bit string for use later in the pipeline. Because the vertex-view visibility string will be used to determine the visibility of entire triangles, we compare the vertex against view frusta that have been increased in size by 20%.

3.2 The geometry shader sets point cloud density

Ideally, a point cloud should be just dense enough to enable a fully detailed rendering in the eye's current view, and no more. Because EPR recreates its point cloud in every frame, it can aggressively optimize for the current view, ensuring only that the point cloud enables full detail in all of the current frame's off-screen views. It need not concern itself with views in future (largely unknown) frames.

When determining point cloud density, EPR ensures that every pixel in the eye's view is sampled at least once across all off-screen views. This *eye-resolution* sampling is a significant departure from iVIR, which ensures instead that every pixel in every off-screen buffer is sampled at least once. EPR's point clouds will typically be much less dense than iVIR's, because they need only ensure that the eye's view is fully detailed, rather than every off-screen view. (On the rare occasions that the eye's view requires more detail than any off-screen buffer, we limit cloud density to the resolution of the maximal off-screen buffer.)

Ensuring that every off-screen view is sampled densely enough to put at least one sample in each pixel of the eye's view requires mapping the off-screen views to the eye's view, which depends on the particular multiview rendering technique being used. Sections 4 and 5 describe how we conservatively approximate this mapping for environment maps, and omnidirectional soft shadows. In both cases, the result is ρ_{mv}, which describes the sampling rate needed by the current triangle.

To produce points for the current triangle, EPR rasterizes it in a view-independent fashion, with each fragment becoming a point. The primary function of the geometry shader is to set the triangle up for view-independent rasterization, at the sampling rate described by ρ_{mv}.

We begin by ensuring that the triangle is front-facing in at least one off-screen view. If so, we center the triangle in the viewport, and align it to be parallel to the view plane. To achieve this, the geometry shader computes and applies the T_{align} transformation matrix unique to each triangle as described in Gavane et al. [12].

Next, we calculate the *eye buffer resolution* sampling rate ρ_{mv} for the current triangle (see Sections 4 and 5 for examples). To ensure the current triangle will be sampled at this rate, we simply scale it by ρ_{mv}. We define EPR coordinate space as world space coordinates, transformed by T_{align}, and scaled by sampling density ρ_{mv}, $\mathrm{S}_{\rho_{mv}}$, as shown in the Eq. 1. We apply this composite transform to each triangle vertex.

$$\mathrm{T}_{epr} = \mathrm{S}_{\rho_{mv}} * \mathrm{T}_{align} \tag{1}$$

If the eye's viewport has a different resolution than the *EPR buffer* used to rasterize triangles into points, the ratio of eye over EPR buffer resolution must be used as an additional scaling factor in T_{epr}. This ensures that every eye pixel is sampled. In addition we have found it useful to allow some flexibility in the rate at which eye pixels are sampled, enabling a tradeoff between point cloud size and splatting costs. We introduce a third scaling factor into T_{epr}, ρ_{scale}: at 1, ρ_{scale} puts one sample into each eye pixel. Lower values reduce sampling (e.g. 1/2 makes 1 sample for every two pixels), while higher values increase sampling (e.g. 2 makes two samples per pixel). (Note that splatting ensures that holes never occur). We examine this tradeoff in detail in Section 4.

Inspired by [23], when a projected triangle has an area of 1/10 of a pixel or less in EPR space, we may stochastically cull it, removing it from the rendering pipeline to avoid overly dense sampling of eye pixels. The smaller the area of the triangle A_t, the more likely it will be culled, with probability $P_c = 1 - 10A_t$. Although we cannot mathematically guarantee it, we have never observed any "holes" resulting in practice, and we estimate their probability to be

extremely unlikely[1]. We find that stochastic culling reduces frame times by roughly 15%.

To the fragment shader, we pass any unculled triangle's vertices and splat length in world coordinates, any material coefficients, and an aggregated triangle-view visibility bit string, with each bit representing the visibility of the triangle in an off-screen view. If any triangle vertex is visible in a (conservative) view, the triangle is visible.

3.3 The rasterizer produces points

The fragments generated as the rasterizer samples the EPR-transformed triangles are EPR's points. Note that a pixel in the eye's viewport can cover many pixels when mapped into off-screen buffers. Thus even if an EPR point's center is outside the triangle, the part of the triangle it does cover can contain many off-screen pixels. Conservative rasterization is needed to ensure the rasterizer produces these points, and avoids any "holes."

3.4 The fragment shader buffers points

The fragment shader next puts points in multiple storage buffers for later splatting by the compute shader. The exact content of these buffers varies depending on the multiview effect being implemented (see e.g. Sections 4 and 5), however at a minimum, the buffers must contain the triangle vertices, point center and splat length in world coordinates, along with the triangle-view visibility string. We currently use five storage buffers: vertices and center are in three `vector4` buffers, length and any materials are in a `uvec4` buffer, and the visibility string is in two 64-bit unsigned integer buffers.

We experimented with a bufferless implementation in which the fragment shader splatted points directly, rather than buffering them for processing in compute. This would have the advantage of avoiding the possibility of buffer overflow. While this produced identical imagery to our current solution, it was consistently less than half as fast. Schutz et al. [37] reported similar results.

3.5 The compute shader splats points

The compute shader processes all points in the storage buffers. For each point and view pairing, if the point's triangle is visible in the view, the shader splats the point into that view's off-screen buffer.

To splat, the shader projects the world-space square defined by the point center and splat length (oriented to its triangle's longest edge) into the current view, resulting a 2D quadrilateral. To simplify shader computation, we then fill the off-screen buffer pixels in the 2D bounding box around the quadrilateral with identical values. If the buffers contain forward rendering results, the lighting computation need only be performed once per point (at *eye resolution*), rather than once per buffer pixel. If off-screen rendering is deferred, every buffer pixel in the bounding box contains an identical reference to lighting inputs in the storage buffers.

