Real-time Rendering with Compressed Animated Light Fields

Babis Koniaris*[1], Maggie Kosek†[1,2], David Sinclair‡[1], and Kenny Mitchell§[1,2]

[1]Disney Research [2]Edinburgh Napier University

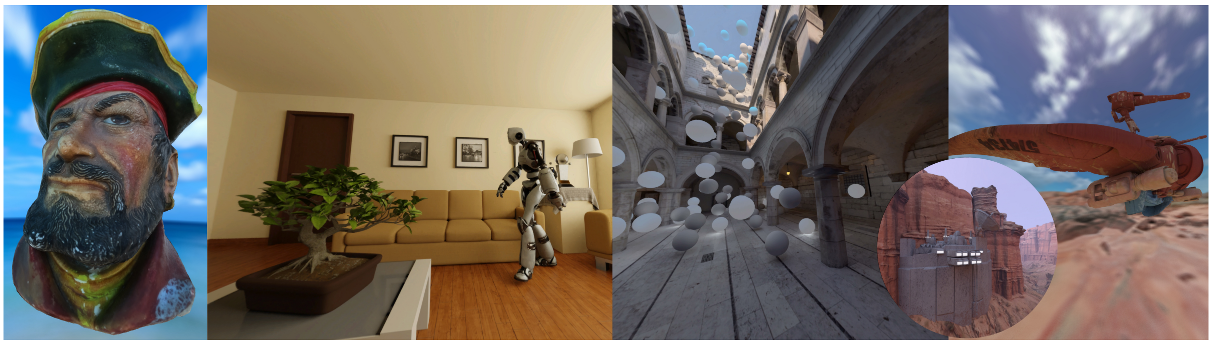

Figure 1: Real-time ray casting reconstructions of offline-rendered animated data from a sparse set of viewpoints. Our method enables online Virtual Reality exploration of sets of pre-rendered cubemap movies with optimized placement, allowing users to experience 6 degrees of freedom with full motion parallax and view dependent lighting effects.

Abstract

We propose an end-to-end solution for presenting movie quality animated graphics to the user while still allowing the sense of presence afforded by free viewpoint head motion. By transforming offline rendered movie content into a novel immersive representation, we display the content in real-time according to the tracked head pose. For each frame, we generate a set of cubemap images per frame (colors and depths) using a sparse set of of cameras placed in the vicinity of the potential viewer locations. The cameras are placed with an optimization process so that the rendered data maximise coverage with minimum redundancy, depending on the lighting environment complexity. We compress the colors and depths separately, introducing an integrated spatial and temporal scheme tailored to high performance on GPUs for Virtual Reality applications. We detail a real-time rendering algorithm using multi-view ray casting and view dependent decompression. Compression rates of 150:1 and greater are demonstrated with quantitative analysis of image reconstruction quality and performance.

Keywords: image-based rendering, compression, light field rendering, multi-view

Index Terms: I.3.3 [Computer Graphics]: Picture/Image Generation—Display Algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism —Virtual Reality I.4.2 [Image Processing and Computer Vision]: Compression (Coding) — Approximate Methods

*email:ckoniaris@disneyresearch.com
†email:maggie.kosek@disneyresearch.com
‡email:david.sinclair@disneyresearch.com
§email:kenny.mitchell@disneyresearch.com

16-19 May, Edmonton, Alberta, Canada

1 Introduction

Recently, we have seen a resurgence of Virtual Reality (VR), mainly due to recent consumer releases of Head-Mounted Displays (HMD), such as the Oculus Rift, HTC Vive and PlaystationVR. Of course, the real-time rendering performance requirements for VR content are much higher than for traditional non-VR rendering, typically necessitating lower-complexity visual fidelity [29].

Besides interactive experiences using video game engines and assets, immersive 360°videos (monoscopic or stereoscopic) have also emerged as a popular form of content. The main challenge with such videos is that they are captured assuming a fixed location, therefore lacking motion parallax and resulting in immersion breaking and feeling of the content being "flat", or even discomfort when viewers eyes diverge from these prescribed locations.

We aim to bridge the gap between cinematic quality graphics and the immersion factor provided by viewing a 3D scene accurately from any point of view. We propose a novel end-to-end solution for content creation and delivery. Our solution involves offline production-quality rendering from multiple 360°cubemap cameras, encoding it in a novel, modular video format, and decoding and rendering the content in real-time from an arbitrary viewpoint within a predefined view volume, allowing motion parallax, head tilting and rotations. Our focus is on *dynamic* scenes, and we design our processing and rendering pipeline with such scenes as the target.

The applications of our solution are numerous. It enables consumption of immersive pre-rendered video allowing six degrees of freedom, using an HMD. It can also be used for rendering film-quality visuals for non-interactive areas of a video game. In virtual production, directors can previsualize shots using a tablet as the virtual camera, streaming lightfield video decoded from a server.

Another benefit and motivator for our suggested solution is the cost reduction in asset production. When making tie-in interactive experiences using assets from films, assets typically have to be re-targetted to a lower quality form, fit for real-time rendering, and the conversion process is very expensive and time-consuming. Similarly, in architectural visualization, assets have to be converted to a lower quality form for use with a game engine, in order to allow an additional form of viewing the data, for example using an HMD. Both scenarios require time and expertise in authoring assets fit for real-time use. Our system completely bypasses the artist-driven conversion stages of the pipeline, automating the process of turning production assets into a form usable in real-time.

Our solution also incorporates three features that allow even more flexibility in terms of use-cases: (a) reconstruction is decoupled from decoding the compressed lightfield, (b) any type and number of virtual cameras can be used for reconstruction and (c) cameras are independent of each other (uncorrelated datasets). For instance, many users could explore a scene at the same time (one decoder, several eye pairs for reconstruction), allowing a Collaborative Virtual Environment with film-quality visuals. The same scene can also be viewed with an HMD, on a standard screen, or even projected in a dome. The uncorrelation of the per-camera data is useful as datasets can be enriched or replaced at a later time, allowing, for example, users to explore a scene from locations not possible before.

1.1 Contributions

Our contributions form an *end-to-end pipeline*, from offline production rendering of an animated scene from sparse optimized viewpoints to real-time rendering of scenes with freedom of movement:

- A real-time image-based rendering method that allows for free-viewpoint rendering of a cinematic quality, pre-rendered animated scene, using data from a sparse set of viewpoints.
- A method to optimize positioning of cameras for offline rendering in order to capture the scene with the least number of cameras for a given lighting environment complexity.
- Two GPU-friendly temporal compression methods (for color and depth data) that reduce video stream memory requirements (including lower GPU upload bandwidth use) and integrate with raw or block-compressed data of any spatial pixel format.
- A view-dependent decompression method that exploits precomputed visibility to optimize GPU fill and bandwidth rates.

2 Related work

Our work uses a sparse set of dynamic light field data to synthesize views of an offline-rendered scene at real-time VR rates (90fps+).

Light fields and Global Illumination. Light field rendering traditionally samples a 2D slice out of a 4D light field description of the scene [10, 19]. As light samples contain no depth information, depth estimation is a typical part of the rendering process [8, 15]. An alternative is to render directly from the light field [13]. However, without depth information, the ability to compose experiences integrated with regular interactive 3D graphics is lost. While reconstruction can yield impressive results for complex scenes [16], the rendering cost is typically very high and prohibitive for real-time rendering. Nevertheless, methods have been exhibited with real-time performance in VR [17]. Another approach is to use a relatively sparse light field probe set arranged in a uniform grid, and ray march using data from multiple probes [9]. The method works well for static scenes (although probes can be updated at an additional cost), and uses the grid information to select the subset of probes to query in rendering. Similar probes have also been used in the context of global illumination reconstruction. Light field probes [20] encode incident radiance, depth and normals, while a world-space ray tracing algorithm uses the probes to reconstruct global illumination in real-time. The probes are placed in a simple uniform grid and are used for static scenes, although the probes can be recomputed at an extra cost. Another approach uses probes to efficiently bake irradiance volumes by reprojecting color and depth data from nearby probes to construct cubemaps for each voxel [11].

