# Camera Space Volumetric Shadows

Johannes Hanika*, Peter Hillman, Martin Hill and Luca Fascione
Weta Digital Ltd


**Figure 1:** *Render of many hundreds of horses in the desert, with dust and volumetric shadows.*

## Abstract

We transform irregularly sampled shadow map data to deep image buffers in camera space, which are then used to create volumetric shadows in a deep compositing workflow. Our technique poses no restrictions on the sample locations of the shadow map and can thus be used with a variety of adaptive approaches to produce more precise shadows closer to the camera. To construct watertight shafts towards the light source forming crepuscular rays, we use a two-dimensional quad tree in light space. This structure is constructed from the shadow samples independent of the camera position, making stereo renders and camera animations for static light sources and geometry more efficient. The actual integration of volumetric light transport is then left to a fast image space deep compositing workflow, enabling short turnaround times for cinematic lighting design. We show a simple scalable ray tracing kernel to convert the quad tree representation to a deep image for each camera, where ray tracing takes only 25% of the processing time.

**CR Categories:** I.3.3 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing;

**Keywords:** Deep Compositing, Volumetric Shadows, Ray Tracing

*e-mail:jhanika@wetafx.co.nz


DigiPro 2012, Los Angeles, CA, August 4, 2012.
© 2012 ACM 978-1-4503-1649-1/12/0008 $15.00

## 1 Introduction

Deep Compositing is an emerging technique in motion picture visual effects. The process relies on rendering *deep images*, in which each pixel stores an arbitrarily long list of depth-sorted samples. Elements rendered into separate deep images can be combined accurately to produce a single composited image of the entire scene, even if the images are interleaved in depth. This allows for great flexibility, since only elements which have changed need to be re-rendered, and the scene can be divided into elements without needing to consider how they will be combined. By comparison, in traditional compositing with regular images the scene must be divided into elements in such a way that they can be recombined without edge artefacts or depth order violations. Volumetric deep image pixels represent a volume as piecewise constant optical density and color as a function of depth. Volumetric renders of participating media are generally computationally intensive, so the ability to compute them independently and composite them correctly into the scene is highly advantageous.

Where one element casts a shadow on another, the flexibility of deep compositing has previously been reduced significantly, since the shadowed element will need to be re-rendered every time the shadowing element moves to ensure the shadow cast on it is correct.

In this paper, we propose a technique for computing volumetric shadowing data independently of the shadowed objects, and for applying this shadow data to a rendered element at composite time. This means that when a shadowing element moves, shadowed elements do not need to be re-rendered: they are rendered without any shadows from other elements, and the shadows are applied at composite-time. Our technique stores shadow maps computed from the point of view of the camera, rather than the light. This is an overall gain, since it allows the shadowmap to be applied to a deep image at interactive speed, and shadows tend to be applied far more frequently than they are recomputed.