Although the splat defined by the bounding box can only be slightly larger than an eye pixel, when combined with conservative rasterization in point generation, the box's use can noticeably increase the size of very small (about 1 eye pixel) details. For this reason, we clip each splat against the edges of its triangle. As we visit each buffer pixel in the splat bounding box, we fill it only if it lies inside the triangle. This quality improvement reduces speed by less than 20%.

The compute shader cannot use Z-buffering hardware, with its guarantee of atomic data access by parallel threads. Instead, we use atomic operations on integer buffers. Each off-screen buffer pixel is a 64-bit unsigned integer with the 32 most significant bits used for storing the depth of the point, and the remaining 32 bits for storing point data. For deferred rendering, because 32 bits is not enough to represent all point data, we adopt the solution described by Burns et al. [6]: we store only a reference to the point data in the storage buffers, rather than the data itself.

To reduce memory and bandwidth requirements, we adjust the sizes of these off-screen buffers to *eye resolution* in every frame. The size required depends on the mapping between these buffers and the eye's buffer, which in turn depends on the multiview effect to which EPR is being applied (e.g., Sections 4 and 5). We set the maximum resolution of any off-screen buffer to half of eye resolution. If buffer sizes change too frequently, temporal aliasing can become visible. To avoid this, we limit the frequency of size changes by requiring that they exceed at least 32 pixels in each dimension. We encourage readers to view the animations accompanying this paper to gauge our success in controlling temporal aliasing.

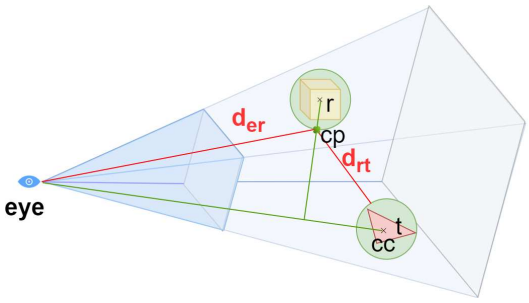


Fig. 3. *Conservative multiview sampling density is computed using the distance d_r, the shortest distance travelled by light from triangle(t) via reflective object(r) to the eye.*

4 ENVIRONMENT MAPPING USING EPR

In this section, we describe our application of EPR to environment mapping. We first discuss how to adapt EPR to environment mapping, and continue by examining the speed and image quality of our EPR-based implementation.

4.1 Adapting EPR to Environment Mapping

To adapt EPR to environment mapping, we must map off-screen views to the eye's view to determine the *eye-resolution* sampling rate

[1]A hole will have no uncullable triangles with area greater than 1/10 projecting to it, and all cullable triangles will be culled. Because well-designed models avoid projecting too many triangles inside single pixels, we estimate the likelihood P_u of a pixel lacking any uncullable triangles at 1%. If the average number of cullable triangles in a pixel is $1/A_t$, then the likelihood P_c of all cullable triangles being culled is ${P_c}^{1/A_t}$. As A_t declines, P_c approaches 5 in 100,000 — and when P_c is combined with P_u, the likelihood of a hole is 5 in 10,000,000. Our EPR viewport typically contains only 256K pixels.

ρ_{mv} and off-screen buffer sizes, defer shading in off-screen buffers, and filter reflections.

4.1.1 Mapping off-screen views to the eye's view. Finding a mapping from each off-screen view to the eye's view lets us constrain computation to *eye-resolution*. The mapping should enable efficient estimation of the size of eye pixels as they are (reverse-) projected into off-screen buffers. In the case of environment mapping, the light paths between views move from triangle t, to reflective object r, to the eye. For each triangle, we find d_r, the shortest distance travelled by the light from t via r to the eye (see Figure 3). To conservatively bound d_r, we approximate each reflective object with a bounding sphere, then locate the point cp on this sphere that is closest to the line from the triangle's circumcenter cc to the eye [36].

To ensure that each eye pixel is sampled, each triangle must be sampled at the densest of the rates required by the modeled scene's reflective objects. This rate ρ_{mv} is the maximum over all reflective objects of the inverse of the sum of the distances d_{er} and d_{rt}, as described in Eq 2 and Eq 3.

$$d_r = distance(e, r, t) = d_{er} + d_{rt} \tag{2}$$

$$\rho_{mv} = \forall_{r \in R}\ max(\frac{1}{d_r}) \tag{3}$$

where R is the set of all reflective object centers r, t is a triangle, d_{er} is the conservative distance from eye to reflective object, and d_{rt} is the conservative distance from reflective object to triangle.

As points are projected into off-screen buffers, they will often cover many pixels. To splat these points and fill these pixels, we must know point size in world space, which we model with length ℓ, the square root of the area of triangle in world space (A_{ws}) over the number of samples it covers in the EPR space. This number of samples is obtained by dividing the area of the EPR transformed triangle A_{epr} by the area of an EPR pixel ($\mathcal{P}_{epr}$), which can be computed from the graphics window size and the number of pixels in the viewport. Eq 4 shows the formula for ℓ. When a triangle in EPR space has subpixel area, we set the ℓ to the square root of the area of the EPR transformed triangle.

$$\ell = \sqrt{\frac{A_{ws}}{A_{epr}} * \mathcal{P}_{epr}} \tag{4}$$

To constrain bandwidth use by *eye resolution*, we resize off-screen buffers every frame. We determine the size needed for a particular view by computing the number of pixels its reflective object's bounding box covers when projected into the eye's view. For environment mapping, we find this optimization particularly important: it reduces frame times by more than 60%.

4.1.2 Eye-based deferred shading. As noted in Section 3.5, EPR's buffers are actually G-buffers, storing references to point shading data rather than RGB colors, and enabling deferred shading. However, rather than deferring shading until after all points are splatted, we realize an additional significant *eye-resolution* efficiency by deferring shading further until the final shading pass over the eye buffer. Thus off-screen pixels are only shaded if they are reflected into eye pixels. Figure 4 illustrates this process, and Table 1 shows the number of shading operations performed by our eye-based deferred for single bounce reflection. The operations performed by eye-based deferred is a magnitude less compared to MVR, which uses deferred shading pass over each of its cube map views.