Multi-view video and view synthesis. Free viewpoint television and 3D television necessitated efficient methods to compress data from cameras with angle and position variations [22, 23], and synthesize novel views using that data [26]. When depth is part of the per-camera data stream, it is important that the compressor handles it differently to color, as encoding artifacts can easily manifest as geometry distortions [21]. An alternative to transmitting and rendering image data is to reconstruct a textured mesh, which has been generated by capturing and processing video from an array of cameras [7]. While the results are good for the provided bitrate, a large number of cameras are required to capture the surface well (more than 100). Additionally, the texture maps are pre-lit, so view-dependent lighting phenomena are not recovered. Our approach employs custom compression methods for multi-view depth and color data that focus on decompression speed and minimal CPU-to-GPU memory transfers. While a mesh-based reconstruction works well for several cases (architecture, humans), it is really challenging to reconstruct high-frequency animated geometry, such as swaying trees and grass. Mesh-oblivious methods such as ours do not suffer from such a limitation on reconstructible content.

Image-based rendering. Due to the tight performance requirements of VR and mobile rendering, a common approach is to reuse data from previously rendered frames to synthesize new frames in nearby locations. One such approach is iterative image warping, that uses fixed-point iteration to generate new frames [6]. *Outatime* generates new frames by employing speculative execution, in order to mitigate wide-area latency [18]. Another approach is to use a novel wide-angle projection and *dual-view* pairs to synthesize a new image: a primary view and a secondary view at quarter resolution to approximately resolve disocclusions [25]. Szirmay-Kalos et al. [27] use color and depth cubemaps to approximate ray-traced effects. Our work, in a VR scenario, synthesizes the views for each eye individually and targets a low reconstruction rendering cost.

3 Overview

Our system is comprised of three main stages: *offline preparation and rendering*, *stream compression* and *real-time decompression and reconstruction* (Figure 2). During the offline rendering stage, we optimize the placement of a number of 360°cubemap cameras in the scene (section 4) and render color and depth images for the desired frame range that each camera should cover. Any renderer that can output color and depth images in a 360°format (either equirectangular or cubemap) can be used. The images, if necessary, are then converted to cubemaps, as they exhibit a number of advantages over the equirectangular format (section 7.4). Color images are processed into compressed streams (section 5.1), one per cubemap face per light field viewpoint sample. Depth images are first processed to determine the depth range for each viewpoint, and then they are similarly compressed into streams (section 5.2), one per cubemap face per viewpoint. The final compressed data are organized per stream, with an additional metadata header that describes the stream configuration and the locations of local sample viewpoints. Finally, the compressed data is fed to the application to reconstruct animated frames in real-time from any viewpoint (section 6).

Use-cases. We used five datasets in order to demonstrate the flexibility of our method in terms of the level of dynamic content, and the freedom of movement the viewers have. The *Sponza* and *Robot* datasets are environments with animation focused in a particular area. The users experience motion parallax within a small volume; this maps to a user standing with full freedom of movement and rotation of the head. The Robot dataset in particular demonstrates challenging reconstruction aspects such as thin geometry (potted plant) and highly reflective surfaces. The *Spaceship* and *Pirate* datasets are a representation of a single, animated object. Users can move around the object and examine it from a variety of angles and viewpoints. The Pirate dataset is a scan of a real-world model, and is used to show that our system is capable of real-time rendered light

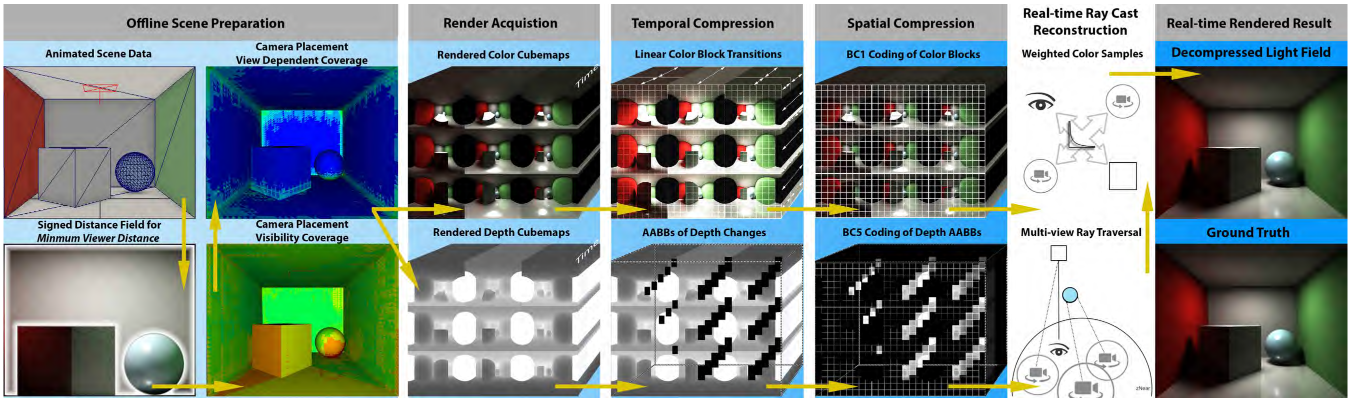

Figure 2: Overview: Animated scene analysis is performed in a preparation phase for optimized camera placement. The light field is then sampled at 360°cubemap camera viewpoints. Per-viewpoint color and depth maps are compressed separately, and then packed into per-viewpoint data clusters. In real-time, we use a subset of the compressed per-viewpoint data to render the scene from any point of view.

fields acquired from real scenes. The *Canyon* dataset demonstrates large-scale user movement in a large animated scene. Users fly over a canyon along a predetermined camera path with a set of 720 placed light field sample viewpoints. In the supplementary video, we demonstrate an additional dataset for *Sponza*, using the same scene but with animated user and viewpoint positions. The users still experience motion parallax within a $50cm^3$ cubic volume along the swept path of viewpoints.

4 Camera placement

Our camera placement algorithm employs the rendering equation [14]:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t)(\omega_i \cdot \mathbf{n}) d\omega_i \tag{1}$$

where L_o, L_e and L_i are the outgoing, emissive and incoming spectral radiances at a light wavelength λ at a time t at a position $\mathbf{x}$ with surface normal $\mathbf{n}$, while ω_o and ω_i are the outgoing and incoming light directions and f_r the bidirectional reflectance distribution function. We calculate a set $\mathbf{C}$ of 360°cubemap cameras that capture all required data for reconstruction of a lighting environment from any point of view. This requires the evaluation of L_o for any potential parameter value. In a simplified lighting model without participating media, the evaluation of this equation is enough to evaluate the time-parameterized formulation of the plenoptic function [4]. We propose to solve the integral evaluation over various levels of lighting environment and scene complexity, such as a diffuse-only static environment, diffuse/specular and dynamic. We keep the above notation, where the integral is defined over all points $\mathbf{x}$ in a union of all surfaces $\mathbf{S} = \cup \mathbf{S}_i$.

4.1 Diffuse lighting

For the diffuse scenario, the f_r reflectance term does not depend on ω_o, therefore the incoming light integral at a point is, regularly, independent of the outgoing direction, therefore the integral in eq. 1 at a point $\mathbf{x}$, time t and wavelength λ can be reused for any angle ω_o. The effect is that points can be rendered from any camera angle and the resulting data can be used to reconstruct these points for any viewing direction. The practical consequence for our camera placement algorithm is that if a point is seen from a camera, it does *not* have to be seen by any other camera.