A significant application of our technique is in the application of

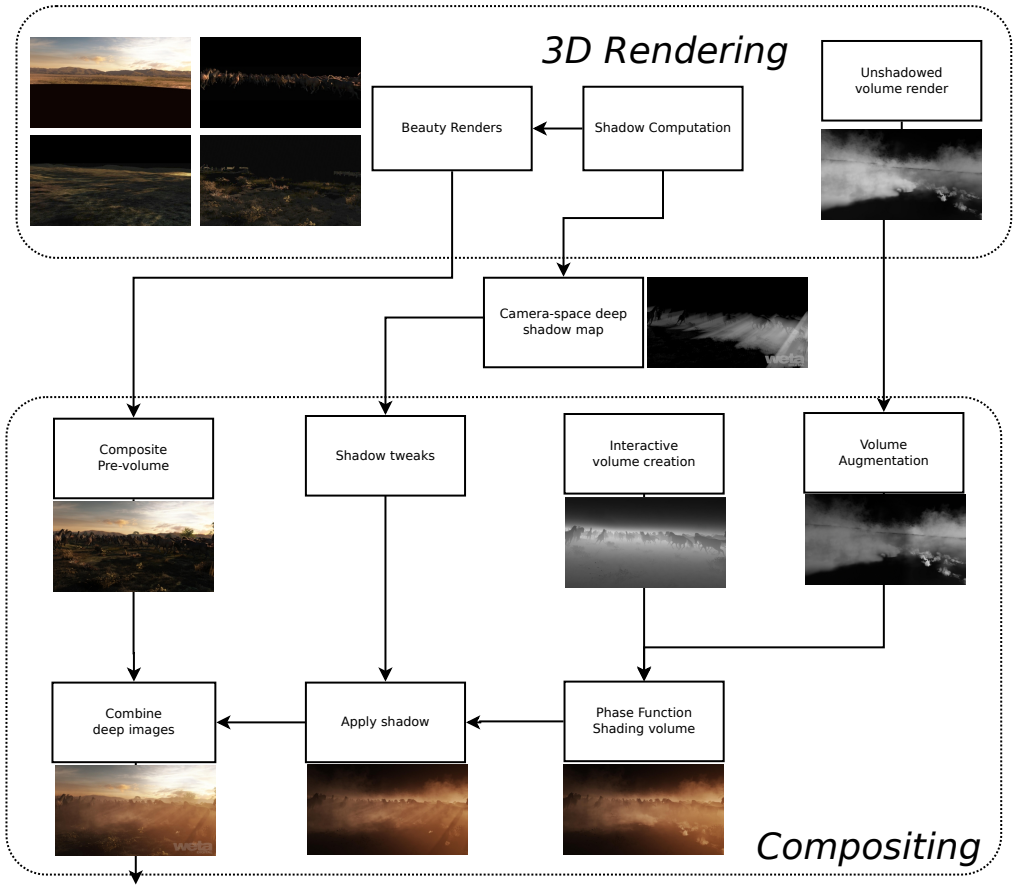


**Figure 2:** *The context of our technique. We use input from 3D rendering, namely a shadow point cloud, and transform it to camera space in form of a deep image. This allows for a very flexible compositing workflow, avoiding costly re-rendering of the 3D volume (Images © 2012 Weta Digital Ltd. All rights reserved).*

crepuscular rays to participating media renders such as dust or fog renders. Elements within the volume will cast shadows through it, creating distinctive "god rays" in the media. With our technique, the rays are computed independently to the volumetric render. For example, the dust cloud of Figure 1 doesn't need to be re-rendered if the horse animation changes, only the horse render and its shadowmap does. As full volumetric renders can have extremely high computational cost, the shot turnaround time is greatly reduced.

For many participating media renders such as atmospheric mist, the volume itself is uniform, low density. The only 'interesting' part of the render is the shadows cast through it by other objects. With our technique, there is no need for a computationally intensive 3D render, since the volume can be created at composite time and adjusted interactively. In a similar fashion volume renders resulting from high resolution simulations, can have their density augmented within the composite in a spatially varying way. This allows for cheap fine tuning of expensive renders, where previously a re-render would be required.

The context visualized in a diagram can be seen in Figure 2. The core of our technique sits between rendering and compositing and converts world space shadow information to camera space by employing a light space acceleration structure. In the following, we briefly discuss deep images and deep compositing and review light-space shadow buffers. We then describe our algorithm for converting light-space shadow buffers. Finally, we explain the process for applying shadow maps to deep images at composite-time.

## 2 Background

**Deep Images** Deep shadow maps were introduced just over a decade ago, to improve the appearance of shadows cast from objects exhibiting various kinds of non-refractive transparency such as hair or smoke with very reduced aliasing [Lokovic and Veach 2000]. This technique has since been adapted for use in compositing [Hillman et al. 2010]. Deep images store a per-pixel list of samples. Each sample has a depth and a number of color channels, usually RGBA. Every sample in the image has the same number of channels. To represent volumes, the sample has an additional *ZBack* channel, allowing the front and back depth of the sample to be represented. The color and optical density of such a sample is assumed to be constant throughout each sample. Where a volume has changes in color and/or optical density, more samples are required per-pixel to represent the density.