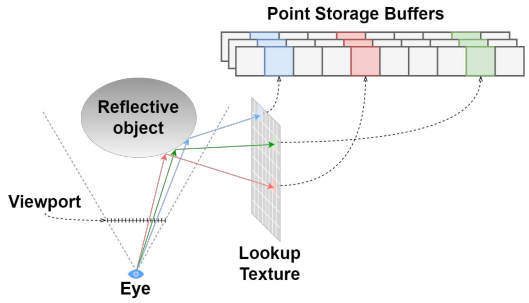


Fig. 4. *Diagrammatic representation of EPR eye-based deferred lighting evaluation*

4.1.3 Filtering reflections. High quality environment maps require filtering to accurately represent reflections. Unfortunately, hardware filtering support such as mipmapping is not available to compute shaders. We therefore filter during the eye's deferred shading pass by obtaining multiple lit samples from lazily shaded off-screen buffers in every pixel depicting a reflective object. To do so, shaders assume that normals on reflective objects change regularly, and that the material within the pixel is unchanging. To support this, in the eye's deferred shading buffer we store the reflective object's local normal gradient, which we use to supersample reflection directions using the Poisson distribution. Such supersampling would be prohibitively slow without EPR's *lazily lit*, *eye-resolution* sampling. We currently supersample reflective pixels 25 times, a number we chose empirically, because more samples did not improve visual quality. We experimented with an adaptive scheme that added four samples until color variance stopped declining, but found that it did not achieve similar quality without reducing speed.

4.1.4 Eye-based deferred recursive reflections. EPR's *eye-based deferred shading* enables further significant *eye-resolution* efficiencies

Scene	#refl objs	avg res	EPR #shading ops	MVR #shading ops	×*times*
Sponza	1	116.5	44K	1.51M	34.32
	10	115.1	146K	15.24	104.38
	20	114.5	335K	30.60M	91.34
	50	110.3	812K	76.57	94.30
Gallery	1	113.0	265K	1.12M	4.37
	10	114.4	558K	13.24M	23.73
	20	114.1	820K	27.52M	33.56
	50	112.2	1.68M	68.54M	40.80
Breakfast	1	174.5	199K	1.54M	7.74
	10	182.9	546K	15.30M	28.02
	20	156.8	743K	30.61M	41.20
	50	106.7	1.09M	78.85M	72.34

Table 1. *Number of shading computations carried out by our eye-based deferred lighting vs the number of shading computations carried out by MVR for a single bounce reflection.*

(a) *one bounce* **(b)** *two bounces* **(c)** *three bounces*

Fig. 5. *Eye-based deferred recursive reflections in the sponza scene, with two cylindrical reflective objects facing each other. The deferred shading time taken by (a) one bounce is 1.64ms with 25 samples , (b) two bounces is 2.92ms with 25 samples on first bounce and 4 samples on the second, and (c) three bounces is 3.80ms with 25 samples for the first and 4 samples for the second and third bounce.*

by enabling no-pass recursive reflections. In an MVR environment map renderer, each recursive reflection requires an additional geometry pass over all off-screen views to a second set of off-screen buffers, with each pass shading every buffer pixel. In constrast, within its G-buffer cubes, EPR can simply perform deferred lighting recursively, following a reflection off of one point onto another. Figure 5 shows multiple recursions in the sponza scene.

4.2 Results

We compared EPR's reflections with sampling to reflections generated by iVIR and MVR. For this comparison, we used OpenGL 4.5 on a PC with an Intel i5-7600K @ 3.80 GHz CPU and an NVIDIA 1080Ti[2] GPU, running Windows 10 OS. For testing, we used the sponza, breakfast, and gallery scenes [28], shown in Figure 1. Scenes were dynamic, with the eye revolving around the scene, and all the reflective objects moving periodically. We used a physically-based rendering shader [30] that accessed roughness and metallic textures. Our cube maps used 64-bit unsigned integer buffers, with a maximum adaptive resolution of 512^2. The MVR implementation for environment mapping renders the cubemap for each reflector using layered rendering, which enables rendering of six views of the cubemap in a single pipeline pass, followed by hardware mipmapping.

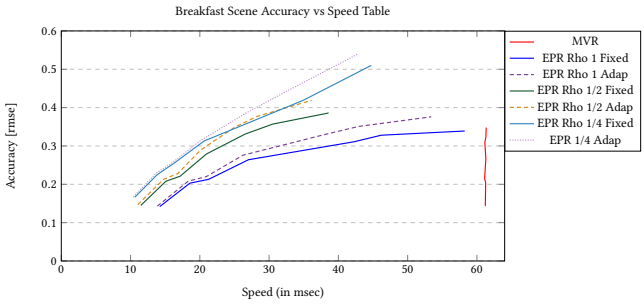


Fig. 6. *Speed vs. accuracy of EPR in the sponza scene as ρ_{scale} and shader sampling method vary. The scenes are viewed from positions that vary the average projected resolution of reflectors across 68, 106, 128, 160, 208,230, and 260 pixels, forming each curve. Accuracy is the RMSE in comparison to an MVR image at 4×HDresolution.*

[2]This GPU is now two generations old.

4.2.1 Trading Off Sampling Rate with Splatting. We began our analysis by investigating the tradeoff between point sampling rate as controlled by ρ_{scale}, and splatting complexity (which grows as sampling rate declines). Figure 6 shows how EPR's speed and quality when rendering the breakfast scene are affected as ρ_{scale} varies across 1/4, 1/2 and 1; and as shading uses fixed or adaptive sampling rates. The green curve with $\rho_{scale} = 1/2$ and fixed sampling are the clear winners, here and in the sponza and gallery scenes. One can clearly see how speed declines as the average projected size of reflectors increases (toward upper right of each curve). We used this configuration of EPR for the remainder of our analysis.