We define an objective function for the "quality" of a camera position $\mathbf{O}$, given an existing set of cameras $\mathbf{C}_i, i \in [1, \mathbf{C}_{\text{num}}]$ for a diffuse lighting environment. The quality depends solely on how much of the surface the camera sees that it is *not already seen*, so that we effectively minimize redundancy. We define a *minimum viewer distance* function $Z(\mathbf{x})$ in order to set a limit on how close a camera can get to a surface [1]. Without such a limit, in order to cover a whole scene at an adequate sample rate, we would need to place an infinite number cameras at an infitesimal distance from all surfaces. Below, we use the visibility function $V: \mathbf{R}^3 \to \{0,1\}$ and define a set of helper functions to calculate the objective function f: I_{suit} is a compound camera "suitability" term (camera-to-point visibility term multiplied by a proximity term) and I_{cov} is the redundancy penalization term due to existing coverage.

$$I_{\text{suit}}(\mathbf{O}, \mathbf{x}) = V(\mathbf{O}, \mathbf{x}) e^{-k \max(|\vec{\mathbf{xO}}| - Z(\mathbf{x}), 0)} \tag{2}$$

$$I_{\text{cov}}(\mathbf{O}, \mathbf{x}, \mathbf{C}) = \max\{I_{\text{suit}}(\mathbf{O}, \mathbf{x}) - \max_{i \in [1, \mathbf{C}_{\text{num}}]} I_{\text{suit}}(\mathbf{C}_i, \mathbf{x}), 0\} \tag{3}$$

$$f(\mathbf{O}, \mathbf{C}, \mathbf{S}) = \int_{\mathbf{S}} I_{\text{cov}}(\mathbf{O}, \mathbf{x}, \mathbf{C}) d\mathbf{x} \tag{4}$$

The proximity term uses exponential decay (with a rate k, see fig. 3) after the threshold distance $Z(\mathbf{x})$ is exceeded, so that closer cameras are preferred but the importance of covering a point would never drop to zero. The optimal camera is obtained simply as the position that maximizes f:

$$h(\mathbf{C}, \mathbf{S}) = \operatorname*{argmax}_{\mathbf{O} \in \mathbf{R}^3} f(\mathbf{O}, \mathbf{C}, \mathbf{S}) \tag{5}$$

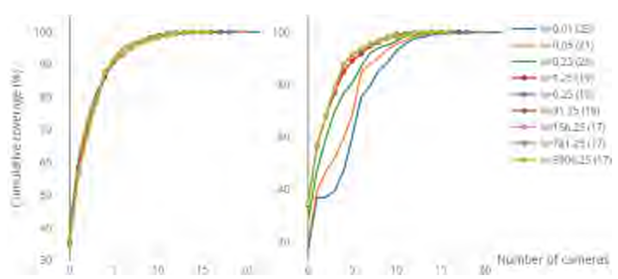

Figure 3: Varying k (eq. 4) for the Pirate dataset. The coverages score on the left include reduced-weight score from surfaces further than $Z(\mathbf{x})$ whereas the scores on the right do not. In parentheses we show the number of cameras at which the optimisation converges. High k values result in better coverage of surfaces within the minimum viewer distance and converge quicker.

A procedure that obtains the minimum number of cameras is displayed in algorithm 1.

[1] This can be defined in the following way: an artist creates a potential viewer location volume as a union of simple primitives, not intersecting with geometry in the scene. At each point $\mathbf{x}$ we can then calculate and store the minimum distance to the volume, effectively creating a sparse distance field.

Algorithm 1 Calculating an optimal set of cameras

Precondition: A union of surfaces **S**

```
function OptimalCameraSet(S)
    C ← ∅
    do
        O ← h(C,S)
        y ← f(O,C,S)
        if y > 0 then
            C ← C ∪ O
    while y > 0
    return C
```

The optimization generates *locally* optimal cameras. While a global solver could potentially generate a smaller set, it would also result in much slower computation, which can make the problem infeasible to solve for complex, long scenes. Further advantages of using a locally optimal but iterative method is that (a) offline rendering can start immediately after a camera has been calculated and (b) if in the future a larger volume would need to be covered using the same 3D scene, the optimization would continue using the existing camera set as a starting point.

4.2 Specular lighting and dynamic scenes

In order to adequately capture a specular lighting environment, we need to render every point on all surfaces from all possible directions. The camera optimization objective function eq. 4 then needs to take into account this new requirement. To express this in a way that the number of calculated cameras remain finite, we specify a minimum view angle θ between the vector from a point on the surface to two camera position: the currently tested one and an existing one. To satisfy that requirement, we modify I_{cov} using an extra angular weight term:

$$I'_{\text{cov}}(\mathbf{O},\mathbf{x},\mathbf{C}) = I(\theta \geq \angle(\overrightarrow{\mathbf{xO}},\overrightarrow{\mathbf{xC}_i}))I_{\text{cov}}(\mathbf{O},\mathbf{x},\mathbf{C}) \tag{6}$$

When the BRDF is known, we can identify where variation occurs most and parameterize θ over the hemisphere accordingly in order to lower the number of required cameras.

To optimize a *fixed* camera set for a dynamic scene, we parameterize the scene geometry **S** in time and integrate over it in the objective function:

$$f(\mathbf{O},\mathbf{C},\mathbf{S}) = \int_{t=t_0}^{t1}\int_{\mathbf{S}(t)} I_{\text{suit}}(\mathbf{O},\mathbf{x})I_{\text{cov}}(\mathbf{O},\mathbf{x},\mathbf{C})d\mathbf{x}dt \tag{7}$$

This will calculate an optimal set of cameras that remain fixed throughout the scene animation, and data generated using such cameras are better compressed with our suggested compression methods.

We have implemented the proposed method and evaluated it in a number of 3D models, for diffuse and specular lighting environments (figure 4). For our initial prototype, we optimize eq. 5 using a brute force approach that uniformly samples the subset of $\mathbf{R}^3$ inside the potential viewer volume. In the supplementary video, we show how and where coverage improves by adding cameras incrementally.

5 Compression

We compress the color and depth streams separately, as they exhibit two main different characteristics. First, color compression can be much lossier than depth compression; Depth inaccuracies result in bad reconstruction, which has a propagating effect to color reconstruction. Additionally, color pixel values change at a much higher frequency than depth. The main reason is due to noise that exists as a result of the rendering equation's approximation of integrals. Another significant reason is because depth is *shading-invariant*; shadows, lighting changes and ray bounces do not affect depth. Our aim is to exploit these characteristics and compress the data to a format that can be rapidly decompressed and uploaded to the GPU as texture data, trying at the same time to minimize the required bandwidth. We aim for low bandwidth and rapid decoding as the

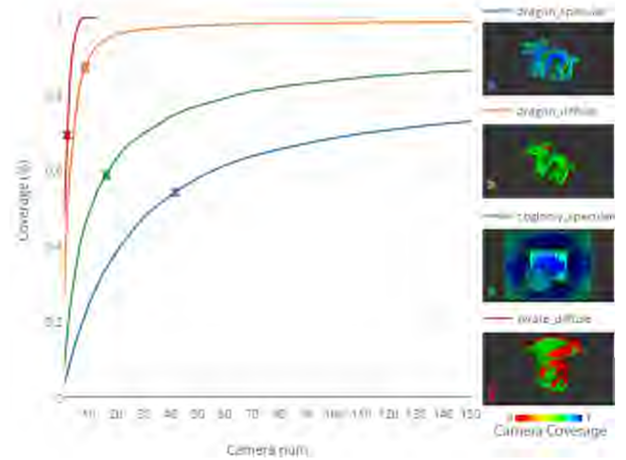

Figure 4: Results of the two variants of the camera placement algorithm (diffuse/specular) for the *Stanford dragon*, *Cornell Box* and our *Pirate* dataset. The camera placement converges to full coverage faster for the diffuse cases. Complex models and placement for specular environment will rarely converge to full coverage, as some parts of the models can either never be seen be seen from some (or any) directions, such as areas with high degree of ambient occlusion, or faces of boxes that touch each other. All the specular examples use a minimum view angle $\theta = 50°$.

potential throughput requirements are very high: nine color+depth 1024×1024 cubemaps amount to the same amount of raw data as a color image in 16K UHD resolution (15360×8640).

5.1 Temporal color compression

The goal of our temporal color compression method is to find the smallest selection of keyframes that can be used to derive the rest frames (bidirectionally predicted, B-frames), on a per-cell basis (see figure 5). The compression/decompression happens *independently*, and therefore in *parallel* for each cell. As such, the first stage is to partition the image to a regular grid, with a cell size ideally between 32 and 256 pixels per dimension (section 7.4).