To convert a deep image to a regular image, the samples are sorted into depth order and composited from back to front using the *over* compositing operator. To combine two deep pixels from multiple images, the sample lists are simply appended together. Multiple images can be combined in any order, since the sorting operation will guarantee the same result. (This order-independence to compositing is perhaps the most desirable advantage of deep images, since they can often be combined automatically with no regard for the contents of the individual elements). In the case of volumetric samples, overlaps are possible. In this case, the volumetric sample must be sub-divided into subsamples which do not overlap using the Beer Lambert equation.

Deep images can be stored efficiently using the OpenEXR-2.0

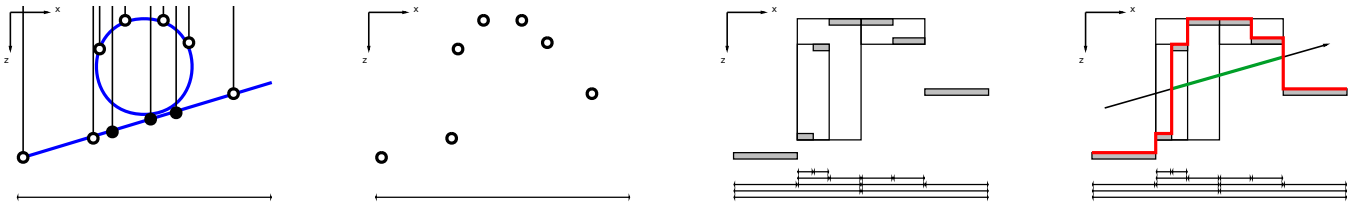

**Figure 3:** *The image (left) shows the geometry transformed into perspective light space $(x, y, z)$ with $z$ coming from the light. The dots visualize the location of the shadow samples, occluded ones (black) are omitted (second image from left). We construct hole-free shafts towards the light by building a quad tree in $(x, y)$ direction. The tree is visualized with the line segments at the bottom, leaves are shown in gray in the third image. In the $z$ direction, each quad tree cell stores an interval. Ray tracing this structure is effectively intersecting a ray with the red surface covering the objects (right image), which allows us to find the shadow volume. The green ray segment represents the data stored in the deep image buffer.*

deepscanline or deeptiled types [Kainz and Bogart 2009] which supports arbitrary channel names and permits separate storage of multiple subimages within the same file for 3D Stereo productions.

Compositing packages such as The Foundry's *Nuke* can read, process and write deep images, and also offer the ability to generate deep images interactively, for example the ability to turn a fractal noise field into a volumetric deep image.

**Shadows** A good overview of recent shadowing techniques in the real-time context can be found in [Eisemann et al. 2011].

A substantial amount of work has been done to improve the artefacts seen in some cases by classic shadow maps [Williams 1978], most of this work lending itself to a characterization in which the methods are essentially novel ways of arranging shadow samples in a buffer. For example, *perspective* shadow maps [Stamminger and Drettakis 2002] pull the samples together towards the camera by distorting the map, and *parallel-split* shadow maps [Zhang et al. 2006] or *cascaded* shadow maps [Dimitrov 2007] achieve a similar result employing multiple maps at different scales, and *adaptive shadow maps* [Fernando et al. 2001] use hierarchical grids for the same reason.

The restrictions on sample locations inside one map are further loosened by *sample distribution* shadow maps [Lauritzen et al. 2011], the *irregular z-buffer* [Johnson et al. 2005] and *imperfect* shadow maps [Ritschel et al. 2008].

*Multilayer transparent shadow maps* [Xie et al. 2007] use a transformation of rays into shadow map space to approximate ray tracing for soft shadows. We, on the other hand, are looking for a fast volume compositing workflow. Also we do not introduce any more approximation over the input, which is a precomputed shadow representation already.

Finally, *alias free* shadow maps [Aila and Laine 2004] use a data structure that is probably most closely related to our approach: They use a light-space 2D BSP-tree to perform adaptive rasterization of the shadow map at exactly the shadow ray's sample position. While we use a similar data structure, we perform ray tracing from the eye on it, in order to obtain shadow volumes in the form of deep data buffers to be used for rendering of participating media.