4.2.2 Memory Comparison. EPR's adaptively sized cube map buffers required 12.6MB of memory during evaluation, which is half of the memory MVR's cube maps required (25.2MB). In EPR, the total memory requirement for 20 cubemap buffers is 252MB, and 80MB per million points, and in MVR, the total memory requirement is 503MB for 20 cubemaps. Current GPU memories can contain more than a dozen GB.

4.2.3 Speed Comparison. To compare performance, we averaged GPU run-time and the number of points generated over 1000 frames of execution, with each technique generating the same views at different rates.

Table 2 compares EPR's speed to iVIR's and MVR's for the sponza, breakfast, and gallery scenes. Starting from the top to bottom, the first three rows name the scene, show the number of reflective objects and triangles in the scene, and give the average projected size of reflectors in the scene in pixels. The next row reports the performance improvement by comparing the EPR time with MVR's time and iVIR's time. The next row reports the normalized RMSE difference of EPR with MVR and iVIR. This is computed as $(RMSE_{epr} - RMSE_{mvr})/255$. The next row show EPR's RMSE in comparison to MVR HDx4 imagery, total rendering time for one bounce reflection (with eye-based deferred shading time for single reflection and a recursive reflection, 2 bounces, in parentheses), view generation time, and point generation time (with number of points in parentheses). In two bounces reported above, the first reflection is sampled 25 times and the second is sampled 4 times. The next four rows show similar measures for iVIR with mipmapping. The last two column report MVR with mipmapping RMSE and total time, The last column reports the performance improvement by comparing the EPR time with MVR's time. We highlight the best performance in the shades of blue.

EPR renders these dynamic, complex cube maps up to 6.7× faster than MVR. Speedups in comparison to iVIR are similar. EPR performed more slowly in the breakfast scene. This is because the gallery scene contains more triangles, and its projected reflectors are larger.

4.2.4 Quality Comparison. Figure 7 shows the sponza, gallery, and breakfast scenes in close and distant views, as rendered by MVR, EPR, and iVIR. The visual quality of reflections produced by our algorithm are quite comparable to that of MVR, despite requiring significantly less time to render.

Figure 8 zooms in on differences in close views, and offers possible explanations for EPR's slightly greater RMSE in comparison

Scene		Sponza				Gallery				Breakfast			
#refl objs (#tris)		1 (1.05M)	10 (1.07M)	20 (1.11M)	50 (1.20M)	1 (1.00M)	10 (1.03M)	20 (1.06M)	50 (1.16M)	1 (1.53)	10 (2.02M)	20 (2.39M)	50 (2.48M)
avg res		116.1	115.1	114.5	110.8	113.0	114.4	114.1	112.2	174.5	182.9	156.8	106.7
EPR speedup	vs MVR	1.86	5.91	6.17	5.11	2.09	6.28	6.70	5.41	1.41	2.57	3.07	3.21
	vs iVIR	1.62	2.47	2.72	7.29	1.69	2.61	2.94	5.13	1.26	2.38	2.94	6.10
EPR RMSE nrm dfc	vs MVR	0%	-0.004%	-0.004%	0.004%	0.004%	0.012%	0.027%	0.067%	0%	0.004%	0.004%	0.016%
	vs iVIR	0%	-0.008%	-0.020%	-0.046%	0%	0%	0.004%	0.008%	0%	-0.94%	0%	-0.008%
EPR	RMSE	1.13	1.15	1.17	1.27	0.81	0.86	0.94	1.17	0.18	0.25	0.31	0.43
	tot time	2.43	7.31	14.62	45.46	1.94	5.78	10.88	34.41	3.04	10.94	20.02	49.40
	(eye dfrd 1B)	(0.51) ‘	(0.83) ‘	(1.33)	(2.62)	(0.65)	(1.22)	(1.83)	(3.95)	(0.79)	(1.81)	(2.31)	(3.17)
	(eye dfrd 2B)	(0.52)	(1.46)	(2.49)	(4.24)	(0.74)	(2.49)	(3.81)	(6.96)	(0.91)	(3.38)	(5.75)	(6.20)
	view gen	0.68	3.84	8.49	27.81	0.15	1.45	3.15	11.37	1.01	5.56	10.45	26.00
	point gen	1.24	2.64	4.81	15.03	1.14	3.12	5.90	19.09	1.25	3.56	7.26	20.23
	(#points)	(603K)	(1.28M)	(1.47M)	(1.75M)	(148K)	(586K)	(677K)	(909K)	(845K)	(2.22M)	(2.88M)	(3.02M)
iVIR	RMSE	1.13	1.17	1.22	1.39	0.81	0.86	0.93	1.15	0.18	2.49	0.31	0.45
	tot time	3.93	18.06	50.25	331.39	3.25	15.11	32.03	176.53	3.83	26.06	58.88	301.29
	(dfrd)	(0.39)	(0.39)	(0.41)	(0.50)	(0.42)	(0.42)	(0.42)	(0.51)	(0.34)	(0.35)	(0.35)	(0.47)
	view gen	1.31	12.40	39.82	285.92	0.90	9.49	20.49	122.97	1.38	16.92	40.70	234.54
	point gen	2.24	5.27	10.02	44.97	1.94	5.21	11.12	53.05	2.11	8.80	17.82	66.27
	(#points)	(2.79M)	(5.73M)	(6.39M)	(7.69M)	(1.05M)	(2.78M)	(3.23M)	(4.38M)	(3.12M)	(12.23M)	(16.20M)	(17.51M)
MVR	RMSE	1.13	1.16	1.18	1.26	0.80	0.83	0.87	1.00	0.18	0.24	0.30	0.39
	tot time	4.51	43.22	90.27	232.07	4.05	36.29	72.88	186.02	4.27	28.11	61.54	158.74

Table 2. *Comparing speed and quality of EPR vs. iVIR and MVR, for the sponza, gallery, and breakfast scenes with 1-50 reflective objects and 1.0M, 990.8K and 1.5M non-reflective triangles, respectively. We report total time and RMSE, along with point cloud size and generation time for EPR and iVIR for one bounce. In EPR row, under total time we also mention the eye-based deferred lighting time for one and two bounces.*