Formally, let $\mathbf{B}_x$ be the image cell of size D, where x the frame index $\in [0,N)$. Below, we demonstrate how to calculate the next optimal keyframe index h given a starting keyframe index m. The reconstruction for a B-frame cell $\mathbf{B}_x$ is simple linear interpolation of two nearest frame cells, m and n where $m \leq x \leq n$, using a per-frame per-cell parameter t:

$$r(n,t) = (1-t)\mathbf{B}_m + t(\mathbf{B}_n), t \in [0,1] \tag{8}$$

We use PSNR as the quality metric q for the reconstruction:

$$q(x,n,t) = \text{PSNR}(\mathbf{B}_x, r(n,t)) \tag{9}$$

Per-frame parameters g are calculated to maximize quality:

$$g(x,n) = \arg\max_t q(x,n,t) \tag{10}$$

Finally, keyframe indices h are calculated so that the distance between them is as-large-as-possible, whilst guaranteeing a minimum level of reconstruction quality:

$$I_{\text{q}}(x,n) = I(\min_{x \in]m,n[} q(x,n,g(x,n)) > Q) \tag{11}$$

$$h = \underset{n \in]m,N[}{\text{argmax}} \left(nI_{\text{q}}(x,n)\right) \tag{12}$$

where I_{q} is an indicator function that returns 1 only if the reconstruction quality for a range of frames is for all above a threshold Q. The whole optimisation process for an animated cell is shown in algorithm 2. In practice, we quantize the **t** values to a byte for each.

This form of compression is agnostic of the how the underlying frame data is stored; the only requirement is that data in a cell needs to be independent from other cells. This allows two further optimisations: view-dependent decoding (section 6.1) and spatial re-compression (section 5.3). Decoding the compressed data in GPU is an efficient linear interpolation operation, as shown in eq. 8.

Algorithm 2 Temporal color compression for an image cell **B**

Precondition: An animated image cell **B** with N frames
Output: Vector **k** of keyframe indices and a vector **t** of per-frame parameters

function COMPRESSCOLORBLOCK($\mathbf{B}$, N)
  $\mathbf{k}_0 \leftarrow 0$
  $i \leftarrow 0$
  do
    $i \leftarrow i+1$
    $k_i \leftarrow h(\mathbf{B}, \mathbf{k}_{i-1})$
    for $x \in [\mathbf{k}_{i-1}, \mathbf{k}_i]$ **do**
      $\mathbf{t}_x \leftarrow g(\mathbf{B}, x, \mathbf{k}_{i-1}, \mathbf{k}_i)$
  while $\mathbf{k}_i < (N-1)$
  return $\mathbf{k}, \mathbf{t}$

Our pipeline fully supports the use of HDR color data, due to the agnostic nature of the compressor and the decoder. In that case, the metric used has to be changed to be better suited for HDR data [28].

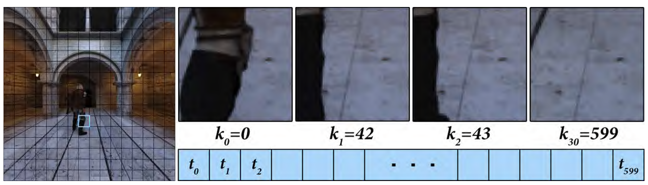

Figure 5: Color compression: An image is partitioned into a regular grid. Each grid cell stores a number of keyframe cells k_i and an array of per-frame parameter values t_j that interpolate the closest keyframes forward and backward in time.

5.2 Temporal depth compression

In terms of reconstruction, depth is more important than color, as otherwise geometry is registered incorrectly. As such, we aim for a near-lossless quality temporal depth compression method. We exploit the fact that depth maps, captured from a static camera, display low frequency of updates. We store video frames as keyframes or P-frames. Keyframes store all data for the frame, while P-frames only encode the *differences* to the last keyframe. The differences are encoded using a set of axis-aligned bounding boxes (AABB): each P-frame stores a list of AABB coordinates and the raw depth data contained in each. The depth compression process is shown in algorithm 3. We choose AABBs because the data memory layout maps well to GPU texture update functions, therefore updating a depth video texture is simply a serial set of texture update calls, using the depth data as-is from the P-frame data stream. Therefore, it is important to calculate as-tight-as-possible AABBs, in order to update the least number of pixels. Calculating tight-fitting AABBs is well studied in collision detection literature, and it is very closely related to calculation of AABB trees [5]. Our use case is slightly different, as 1) we are only interested in the "leaf level" of an AABB hierarchy, 2) calculation time is not a priority, as compression happens offline and 3) too many AABBs can cause a high overhead of GPU texture update calls.

Algorithm 3 Temporal compression for depth frames **D**

Precondition: Depth images **D** for N frames, of dimensions w, h
Output: A vector **C** of compressed frames, each stored as a list of rectangular frame data with the corresponding rectangle

function COMPRESSDEPTHIMAGES($\mathbf{D}$, N)
  $r \leftarrow (0, 0, w, h)$
  $\mathbf{C}_0 \leftarrow \{(\mathbf{D}_0, r)\}$
  for $i \in [1, N-1]$ **do**
    $\mathbf{C}_i \leftarrow \{\varnothing\}$
    $\mathbf{D}_{\text{diff}} \leftarrow I(|\mathbf{D}_i - \mathbf{D}_{i-1}| > \varepsilon)$ ▷ computes a binary difference map
    $\mathbf{R} \leftarrow \text{CalculateAABBs}(\mathbf{D}_{\text{diff}})$
    for $r \in \mathbf{R}$ **do**
      $\mathbf{C}_i \leftarrow \mathbf{C}_i \cup (\text{SubImage}(\mathbf{D}_i, r), r)$
  return C

The AABBs are calculated using a simple exhaustive search that starts with a single tight AABB of the difference map and iteratively splits it to smaller, tighter AABBs until the maximum number of AABBs has been reached. We show an example in figure 6.

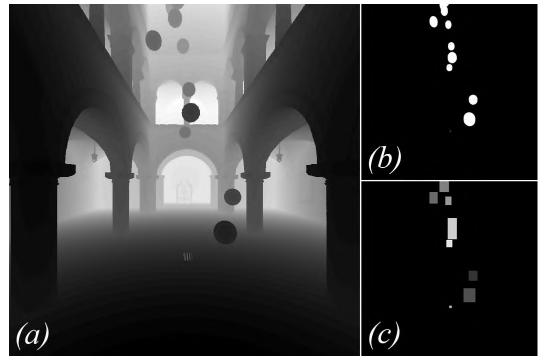

Figure 6: Depth compression: AABB fitting example for a cubemap face of a view at a single frame. a) Depth Map, b) Depth difference with previous frame, c) AABB set that encloses the differences as tightly as possible. A maximum of 8 are used in this example.

5.3 Spatial recompression and fixed-point depth

The compression methods that we described reduce data only in the temporal domain. They were designed as such, so that they could be used directly on already *spatially compressed* data. We exploit the properties of hardware-accelerated block-compression texture formats, such as S3TC [12], BPTC [2], RGTC [3] or ASTC [24], as they have fixed data rate and fixed block dimensions. In color compression, if the BCn block dimension is a divisor of the cell dimension, the block-compressed data can be stored directly in the cell data stream. Similarly, in depth compression, if the block dimension is a divisor of the AABB corner points, block-compressed data can be stored instead of raw for each AABB. For color data, we can use formats BC1, BC6 and BC7 depending on the quality requirements and dynamic range. Depth values are originally generated as 32-bit floating point values. In order to reduce bandwidth and maintain high precision, we map the values to 16-bit unsigned integers in logarithmic space, using the following conversion:

$$z_u = (2^{16}-1)log(\frac{z}{z_{\text{near}}})/log(\frac{z_{\text{far}}}{z_{\text{near}}}) \quad (13)$$

where $z_{\text{near}}, z_{\text{far}}$ the minimum and maximum depth values respectively. Logarithmic space provides a better distribution for the depth values, offering more precision closer to the camera. As there is no hardware-accelerated compression for 16-bit unsigned values, we split the data to two 8-bit channels and compress them using the BC5 texture compression format.