The most accurate method to compute shadows is ray tracing [Whitted 1980] and has received a vast amount of research over the last thirty years (see e.g. [Havran 2000] for a thourough overview of the subject). We build on the extensive experience of the ray tracing community to implement a very small and simple specialized ray tracing core for almost two-dimensional quad trees.

**Volumetric Rendering** The technique proposed by Engelhardt and Dachsbacher [Engelhardt and Dachsbacher 2010] uses epipolar rectification of camera and light source and achieve real-time performance via sub-sampling. While very reasonable in the context of real-time renders, it was our finding that this method introduces too much blur for the quality targets of cinematic offline renders.

It has been shown that it is possible to even get away with only a one-dimensional min-max mipmap [Baran et al. 2010; Chen et al. 2011]. This approach not only rectifies lines to the light, but also camera rays to gain even more performance. Unfortunately these techniques can suffer from numerical stability issues when the light source is inside the camera's viewing frustum and need to fall back to different approaches for such configurations.

Our scheme is slightly more general, as it is two-dimensional and thus gives us more freedom with respect to shadow sample positions and viewing rays. In particular, it works on the output of completely arbitrarily sampled shadow buffers, including classic, perspective and cascaded shadow maps, as well as ray traced point clouds. Another benefit is that the transformation we propose is independent of the camera, so that our data structure can be reused for both the left and right eye in stereoscopic renders of animation, or even entire frame sequences for static scene configurations.

The only assumption on the light position is that there is a minimum distance between interesting parts of the scene (geometry and camera rays) and the light source (see Section 4 for details). This is similar in spirit to the role of the near clipping plane.

## 3 Algorithm

We start with a high level overview of our proposed technique and follow with implementation details in Section 4. A schematic overview of the process can be found in Figure 3.

**Camera-Space shadow maps** Our shadow maps are rendered from the point of view of the camera. Each pixel stores every shadow intersection of the corresponding ray. Each shadow sample stores its front and back depth, as well as the shadow density, where 0 implies no shadowing and 1 implies full attenuation. Since each ray may pass through multiple shadows, there will be an arbitrary number of shadow samples per pixel. The shadow maps are therefore stored as deep images. Note that, in contrast with many other shadow formats, the shadow density is not cumulative along the ray; rather, the density of each shadow is stored independently.

**Overview** The input is any kind of shadow buffer. In an effort to keep the discussion most general, we assume the data is in the form of a 3D point cloud with shadow information, that is we assume each point in the cloud has data about it being in shadow or light. Classic shadow maps will work as well, and would actually enable a few performance enhancements during quad tree construction.

In order to produce shadow volumes, we need to connect the silhouettes of the shadow casting objects to their respective shadows. To achieve this, we create a connected surface by inserting the sample points in light space $(x, y)$ into a two-dimensional quad-tree, obtaining a closed surface.