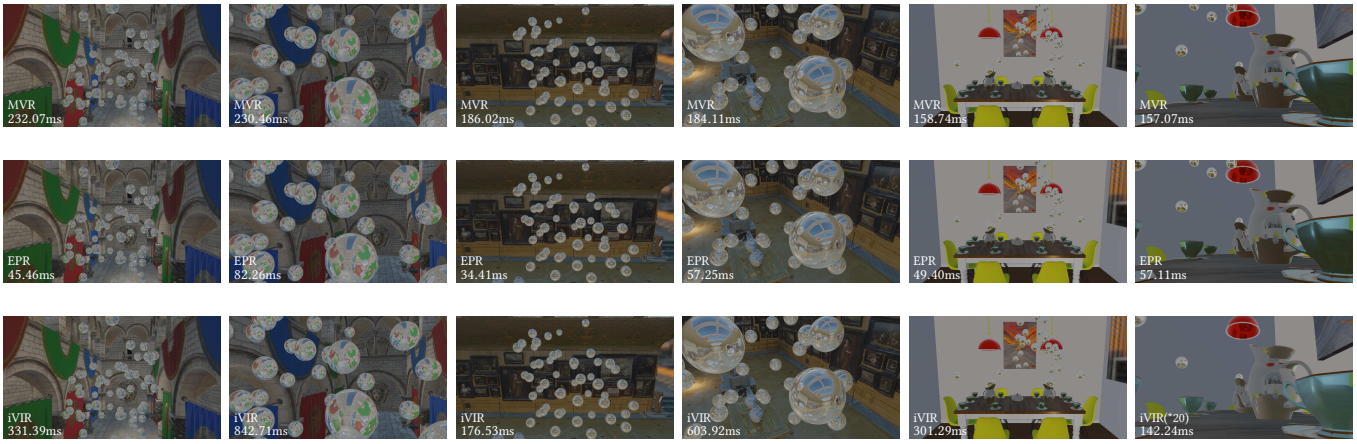


Fig. 7. *Cube-mapped reflections generated using MVR (top row), EPR (middle row) and iVIR (bottom row); in the sponza (first and second columns), gallery (third and fourth columns), and breakfast (fifth and sixth columns) scenes; using distant (first, third, and fifth columns) and close (second, fourth, and sixth columns) views. All use 50 cube maps (300 views), except close iVIR on the bottom right, it renders 20 cubemaps. The close times were measured as described at the beginning of the subsection, but with different view paths.*

to MVRx4. Because it uses conservative rasterization, EPR renders small details that MVR often ignores. This can be observed in the red boxes in Figure 8's second row (sponza scene) and fourth row (gallery scene) on the sphere magnified, EPR renders the flagpoles (in sponza) and handlebars (in gallery) , while MVR blurs them. In the magnified orange boxes for all scenes, EPR preserves the details at the edge of the sphere (sponza), the wall art (gallery scene), and the teacup and chair reflections (breakfast scene) and MVR blurs them.