6 REAL-TIME RENDERING

The real-time rendering algorithm reconstructs the scene from a given camera using data for a set of viewpoints (locations and color/depth textures) and camera parameters using ray marching, and is shown in algorithm 4 (last page). The rendering algorithm is comprised of two parts: calculating the intersection with geometry and calculating the color contribution from all views. The intersection step involves marching along the ray until an intersection is found. The ray marching step size is constant in the *non-linear space* that the depths are stored (eq. 13), so that resolution is higher near the camera. A point on the ray is guaranteed to hit a surface at the first occurence where BetweenViewpointAndGeometry [2] is false for *all* viewpoints. Conversely, if there is *even one* case where the point on the ray is between a viewpoint and its reconstructed

[2]BetweenViewpointAndGeometry returns true if, at a point **p**, the distance from a viewpoint **v** sampled using the direction $\mathbf{p}-\mathbf{v}$ is farther than $|\mathbf{p}-\mathbf{v}|$

position at that direction, then the point is in empty space. Color calculation is based on the product of two weighting factors: the distance of the ray point to the closest depth sample from a cubemap (w_{depth}) and the distance of the ray point to the camera (w_{cam}). Both are exponential decay functions, with w_{depth} decaying at a faster rate. The weight w_{depth} ensures that the contributed color will be from a sample as near as possible to the ray point. The weight w_{cam} ensures that, if we have a set of samples with similar w_{depth} values, those near the camera are preferred, ensuring better reconstruction of view-dependent shading.

6.1 View-dependent decoding

Users can only see part of a the reconstructed 3D scene at any given time, due to the limited field of view. Therefore, decoding full cubemap frames is unnecessary and hurts performance. Our compression methods support *view-dependent decoding*: the ability to decode only the parts of the video that are visible to viewers, which we make use of to lower the per-frame bandwidth required to update viewpoint texture data. Using our compression scheme (section 5.1), color video streams are formed by smaller compressed streams of independent cells. Each cell of a particular viewpoint creates a world-space pyramid that extends to infinity, formed by all lines originating from the viewpoint and intersecting the cell surface on a cube centered at the viewpoint. Video content for the cell can only be projected within the frustum that is obtained by cutting off the volume of the pyramid before and after z_{near} and z_{far} values, as the depth values can only lie within that range. If this frustum does not intersect with the camera frustum at a given frame, the cell data do not need to be updated, as they are not seen by the viewer. Depth video streams (section 5.2) can benefit from the same optimisation by splitting each stream into tiles, and compressing each individually. For our examples, we use a 2×2 tile grid per cubemap face for depth data, to benefit for the view-dependent decoding optimisation without introducing too many new partitions and AABBs.

6.2 View-selection heuristics

Typically, not *all* cameras can provide useful data at any given time. For example, in the *Spaceship* dataset, viewpoints on the other side of the ship (with regards to eye location) will provide very little useful data compared to viewpoints near the eye. In other cases, we might want to use a lower number of viewpoints in order to maintain a high framerate, which is crucial for a VR experience.

We formulate heuristics that calculate a prioritized set of viewpoints each frame. Prioritisation is important for maintaining coherence of the set elements across frames. Coherence is important for the rate of updates, as every time the viewpoint changes, the associated texture data need to be updated as well. Besides prioritisation, we also use viewpoint culling for additional control over the active viewpoint set. We use two different prioritisation methods and two different culling methods accordingly.

Distance-based prioritisation. The viewpoints are sorted based on their distance to the eye (closest point is highest priority). This prioritisation is useful in scenarios where the user moves through a large space of distributed viewpoints, as only nearby viewpoints provide useful data. It works well in conjunction with the rendering contribution weight w_{cam} in algorithm 4. **Angle-based prioritisation.** The viewpoints are sorted based on their angle to the eye location, using a reference point as the origin (smallest angle is highest priority). This is useful in scenarios like the *Spaceship*, where a model is captured from all angles. In that case, the reference point is set as the center of the model. This scheme works well with heavy view-dependent effects, as the prioritised cameras have the best matching data. **Angle-based culling.** Viewpoints forming an angle with another, higher-priority viewpoint that is smallest than a given threshold, using the eye location as the origin, are not placed in the active viewpoint set. The reasoning for this type of culling is that when the angle between two viewpoints is very small, the data redundancy is very high, therefore the higher-priority viewpoint is used instead. **Performance-based culling.** Low-priority viewpoints are culled if the runtime performance is below a given requirement. Given an estimated cost that a view incurs on the runtime and the current runtime performance, we can estimate how many views need to be culled in order to reach a performance target. This is important when maintaining a high framerate is crucial for user experience, for example in VR applications where a low framerate can introduce flickering with physical discomfort.

To form a heuristic, we use a combination of the above. We select a prioritisation method to sort the view locations and then apply one or more culling methods: for example we apply angle-based culling to skip redundant views and then performance-based culling to ensure that we maintain performance. Finally, we rearrange the resulting set of views, so that the order of the active viewpoints is maintained as much as possible with respect to the previous frame.

7 Results

Our test system is an Intel Core i7 6700K with 32GB RAM and an NVIDIA TITAN X GPU card with 12GB RAM. The input datasets were created using Pixar's RenderMan. The *Sponza* dataset consists of nine 360°cameras, 600 frames each. The cameras are placed as follows: one in the middle of the cubic volume, and the rest at the eight corners. The *Spaceship* dataset consists of fourteen 360°cameras surrounding the object, 300 frames each. In this example, we dynamically select a subset of the cameras, based on the angle of the vectors from the center of the object to the viewer and camera. The *Pirate* dataset uses fifteen 360°cameras distributed in front of the face of the pirate, each consisting of 150 frames. This dataset demonstrates the capability for an end-to-end pipeline from real-world data to a real-time lightfield playback. Camera positions for this model have been generated using our camera placement algorithm. The *Robot* dataset consists of sixteen 360°cameras, 78 frames each. The cameras are arranged in a $4\times2\times2$ oriented grid. This example poses several challenges for a faithful reconstruction, such as thin geometry (potted plant leaves) and highly specular and reflective surfaces (robot, floor, table). The *Canyon* dataset consists of 720 360°cameras, distributed along a flight path. The dataset contains an animated satellite dish. For this flythrough example we use a conservative rendering scheme, where the location along the path is tied to the animation time. Therefore, each camera only renders the time range mapped to nearby path segment. In our scenario, the environment is generally static, so most cameras render a single frame. As such, most cameras in this example do not benefit by our temporal compression codecs therefore they are omitted from the compression and bitrate tables. In all examples, the 360°virtual cameras generate $1024\times1024\times6$ cubemaps. The color compressor uses a threshold PSNR of 50 dB and a cell size of 64 pixels. Below, we discuss results in compression efficiency, throughput optimisations, runtime performance and reconstruction quality.

	Raw (GB)	Stream	Temporal Window	Depth AABBs	Temporal (GB)	+Spatial (GB)	Ratio (%)	+LZMA2 (GB)	Ratio (%)
Spaceship	73.83	Color	-	-	0.97	0.162	0.22	0.016	0.02
	98.43	Depth	16	8	3.18	1.59	1.61	0.076	0.07
			2	8	2.82	1.41	1.43		
			2	16	2.58	1.29	1.31		
Robot	22.46	Color	-	-	1.76	0.293	1.30	0.100	0.44
	29.25	Depth	2	16	0.32	0.159	0.54	0.053	0.18
Pirate	39.55	Color	-	-	3.23	0.539	1.36	0.129	0.32
	52.73	Depth	2	16	1.58	0.79	1.49	0.27	0.51
Sponza	94.9	Color	-	-	8.94	1.49	1.57	0.590	0.62
	126.56	Depth	2	8	3.26	1.63	1.28	0.342	0.27

Table 1: Color and depth compression (in GB). *Raw* corresponds to 32-bit floating point depth values and 24-bit RGB data, followed by our temporal compression methods (5.1, 5.2), spatial re-compression (5.3) and finally the total percentage over the raw color or depth dataset. We additionally provide results after lossless compression, using LZMA2, to demonstrate the further compressibility of the output data. We provide three cases for depth compression for the *Spaceship* dataset, to show the effect of varying temporal frame window size and number of AABBs.