We use a quad tree builder as opposed to a meshing algorithm be-

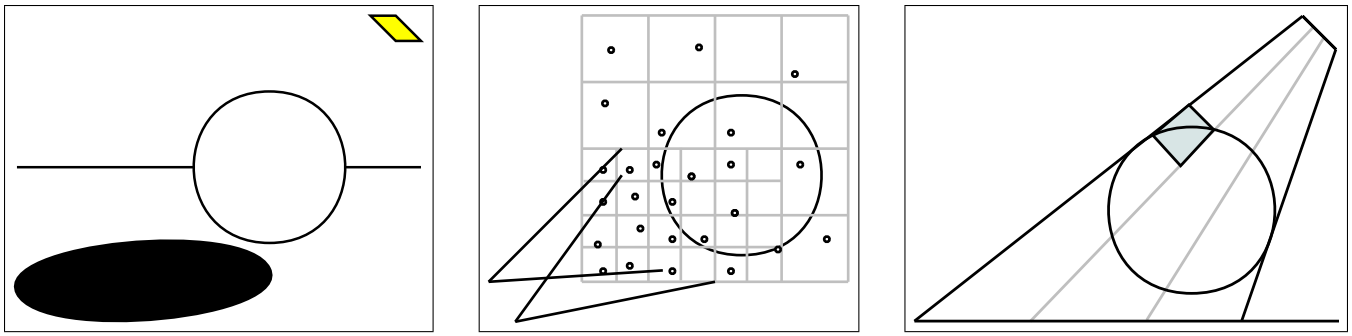

**Figure 4:** *To compute the shadow volume between the object and its shadow (left), we create a set of shadow samples on the geometry and build a min-max quad-tree on this set to fill holes (center). We can do that once per frame and use it for both cameras in a stereoscopic pair, since the transform only depends on the location of the samples. The quad tree cell outlined in the right picture will be classified as unoccluded so that the shadow volume will not stick out of the object.*

cause it is faster to build and traverse. It is unclear that meshing would result in better silhouette edges. The cells of the quad tree live in light $(x, y)$ space, which is mostly smoothed out by integration over the volume scattering terms as observed from the point of view of the scene camera. The characteristic sharp lines at god ray or volumetric shadow boundaries live in the light's $z$-dimension.

Unlike [Aila and Laine 2004], we use a quad tree instead of a BSP-tree to form regular-sized cells, which reduces artefacts in case of low sampling densities.

Each camera ray is transformed to the same perspective light space as the shadow samples, and then intersected with the quad tree in two dimensions, nodes closer to the ray origin being traversed first. The output depends on strict ordering of the samples, which is also the reason why we don't use an object partitioning scheme such as a bounding volume hierarchy, which would necessitate an extra reordering step.

Whenever a ray dives below the height field and then finds its way back to a point in light (that is we have found a ray segment like the one highlighted in green in Figure 3, right), we record a deep sample constituted by the enter and exit positions of the ray in the volume of the shadow.

Once a deep buffer is constructed, the shadows are applied during compositing by a simple multiplication operator. Various look tweaks are possible at this stage, such as reducing the density of the shadow or modifying its effective colour. The shadow operator can easily be implemented so that it runs at interactive speeds.

Further, in the case of a dust cloud or light fog, the volumetric element can be created interactively in the compositing package, so that no 3D volume rendering is necessary at all (for example, we have implemented a plugin for The Foundry's *Nuke* product that permits colour and density variations and the application of a height map to sculpt volumetric deep images).

## 4 Implementation

**Data preparation** First we transform the input data into coordinates in the perspective space of the light source, similar to what would be stored into an ordinary shadow buffer ($x$ and $y$ in a plane orthogonal to the principal direction of the light and $z$ normal to that). Also we discard all points in shadow and obtain a point set that defines a distance field on the light plane. This field can be thought of as a height map on an arbitrarily oriented plane, against which we want to trace camera rays.

If irregular (3D or 2D) sample points are used as input, they need to be transformed with the perspective shadow map transform (as mentioned in Section 3) in order to map $z \in [1, \infty)$ to a normalized range $[0, 1)$. This means that we have to clamp points with $z < 1$ to $z = 1$, in an operation akin to implementing a near clipping plane, imposing a minimum world-space distance (such as 1 cm or maybe 0.1 world units) between the occluders and the light source.

The full perspective transform takes place in this step, in particular $z$ is transformed along with $x$ and $y$, and crucially can't just be the distance to the light source (as is often done for shadow maps to increase precision). This is because the ray tracing that happens in the final step assumes that rays in the mapped space are straight lines, whereas leaving $z$ unchanged would map camera rays into arcs of hyperbolae.

**Quad tree construction** Given a set of points $(x, y, z)$ in perspective light space, we construct a quad tree on the $(x, y)$ dimensions. The tree is represented as a node array of structs containing a child pointer `children` and height bounds `zmin, zmax`.