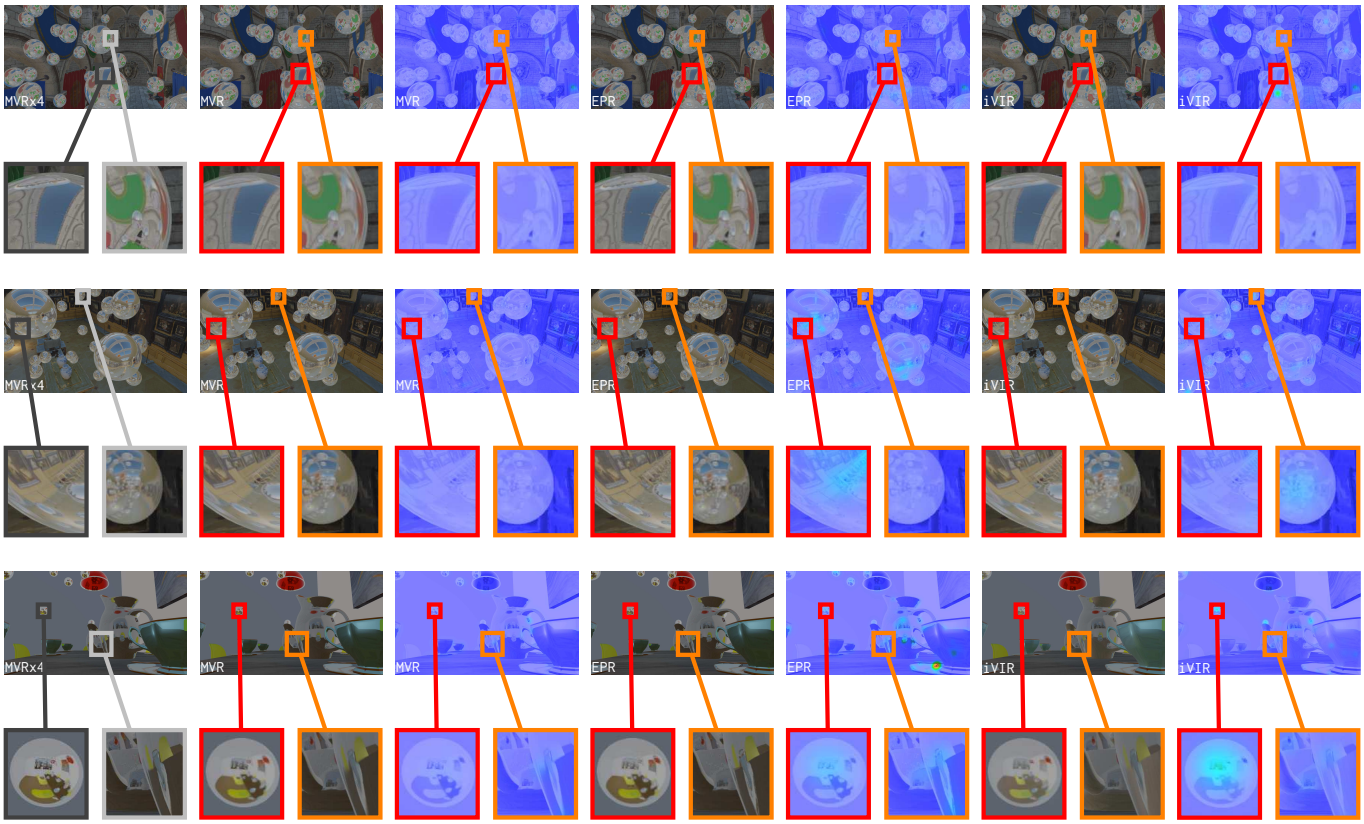


Fig. 8. *Close view quality comparisons of reflections rendered by MVRx4 at* 7680 × 4320 *resolution (leftmost column), MVR at* 1920 × 1080 *resolution (next two columns, with HDR-VDP2 perceptual comparison), EPR (next two columns), and VIR (last two columns). All HDR-VDP2 imagery compare against MVRx4, with red indicating that differences are very perceivable, and blue rarely perceived. (Note: iVIR renders only 20 reflective objects in the breakfast scene on the bottom right corner)*

As our supplementary videos show, even though the eye and reflective objects are moving and EPR generates point clouds frame-by-frame, reflections are temporally stable under animation.

5 OMNIDIRECTIONAL SOFT SHADOWS USING EPR

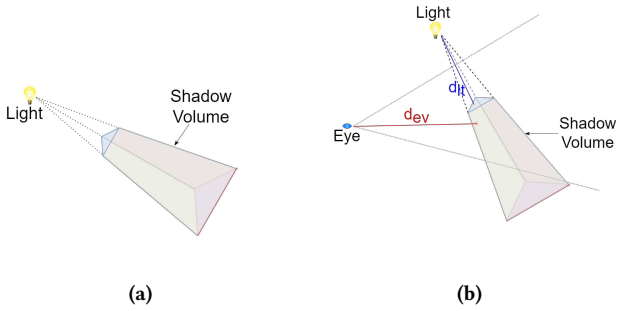
To examine the generality and the limits of EPR, we next applied it to omnidirectional soft shadows. On one hand, omnidirectional shadows are similar to environment maps, in that they build buffers (often using cube maps) describing many views of the scene. On the other hand, they are different, since building buffers containing only depth gives shaders minimal loads; and unlike reflective objects for environment maps, the location of shadow receivers (via which light would flow to the eye, were it not occluded) is not well known.

5.1 Adapting EPR to omnidirectional shadow mapping

To adapt EPR to omnidirectional soft shadow mapping, we must again map off-screen views to the eye's view to determine the *eye-resolution* sampling rate ρ_{mv} and off-screen buffer sizes. However, there is no need for deferred shading or filtering of reflections.

When rendering reflections, the position and size of reflective objects are known, making it simple to find an efficient mapping that is not overly conservative. But when rendering shadows, things are not so simple. To build a mapping between the eye and depth views, the approach we used for environment maps would follow the (occluded) light path from the eye, to the shadow receiver, to the light. Unfortunately, the location of the receiver is not well known, and indeed changes in dynamic scenes. Instead, we find the distance from the eye to an intermediate surface, the triangle's shadow volume.

To calculate EPR's sampling rate ρ_{mv} for shadows, we find the ratio of the shortest distance from a light $l \in L$, where L is a set of omnidirectional light samples, to the occluding triangle d_{lt} (see 9b). We also find the shortest distance from the eye to the occluding triangle's shadow volume d_{ev}, by finding the minimum of the distances from the eye to the two shadow's volume's two triangles (the occluder and a copy at the light's Z_{Far}) and three quadrilaterals. We then find the ratio of these distances. Note that when the eye is close to or indeed inside the shadow volume, this sampling density may approach infinity. But the practical shadow map sampling rate



(a) **(b)**

Fig. 9. *(a) The shadow volume created by a triangle. (b) The shortest distance from eye to volume d_{ev}, and the shortest distance from light to occluding triangle d_{lt}.*

is always capped by the resolution of the shadow buffers, so in this case we use iVIR's sampling density s_{ortho} (as discussed in [12]). Eq 5 describes these relationships:

$$\rho_{mv} = min(\forall_{l \in L} max(\frac{d_{lt}}{d_{ev}}), s_{ortho}) \tag{5}$$

(a) *MVR* **(b)** *iVIR* **(c)** *EPR*

Fig. 10. *Omnidirectional soft shadows rendered using (a) MVR, (b) iVIR, and (c) EPR. The omnidirectional light is a spherical light (yellow sphere at the center of the scene) with 20 samples on its surface, requiring 120 views.*

Computation of splat length is the same as for environment mapping, described in Section 4.1.2. Shadow splats need only write their projected depth into buffers, since they are not lit. For this reason their storage buffers need not describe surface material. Note that for shadows, we found it more efficient to let the fragment shader write subpixel points directly into off-screen shadow buffers, while the compute shader splatted larger points.

To adaptively resize shadow map buffers, we must know how large the eye's pixels would be, if they were (reverse-) projected onto the receiver. This depends on the distance from the eye to the receiver. As we generate the eye view's deferred shading buffer, for each triangle, we compute the maximum ratio of the shortest distance of the current triangle from each omnidirectional light ($l \in L$) sample (d_{lt}), and the shortest distance of any receiver from the eye (d_{er}). The required adaptive shadow map resolution Res_{asm} is then the maximum of all these per-triangle ratios, over all triangles, multiplied by the maximum buffer resolution Res_{max}. This relationship is shown in the Eq 6.

$$Res_{asm} = \forall_{l \in L} max(\frac{d_{lt}}{d_{er}} * Res_{max}) \tag{6}$$

5.2 Results

Here we briefly compare the performance of EPR while rendering omnidirectional soft shadows to iVIR and MVR. We used the same testing configuration is same as discussed in 4.2, except that we used 32-bit unsigned integer buffers, with a maximum adaptive resolution of 1024^2. Figure 10c shows that these three algorithms produced imagery of similar quality.