	Size (GB)	Ratio (%)	Decode (ms)
HEVC-hq	0.009	0.031	57.5
HEVC-lossless	0.208	0.71	57.69
Ours (+LZMA2)	0.159 (0.053)	0.54 (0.18)	1.62

Table 2: Depth compression comparison in Robot dataset. We compare compression and decoding performance against HEVC using high quality and lossless presets. Decoding measures decoding time for all faces from a subset of 9 views (54 streams in total). Our method has a clear advantage in decoding speed, which is paramount for the required data throughput. LZMA2 streams are decompressed asynchronously, so LZMA2 decompression times are not included in the decoding times.

7.1 Compression

In table 1 we show compression results obtained using our methods. Unlike typical codecs, they are primarily optimized for pure decoding performance, due to the large number of pixels that needs to be decoded in runtime. The decoding cost of our methods is insignificant, as it is limited to memory transfers (parts of a texture, each frame) and, in the case of colors, a linear interpolation operation, which is performed in hardware. Decoding the block-compressed data is also performed in hardware. The *Spaceship* and *Pirate* datasets exhibit very good spatial compression ratio, mainly because many of the cubemap faces contain no data: in such a case a cell would compress to two keyframes (out of 300) using any parameter value. In contrast, the depth compression is not as effective, partly because the color block-compressor provides a better ratio (6:1 for DXT1 over 2:1 for BC5) and partly because of the 16-frame depth window and number of AABBs, which increase the area that needs to be updated each frame. The *Sponza* and *Robot* datasets in contrast offer better compression in depths rather than colors. This is because of the effect of light bounces in the scene: For example, the falling spheres cause light to be bounced on the walls and floor, causing changes in shading in areas where depth remains constant. All results can be further compressed with general lossless methods, such as LZMA2. As can be seen in the table, such compression can significantly reduce the data to rates that can be easily transmitted over a network. The reason for the higher lossless compression of the *Spaceship* dataset is because of the redundancy of data (e.g. no colors or depths) across several views and cubemap faces.

Quality and resolution of depth maps are critical for reconstruction (moreso than color data), therefore our compression scheme for depths is not very aggressive. As the rendering algorithm samples multiple views for depth, inconsistencies would manifest as holes on surfaces (a single intersection test incorrectly failing in algorithm 4) or floating geometry (all intersection tests incorrectly pass in algorithm 4). Similarly, during color reconstruction, w_{depth} factor could be incorrectly set, leading to wrong color contribution. If compression is of paramount importance for a particular implementation, the depth streams can be compressed with *any* compressor, including a stream of AABBs for each frame, pre-calculated using our method. In real-time, the frame can then be decompressed and we can then use the AABB info to update the appropriate texture region.

In table 2 we show a comparison of depth compression against a hardware-accelerated HEVC implementation by NVIDIA [1] for the Robot dataset. Floating point depth values were mapped to 24-bit integers using equation 13 and split to RGB channels, swizzling bits for better compression: $\mathrm{SetBit}(c_{(i \mod 3)}, \mathrm{GetBit}(u,i) \ll (i \backslash 3)) \forall i \in [0,23]$ where c is the output 3-channel color and u the input 24-bit integer. Our method has better overall compression than the lossless HEVC and the decoding speed is an order of magnitude better.

7.2 Runtime performance

The performance in all our examples is real-time and can be used with performance-demanding HMDs (often requiring 90Hz rendering rate for a smooth experience). The performance bottleneck of our runtime algorithm is the texture update stage, where all visible parts from all views in the active set need to be updated to display the current frame correctly. As such, we attempt to reduce the volume of data by only updating visible parts for each viewpoint. Even with such an optimisation, in cases where the user is looking towards a direction with moderate to heavy animation, the performance generally drops due to increased texture updates. In such cases, we use heuristics (section 6.2) to detect such performance drops and adjust the active set size by dropping lower priority views. Reducing the number of views improves perfomance both by reducing the data that needs to be updated each frame, but also because fewer textures are sampled in the ray marching shader.

Table 3 shows timings for the texture update and rendering parts of the runtime, as well as the effective bitrate for color and depth data. We measured the bitrate by recording the updated color and depth at each frame, for all frames over several repetitions of the video. It is clear that the view-dependent optimisation significantly reduces the bitrate, and as a consequence it reduces texture update time, resulting in improved performance. Depth bitrate is typically higher than color as the per-texel storage cost is higher.

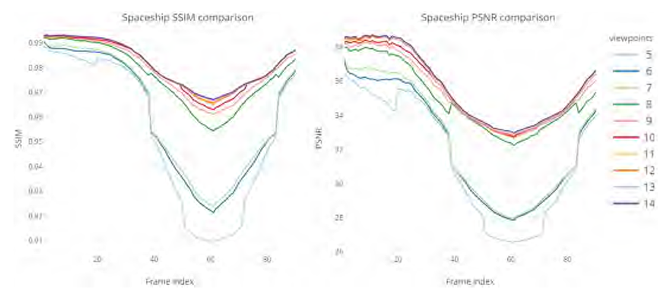

Figure 7: PSNR and SSIM of reconstruction against reference for the *Spaceship* dataset using a variable number of active viewpoints, selected via the angle-based prioritisation heuristic.

7.3 Reconstruction quality

The quality of our reconstruction largely depends in the number and placement of cameras in the animated scene. Challenging cases from a geometrical point of view involve thin geometry (e.g. chains of hanging braziers in *Sponza*, potted plant in *Robot*) and deep crevices (e.g. parts of *Spaceship*, especially in front). To evaluate the reconstruction quality of our method, we rendered 90 frames of turntable animation for the *Spaceship* dataset using Renderman, and compared it with our method by using the camera path information to reconstruct the views. Challenging cases in terms of lighting complexity involve highly specular reflections (e.g. table, floor and robot in *Robot*). To demonstrate how our method compares to ground truth, we rendered a small set of images for the *Robot* dataset using a constant frame and a camera moving between two view points (figure 9). It can be observed that the largest errors are disocclusions, followed by edge reconstruction. The latter can be explained by the nearest-depth filter that is used for the depth image generation (sec. 7.4), as multiple depth samples (typically prominent at the edges) are lost. We evaluate the PSNR between our reconstruction and the reference against a simple point rendering of the *Spaceship* dataset, where every texel of every cubemap of every view is projected into the world-space position using the depth map information (see supplementary video). We also compare the PSNR values obtained using different number of active viewpoints, shown in figure 7. It can be observed that the greater the number of views is used at any given time, the better the reconstruction becomes. The drop in reconstruction quality around frame 60 can be explained by the fact that at those frames the camera is pointing directly at the ship's crevices, where data is not present among the viewpoint samples. This could be solved by having a further viewpoint sample at such a location (figure 8).

7.4 Implementation Analysis

Cube mapping vs equirectangular. Equirectangular mapping is often used to generate 360°images. While this is a convenient, single-

	View Dependent	Active Viewpoints			Animated Color Data at 24fps (in Mbps)			Animated Depth Data at 24fps (Mbps)			Per Frame Update Time (ms)			Render Time (ms) 1024x1024		
Spaceship	no	5	9	14	36.00	91.04	119.12	216.16	880.64	1092.48	0.74	2.18	2.82	2.83	2.94	3.30
	yes	5	9	14	19.52	35.12	40.48	205.20	471.68	589.76	0.60	1.13	1.40	2.84	2.94	3.31
Sponza Soldier	no	5	9	-	292.43	521.35	-	323.93	557.61	-	3.77	6.79	-	1.40	1.60	-
	yes	5	9	-	153.07	241.25	-	185.49	287.49	-	2.08	3.22	-	1.42	1.63	-
Sponza Floor	yes	5	9	-	54.95	124.59	-	46.21	116.70	-	0.64	1.45	-	0.98	1.18	-
Sponza Spheres	yes	5	9	-	213.54	368.56	-	253.24	404.84	-	2.71	4.61	-	1.85	2.23	-

Table 3: Timings and bitrates in the runtime. We show the effective bitrate (in Mbps) of texture updates for a 24fps video, and the time needed for texture updates and rendering (in milliseconds). We vary the number of active viewpoints and we show the results of view-dependent decoding (VD) versus without. Rendering cost is for a 1024×1024 image. Floor, soldier and spheres view implies looking towards areas of low, medium and high degree of animation. We can observe that using view-dependent decoding typically halves the bitrate and significantly improves performance.