The array is laid out so that all children of an internal node are consecutive and in a canonical order, so the node field `children` only needs to index into the node array pointing to the first of the children. The other two fields `zmin` and `zmax` are the bounds of the node contents in $z$. Due to the size of the scenes our implementation works on, we need double precision in the input data to help with temporal aliasing of the shadow samples from frame to frame, but even for smaller scenes using double precision helps substantially to reduce banding artefacts.

Tree construction is done by recursively sorting the input point array so that the partitions for each child quad are contiguous. For maximum accuracy, this continues until only one sample is left in a node, or the bounding box of the quad cannot be split in two anymore due to numerical limits.

As we keep track of `zmin` and `zmax`, we don't need the input points in the tree after the construction is completed. As a side effect of this construction scheme, all quad tree cells are filled with a valid bound, and we end up with a watertight height field (as illustrated in Figure 4).

**Quad tree ray tracing** Once the quad tree is constructed, tracing camera rays through it proceeds as follows: first the rays are transformed in a similar manner to how the input points have been. The transformed rays are then intersected against the two-dimensional bounding boxes of the quad tree nodes, which we construct on the fly during traversal. We also keep track of the minimum and maximum ray height $z_{\min}$ and $z_{\max}$ while traversing, so that we can prune the quad tree by comparing the node's $z$-interval with the ray's $z$-interval.

When the ray dives below the height field surface and then finds its way back above it, into the side of the light source, we record a deep sample, storing the ray distance to the previous occlusion event $t_{\text{front}}$, and the ray distance $t_{\text{back}}$ where the ray got out of the shadow volume again together with the maximum distance towards

```
state = in light
push root node to stack
while (1)
{
  compute zmin and zmax ray height in [tmin, tmax)
  if (leaf)
  {
    if (node zmax < ray zmin) // ray below node
      if(state == in light)
        remember transition to shadow,
        state = in shadow
    else if (node zmin > ray zmax) // ray above node
      if(state == in shadow)
        record deep sample, state = in light
    else if (state == in shadow) // ray through node
      record deep sample, state = in light

    if (stack)
      pop stack
    else
      check if there is a last sample to record
      return
  }
  else
  {
    children = sort node.children in reverse ray dir
    foreach child in children
    {
      if (state == in shadow &&  // in shadow already
        node zmax < ray zmin)    // and ray below node
        prune child
      else if (state == in light && // in light already
        node zmin > ray zmax)       // and ray above
        prune child
      else if (ray overlap child in x,y)
        push child
    }
    pop stack
  }
}
```

**Figure 5:** *Pseudo code for the quad tree traversal.* $(x, y, z)$ *has positive* $z$ *pointing away from the light into the scene.*

the light between a shadow casting object and a point on the ray segment $[t_{\text{front}}, t_{\text{back}})$. This last bit of information is used during compositing to infer a rough estimate of in-scattering from multiple bounces.

In more detail, the algorithm proceeds as in the pseudo-code presented in Figure 5, where the traversal stack holds an index to the current node, its bounding box in two dimensions (created on the fly by recursively splitting the scene bounding box), and the currently valid ray segment $[t_{\min}, t_{\max})$.

We initialize the ray state as in light, so that a shadowed camera will create a deep sample to enter the shadow volume right away. This is necessary for correct compositing.

A certain amount of care is needed in determining when to record an entry and exit point into the shadow volume $[t_{\text{front}}, t_{\text{back}})$, to avoid artefacts at object boundaries. The third branch in the leaf handling code, where a ray pierces right through a leaf node, slightly biases the ray towards preferring to be in light rather than in shadow. This has two implications: Firstly it avoids switching back and forth when passing a surface close to grazing angle. Secondly objects are most often modelled as closed meshes so that the shadow volumes should not stick out of the light side of it. Our biasing ensures that the start of the volume will be inside the object, resulting in a clean transition on the shadow side (see Figure 4, right, for an example configuration).

**Applying shadow volumes** Shadow volumes are applied to deep images at composite time. Since both images are camera aligned, processing is pixel aligned. That is, to process pixel $ij$ of the deep image, only pixel $ij$ of the shadow map is required. Image samples which are entirely within a shadow sample are simply multiplied by the optical inverse of the shadow value. In the case that an image sample overlaps a deep sample (*i.e.* is partly inside a shadow sample and partly outside it) the sample must be split into two samples at the edge of the shadow sample, and each part attenuated accordingly.