#light samples	EPR				iVIR		MVR time
	EPR #points	Frag Shader time	Compute time	total time	#points	total time	
1	8.59M	2.06	1.80	3.86	10.94 M	4.03	2.97
5	8.91M	4.58	2.53	7.11	11.43 M	5.86	7.22
10	9.04M	8.63	4.76	13.39	11.69 M	7.98	15.95
20	9.12M	21.00	7.54	28.54	11.74 M	30.74	31.80

Table 3. *Speed comparison of EPR, iVIR, and MVR while generating omnidirection soft shadows in the dining scene with* 1.4M *triangles. As the number of light samples varies, we report total time and point cloud size.*

Table 3 shows rendering times for the breakfast scene with 1.4M triangles. The leftmost column shows the number of light samples on a spherical omnidirectional light. The next column shows EPR's point cloud size. The third, fourth and fifth columns show the the total time for EPR to generate point clouds and make shadow cube maps using our hybrid approach, with the third column showing time in the fragment shader, the fourth time in the compute shader, and the fifth total time. For comparison, the next two columns report iVIR's point cloud size and total rendering time, while the last column reports MVR's total time. We observe that EPR generates slightly fewer points than iVIR, and is slightly faster than iVIR and MVR.

While this application of EPR to omnidirectional soft shadows does begin to show its generality, it also shows two important limitations of EPR. First, EPR works best when the size of the point cloud is well within an order of magnitude of the number of triangles in the model. For omnidirectional shadows, the locations of shadow receivers are not as well-known as are reflective objects in environment mapping, requiring a conservative estimate of receiver distance that increases the size of the point cloud, and reduces the effectiveness of other eye-resolution constraints on computation. Second, EPR is most effective when it can amortize large eye-pixel shader loads across many buffer pixels. Shader loads for shadows are light, since they need only compare and write depth information.

6 CONCLUSIONS AND FUTURE WORK

This paper describes eye-based point rendering (EPR), a new technique that can make multiview effects such as dynamic environment mapping much more practical. EPR achieves these improvements by introducing new *eye-resolution constraints* that significantly reduce point cloud size, and shader loads. In particular, *eye-based deferred*

shading realizes efficiencies not only on the first reflection bounce, but also on recursive bounces by avoiding any additional passes. We applied EPR to environment mapped reflections and showed that it is up to 6.7x faster than MVR, with similar speedups in comparison to VIR.

Yet EPR does have limitations. An implementation of omnidirectional soft shadows with EPR demonstrated that when shader loads are low (or in the case of shadow maps, non-existent), the efficiencies of EPR's eye-resolution constraints on shading computation are reduced. An additional lesson was that when applying EPR to a new multiview effect, the mapping between the eye and off-screen views should not be too conservative, to avoid prohibitively large point clouds.

Nevertheless, our application of EPR to environment mapping was quite promising, and as a potential application for EPR, shadow mapping is fairly unique in requiring so little computation for off-screen views. We will therefore be continuing our work with EPR.

In the near term, we will improve EPR's triangle visibility check, making it more exact by performing the check in either the geometry or fragment shader. We are convinced that our *eye-based deferred shading* scheme could be used in other renderers including MVR, and plan to demonstrate this soon.

In the medium term, because EPR does not perform well when scenes have many large triangles, we might implement a hybrid rasterization-EPR renderer, with EPR handling only smaller triangles. Longer term, we will explore the use of EPR with other multiview effects such as defocus and motion blur. We also plan to examine the use of EPR for foveation and with light field displays, which demand tens or hundreds of views in every frame.

At least two GPU improvements might make EPR still more practical. First, we often used compute shaders to accelerate splatting. However, while using compute shaders, we could not use graphics functionality such as hardware depth buffers, mipmapping for filtering and rasterization or variable rate shading for splatting. On the other hand, fragment shaders were relatively slow — likely due to threading constraints — and also could not access rasterizers or (flexible) variable rate shading. We look forward to more global, flexible access to such functionality.

REFERENCES

[1] Tomas Akenine-Möller and Ulf Assarsson. 2002. Approximate Soft Shadows on Arbitrary Surfaces using Penumbra Wedges. *Rendering Techniques* 2002 (2002), 297–306.

[2] Tomas Akenine-Möller and Jim Nilsson. 2019. Simple Environment Map Filtering Using Ray Cones and Ray Differentials. In *Ray Tracing Gems*. Springer, 347–351.

[3] Lionel Atty, Nicolas Holzschuch, Marc Lapierre, Jean-Marc Hasenfratz, Charles Hansen, and François X Sillion. 2006. Soft shadow maps: Efficient sampling of light source visibility. In *Computer graphics forum*, Vol. 25(4). Wiley Online Library, 725–741.

[4] Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson. 2019. Hybrid rendering for real-time ray tracing. In *Ray Tracing Gems*. Springer, 437–473.

[5] James F Blinn and Martin E Newell. 1976. Texture and reflection in computer generated images. *Commun. ACM* 19, 10 (1976), 542–547.

[6] Christopher A Burns and Warren A Hunt. 2013. The visibility buffer: a cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT)* 2, 2 (2013), 55–69.

[7] Franklin C Crow. 1977. Shadow algorithms for computer graphics. In *Acm siggraph computer graphics*, Vol. 11(2). ACM, 242–248.

[8] François De Sorbier, Vincent Nozick, and Hideo Saito. 2010. Gpu-based multi-view rendering. In *Computer Games, Multimedia and Allied Technology*. 7–13.

[9] Elmar Eisemann, Michael Schwarz, Ulf Assarsson, and Michael Wimmer. 2016. *Real-time shadows*. AK Peters/CRC Press.

[10] Thomas Engelhardt and Carsten Dachsbacher. 2008. Octahedron Environment Maps.. In *VMV*. Citeseer, 383–388.