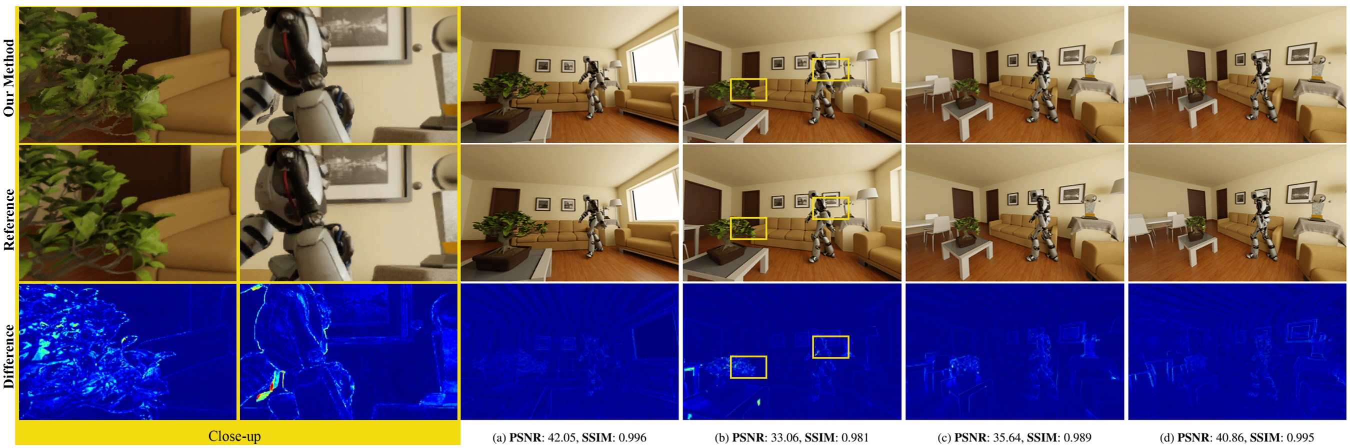

Figure 9: Comparison of our reconstruction (top row) with ground truth (middle row) for the Robot dataset. We render images from four locations along the line between two 360°cameras (scene views (a) and (d) are the endpoints, with two zoomed sections on the left). The reconstructions at the endpoints have higher PSNR/SSIM, but still exhibit heat map differences due to compression and loss of depth information on edges. Specular effects are handled gracefully (e.g. glass sphere), while thin and transient geometry poses a challenge.

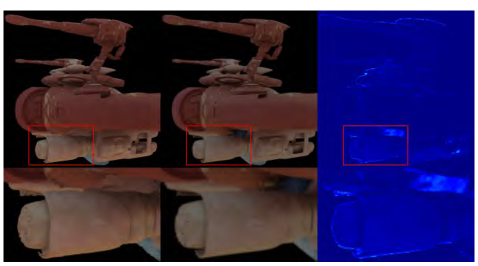

Figure 8: Ground truth comparison for *Spaceship* dataset. We compare the reconstruction (left) from a random angle to an image generated explicitly from that angle by the offline renderer (middle).

image representation, it has several drawbacks compared to cube maps, which have been used for a long time in computer graphics: *Mapping distortion.* The equirectangular mapping exhibits higher distortions the closer a point is to any of the two poles. At its most extreme distortion, the single points at the poles cover the same number of pixels as the great circle. Standard video codecs also perform better with cube maps, as they assume motion vectors as straight lines. *Storage cost.* A cubemap needs 75% of the storage space required for an equirectangular map at the same pixel density. In such maps, the excess storage cost is spent near the distorted poles. *Sampling performance.* Cubemap sampling using 3D cartesian coordinates has been implemented in hardware since 2000. Equirectangular mapping requires the use of trigonometric functions, which increases the sampling cost when used frequently.

Cell dimensions for temporal color compression. Our temporal color compression scheme first partitions an image into a regular grid of $N \times N$ cells, which are then compressed and decompressed independently of each other. There is no globally ideal cell size that is always the best for any given case; this depends on the content and the hardware that is used for decompression. For the following comparisons, we assume cubemap faces of dimensions 1024×1024 and a reference cell size of 64. Selecting a very small cell size (e.g. $N = 16$) results in better identification of static or animated cells, therefore the cumulative ratio of keyframes over frames will be lower (better). But the smaller cell size also reduces the compression ratio as the per-cell, per-frame data becomes higher (bytes used by t over bytes used by cell). During decompression, the performance can also be lower, as the number of texture update calls and frustum culling checks becomes higher ($16\times$). Conversely, selecting a very large cell size (e.g. $N = 256$) results in higher (worse) cumulative ratio of keyframes over frames, but in a better per-cell, per-frame compression rate. During runtime, frustum culling is faster due to lower number of checks, but texture update cost can be higher, as the coarser cell granularity results in more data in need for update.

Depth map filtering Production renderers often apply filtering on outputs to reduce aliasing artifacts and noise. Such a filter is destructive for depth output, distorting silhouettes by linearly interpolating depths of foreground and background disjoint geometry.

8 Conclusion

We have presented a set of methods than enable real-time reconstruction of pre-rendered video from an arbitrary point-of-view within an animated light field, that is capable to run at 90Hz on modern hardware, allowing smooth, high-quality VR experiences. Our camera placement method ensures that the datasets minimize redundancy among views. Our temporal compression methods are specialized for the color and depth streams, whereas they can also be used in tandem with hardware-accelerated, spatial texture compression formats. Decompression for both methods is very fast to evaluate and minimizes GPU memory bandwidth by only updating out-of-date and visible texture regions. Our runtime rendering algorithm is very easy to integrate due to its simplicity and uses prioritization heuristics to control the number of active viewpoints, and by extension, the performance versus quality tradeoff.

In the example scenes, we have purposefully not used offline-rendered images containing camera effects such as depth of field and motion blur or participating media and other volumetric effects. Our method uses images capturing a single depth value per pixel, so there is a direct mapping of depth pixels to color pixels. As such, reconstruction using imagery with such effects would result in artifacts. In further work, we plan to add support for camera effects, such as depth of field and motion blur, as runtime components.

In the future, we would like to improve support for thin geometry and complex light paths, such as mirror-like specular reflections, refractions, caustics and transparency. We would also like to improve the spatial adaptivity of the compression codecs by using a subdivision scheme such as a quadtree. Hardware texture decompression informed by our scheme could reveal a much higher performance towards ultra high resolution VR.

Acknowledgements

We would like to thank Maurizio Nitti for the Robot scene, Llogari Casas for the Pirate model, as well as Fraser Rothnie and Desislava Markova for their work on the modified Sponza scene. We are also thankful for the insightful feedback provided by the anonymous reviewers. This work was funded by InnovateUK project #102684.

References

[1] NVIDIA VIDEO CODEC SDK. `https://developer.nvidia.com/nvidia-video-codec-sdk`.

[2] RGTC Texture Compression. `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_texture_compression_bptc.txt`.

[3] RGTC Texture Compression. `https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_texture_compression_rgtc.txt`.

[4] E. H. Adelson and J. R. Bergen. *The plenoptic function and the elements of early vision*. Vision and Modeling Group, Media Laboratory, Massachusetts Institute of Technology, 1991.

[5] G. v. d. Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

[6] H. Bowles, K. Mitchell, R. W. Sumner, J. Moore, and M. Gross. Iterative image warping. In *Computer graphics forum*, volume 31, pages 237–246. Wiley Online Library, 2012.

[7] A. Collet, M. Chuang, P. Sweeney, D. Gillett, D. Evseev, D. Calabrese, H. Hoppe, A. Kirk, and S. Sullivan. High-quality streamable free-viewpoint video. *ACM Transactions on Graphics (TOG)*, 34(4):69, 2015.

[8] L. Dabala, M. Ziegler, P. Didyk, F. Zilly, J. Keinert, K. Myszkowski, H.-P. Seidel, P. Rokita, and T. Ritschel. Efficient Multi-image Correspondences for On-line Light Field Video Processing. *Computer Graphics Forum*, 2016.

[9] S. Donow. Light probe selection algorithms for real-time rendering of light fields, 2016.

[10] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 43–54. ACM, 1996.

[11] J. Hooker. Volumetric Global Illumination at Treyarch. In *Advances in Real-Time Rendering Part I Course*, SIGGRAPH '16, New York, NY, USA, 2016. ACM.

[12] K. I. Iourcha, K. S. Nayak, and Z. Hong. System and method for fixed-rate block-based image compression with inferred pixel values, Sept. 21 1999. US Patent 5,956,431.

[13] Y. J. Jeong, H. S. Chang, Y. H. Cho, D. Nam, and C.-C. J. Kuo. Efficient direct light field rendering for autostereoscopic 3d displays. SID, 2015.

[14] J. T. Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[15] C. Kim, K. Subr, K. Mitchell, A. Sorkine-Hornung, and M. Gross. Online view sampling for estimating depth from light fields. In *Image Processing (ICIP), 2015 IEEE International Conference on*, pages 1155–1159. IEEE, 2015.

[16] C. Kim, H. Zimmer, Y. Pritch, A. Sorkine-Hornung, and M. H. Gross. Scene reconstruction from high spatio-angular resolution light fields. *ACM Trans. Graph.*, 32(4):73–1, 2013.

[17] B. Koniaris, I. Huerta, M. Kosek, K. Darragh, C. Malleson, J. Jamrozy, N. Swafford, J. Guitian, B. Moon, A. Israr, S. Andrews, and K. Mitchell. Iridium: immersive rendered interactive deep media. In *ACM SIGGRAPH 2016 VR Village*, page 11. ACM, 2016.

[18] K. Lee, D. Chu, E. Cuervo, J. Kopf, Y. Degtyarev, S. Grizan, A. Wolman, and J. Flinn. Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 151–165. ACM, 2015.

[19] M. Levoy and P. Hanrahan. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42. ACM, 1996.

[20] M. McGuire, M. Mara, D. Nowrouzezahrai, and D. Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, page 2. ACM, 2017.

[21] P. Merkle, Y. Morvan, A. Smolic, D. Farin, K. Mueller, P. de With, and T. Wiegand. The effects of multiview depth video compression on multiview rendering. *Signal Processing: Image Communication*, 24(1):73–88, 2009.

[22] P. Merkle, K. Muller, A. Smolic, and T. Wiegand. Efficient compression of multi-view video exploiting inter-view dependencies based on h. 264/mpeg4-avc. In *2006 IEEE International Conference on Multimedia and Expo*, pages 1717–1720. IEEE, 2006.

[23] P. Merkle, A. Smolic, K. Muller, and T. Wiegand. Multi-view video plus depth representation and coding. In *2007 IEEE International Conference on Image Processing*, volume 1, pages I–201. IEEE, 2007.

[24] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson. Adaptive scalable texture compression. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 105–114. Eurographics Association, 2012.

[25] B. Reinert, J. Kopf, T. Ritschel, C. Eduardo, D. Chu, and H. Seidel. Proxy-guided image-based rendering for mobile devices. In *Computer Graphics Forum: the international journal of the Eurographics Association*, volume 35, 2016.

[26] S. M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski. A comparison and evaluation of multi-view stereo reconstruction algorithms. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 1, pages 519–528. IEEE, 2006.

[27] L. Szirmay-Kalos, B. Aszodi, I. Lazanyi, and M. Premecz. Approximate Ray-Tracing on the GPU with Distance Impostors. *Computer Graphics Forum*, 2005.

[28] G. Valenzise, F. De Simone, P. Lauga, and F. Dufaux. Performance evaluation of objective quality metrics for hdr image compression. In *SPIE Optical Engineering+ Applications*, pages 92170C–92170C. International Society for Optics and Photonics, 2014.

[29] A. Vlachos. Advanced vr rendering performance. Game Developers Conference 2016, 2016.

Algorithm 4 Real-time rendering algorithm

Precondition: Set of N_{view} cubemaps C_j(color), D_j(depth) and their origins $\mathbf{P}_j$. Eye location $\mathbf{o}$ and direction $\mathbf{d}$. Number of raycasting steps N_{step}. Near/far clipping planes $z_{\text{near}}, z_{\text{far}}$. Z conversion functions $\text{lin}(), \text{nonlin}()$ between linear and nonlinear space using eq. 13

Output: A color $\mathbf{c}_{\text{out}}$

function BetweenViewpointAndGeometry($\mathbf{p}_{\text{cur}}, j, \mathbf{P}, D$)
    $\mathbf{x} \leftarrow \mathbf{p}_{\text{cur}} - \mathbf{P}_j$
    $d_{\text{tex}} \leftarrow \text{SampleCubemap}(D_j, \mathbf{x})$
    return $d_{\text{tex}} > \text{nonlin}(||\mathbf{x}||)$

function SampleColor($\mathbf{p}_{\text{cur}}, j, \mathbf{P}, C, D$)
    $\mathbf{x} \leftarrow \mathbf{p}_{\text{cur}} - \mathbf{P}_j$
    $d_{\text{tex}} \leftarrow \text{SampleCubemap}(D_j, \mathbf{x})$
    $c_{\text{tex}} \leftarrow \text{SampleCubemap}(C_j, \mathbf{x})$
    $d_{\text{diff}} \leftarrow |d_{\text{tex}} - \text{nonlin}(||\mathbf{x}||)|$
    $w_{\text{depth}} \leftarrow 1/(d_{\text{diff}} + \varepsilon)$
    $w_{\text{cam}} \leftarrow 1/(||\mathbf{o} - \mathbf{P}_j|| + \varepsilon)$
    return $(c_{\text{tex}}, w_{\text{depth}} \cdot w_{\text{cam}})$

function Main($C, D, \mathbf{P}, \mathbf{o}, \mathbf{d}$)
    $\mathbf{c}_{\text{out}} \leftarrow (0,0,0,0)$ ▷ Initialize output with transparent color
    $s_{\text{nonlin}} \leftarrow \frac{\text{nonlin}(z_{\text{far}}) - \text{nonlin}(z_{\text{near}})}{N_{\text{step}}}$ ▷ Step magnitude
    for $i \in [1, N_{\text{step}}]$ **do** ▷ Non-linear raymarching
        $z_{\text{cur}} \leftarrow \text{lin}(\text{nonlin}(z_{\text{near}}) + i \cdot s_{\text{nonlin}})$
        $\mathbf{p}_{\text{cur}} \leftarrow \mathbf{o} + \mathbf{d} \cdot z_{\text{cur}}$
        $r_{\text{isect}} \leftarrow \text{true}$
        for $j \in [1, N_{\text{view}}]$ **do**
            if $\text{BetweenViewpointAndGeometry}(\mathbf{p}_{\text{cur}}, j, \mathbf{P}, D)$ **then**
                $r_{\text{isect}} \leftarrow \text{false}$ ▷ We are on empty space, continue marching
                break
        if r_{isect} **then**
            break
    if $i = N_{\text{step}}$ **then** ▷ Check if ray did not intersect
        return $\mathbf{c}_{\text{out}}$
    $\mathbf{c}_{\text{sum}} \leftarrow (0,0,0,0)$
    $w_{\text{sum}} \leftarrow 0$
    for $j \in [1, N_{\text{view}}]$ **do**
        $(\mathbf{c}, w) \leftarrow \text{SampleColor}(\mathbf{p}_{\text{cur}}, j, \mathbf{P}, C, D)$
        $\mathbf{c}_{\text{sum}} \leftarrow \mathbf{c}_{\text{sum}} + \mathbf{c}$
        $w_{\text{sum}} \leftarrow w_{\text{sum}} + w$
    $\mathbf{c}_{\text{out}} \leftarrow \frac{\mathbf{c}_{\text{sum}}}{w_{\text{sum}}}$
    return $\mathbf{c}_{\text{out}}$