[11] Randima Fernando. 2005. Percentage-closer soft shadows. In *ACM SIGGRAPH 2005 Sketches*. ACM, 35.

[12] Ajinkya Gavane and Benjamin Watson. 2022. Improving View Independent Rendering for Multiview Effects. (2022).

[13] Philipp Gerasimov. 2004. Omnidirectional shadow mapping. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics* 20 (2004), 193–204.

[14] Ned Greene. 1986. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications* 6, 11 (1986), 21–29.

[15] Markus Gross and Hanspeter Pfister. 2011. *Point-based graphics*. Elsevier.

[16] J-M Hasenfratz, Marc Lapierre, Nicolas Holzschuch, François Sillion, and Artis GRAVIR. 2003. A survey of real-time soft shadows algorithms. In *Computer Graphics Forum*, Vol. 22(4). Wiley Online Library, 753–774.

[17] Paul S Heckbert and Michael Herf. 1997. *Simulating soft shadows with graphics hardware*. Technical Report. Carnegie Mellon Univ Pittsburgh PA Dept of Computer Science.

[18] Wolfgang Heidrich and Hans-Peter Seidel. 1998. View-independent environment maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. 39–ff.

[19] Matthias Hollander, Tobias Ritschel, Elmar Eisemann, and Tamy Boubekeur. 2011. Manylods: Parallel many-view level-of-detail selection for real-time global illumination. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 1233–1240.

[20] Thomas Hübner, Yanci Zhang, and Renato Pajarola. 2006. Multi-view point splatting. In *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*. 285–294.

[21] Thomas Hübner, Yanci Zhang, and Renato Pajarola. 2007. Single-pass multi-view rendering. *IADIS International Journal on Computer Science and Information Systems* 2, 2 (2007), 122–140.

[22] James T Kajiya. 1986. The rendering equation. In *ACM Siggraph Computer Graphics*, Vol. 20(4). ACM, 143–150.

[23] Aravind Kalaiah and Amitabh Varshney. 2005. Statistical geometry representation for efficient transmission and rendering. *ACM Transactions on Graphics (TOG)* 24, 2 (2005), 348–373.

[24] Nikolas Kasyan. 2013. Playing with real-time shadows. *SIGGRAPH: ACM SIGGRAPH 2013 Courses* (2013).

[25] Jaakko Lehtinen, Timo Aila, Jiawen Chen, Samuli Laine, and Frédo Durand. 2011. Temporal light field reconstruction for rendering distribution effects. In *ACM Trans. Graphics*, Vol. 30(4). ACM, 55.

[26] Marc Levoy and Turner Whitted. 1985. *The use of points as display primitives (Technical Report TR 85-022)*.

[27] Adam Marrs, Benjamin Watson, and Christopher G Healey. 2017. Real-Time View Independent Rasterization for Multi-View Rendering.. In *Eurographics (Short Papers)*. 17–20.

[28] Morgan McGuire. 2017. Computer Graphics Archive. https://casual-effects.com/data

[29] GENES MILLER. 1984. Illumination and reflection maps: Simulated objects in simulated and real environments. *ACM SIGGRAPH'84 course notes* (1984).

[30] Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

[31] William T Reeves, David H Salesin, and Robert L Cook. 1987. Rendering antialiased shadows with depth maps. In *ACM Siggraph Computer Graphics*, Vol. 21(4). ACM, 283–291.

[32] Tobias Ritschel, Elmar Eisemann, Inwoo Ha, James DK Kim, and Hans-Peter Seidel. 2011. Making imperfect shadow maps view-adaptive: High-quality global illumination in large dynamic scenes. In *Computer Graphics Forum*, Vol. 30. Wiley Online Library, 2258–2269.

[33] Tobias Ritschel, Thomas Engelhardt, Thorsten Grosch, H-P Seidel, Jan Kautz, and Carsten Dachsbacher. 2009. Micro-rendering for scalable, parallel final gathering. *ACM Transactions on Graphics (TOG)* 28, 5 (2009), 1–8.

[34] Tobias Ritschel, Thorsten Grosch, Min H Kim, H-P Seidel, Carsten Dachsbacher, and Jan Kautz. 2008. Imperfect shadow maps for efficient computation of indirect illumination. *ACM transactions on graphics (tog)* 27, 5 (2008), 1–8.

[35] Thales Luis Sabino, Paulo Andrade, Esteban Walter Gonzales Clua, Anselmo Montenegro, and Paulo Pagliosa. 2012. A hybrid GPU rasterized and ray traced rendering pipeline for real time rendering of per pixel effects. In *International Conference on Entertainment Computing*. Springer, 292–305.

[36] Philip Schneider and David H Eberly. 2002. *Geometric tools for computer graphics*. Elsevier.

[37] Markus Schütz, Bernhard Kerbl, and Michael Wimmer. 2021. Rendering Point Clouds with Compute Shaders and Vertex Order Optimization. *arXiv preprint arXiv:2104.07526* (2021).

[38] Peter Shirley, Timo Aila, Jonathan Cohen, Eric Enderton, Samuli Laine, David Luebke, and Morgan McGuire. 2011. A local image reconstruction algorithm for

stochastic rendering. In *Symp. Ictv. 3D Graphics & Games*. ACM, 5.
[39] Johannes Unterguggenberger, Bernhard Kerbl, Markus Steinberger, Dieter Schmalstieg, and Michael Wimmer. 2020. Fast Multi-View Rendering for Real-Time Applications.. In *EGPGV@ Eurographics/EuroVis*. 13–23.
[40] Turner Whitted. 1979. An improved illumination model for shaded display. In *ACM SIGGRAPH Computer Graphics*, Vol. 13(2). ACM, 14.
[41] Lance Williams. 1978. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, Vol. 12(3). ACM, 270–274.
[42] Lance Williams. 1983. Pyramidal parametrics. In *Proceedings of the 10th annual conference on Computer graphics and interactive techniques*. 1–11.