The shadow map can be adjusted to apply artistic tweaks before application, for example selectively softening it, or adjusting its colour.

| | |
|---|---|
| read input | 20.89 s |
| construct quad tree | 4.19 s |
| ray trace | 15.93 s |
| | (1.88Minters/s/core) |
| write output | 40.24 s |
| total | 81.25 s |

**Table 1:** *Timing breakdown for the scene in Figure 6, with a screen resolution of 2048×1152, supersampled with a 3×3 pattern. For more discussion about the ray tracing performance, see the last paragraph in Section 5. The overall run time is I/O-bound (the output here was 400MB, but these files can exceed 3GB). Timings were done on a dual quad-core Intel E5620 at 2.4GHz.*

## 5 Results

The results of employing our technique can be seen in Figures 1 and 6, for a medium-sized scene. It does not contain hero characters and all small plants have been removed from Figure 6 for clarity of the illustration. There are 3.2 million irregularly spaced input shadow samples. These are obtained from a point cloud processed with PantaRay [Pantaleoni et al. 2010] to result in more precision than a shadow map. We process this example for one camera front to end in just over 80 seconds, a detailed timing breakdown can be found in Table 1. We compute just under 2 million deep intersections per second and per core in this example. Some care should be taken in comparing this figure to state-of-the art ray tracing cores [Aila and Laine 2009; Ernst and Woop 2011]. An accurate comparison will take into account that our figure includes transformation to and from perspective light space, that we use double precision for all computation and most importantly that our ray cast operation collects and returns all intersections along a ray, not only the first one.

The design of our proposed algorithm is quite robust with respect to increases in depth complexity $d$ of a geometry-camera configuration (it is not uncommon for a scene to have an average $d$ between 10 and 30). Switching on the ground cover (grass and small plants) in the scenes of Figures 1 and 6 raises the number of intersections per second per core from $1.88M$ to $2.2M$ as an effect arising from the increased average depth complexity. This effect is most pronounced when the camera is closer to the ground, as one would imagine.

## 6 Limitations and Extensions

If irregular shadow samples are used, our technique needs a background plane to be present, to be able to distinguish between holes between samples that should be filled by the quad tree, and holes that are due to geometric features (see Figure 6, top right). It is indeed rather easy to insert these automatically when needed and this is part of our setup.

Motion blur and soft area light shadows are not reflected in our shadow volumes. While both can be re-introduced by clever deep blur techniques exploiting more detailed annotations in the deep samples (such as the distance to the closest occluder and its motion

vector), these were not wanted by the compositing artists for this show. Here, god rays should be sharp and crisp in the final image. We are introducing this, however, for more recent productions.

While the output deep shadow map can easily support transparent objects, the input as described here does not. To extend the method to support this, some sort of depth peeling from the light source can be applied, and the proposed transformation can be performed several times.

Very large scenes (bigger than system RAM) could be facilitated by a bucketing step before tree building [Pantaleoni et al. 2010], and forcing an adaptive early construction termination, to make the final tree fit in memory again.

## 7 Conclusion

We described a simple ray tracing technique which is useful to create deep shadow volumes for stereoscopic camera setups for large input scenes. It works with irregular shadow samples and thus offers very fine control over the local resolution of the results. The ray tracing stage is negligibly fast compared to the final rendering and delivers a marked increase in artistic freedom during the compositing stage for cinematic lighting design.

The implementation is only a few hundred lines of code and can reuse the acceleration structure for multiple cameras. The two-dimensional height field formulation allows practically every constellation of light and camera, the only restriction being a minimum distance between the light and the first shadow caster, similar to a near clipping plane.

## Acknowledgments

We thank Joe Letteri and Sebastian Sylwan for supporting our work within Weta Digital Ltd and to Julian Bryant for suggesting composite-time deep shadows. We are also grateful to Twentieth Century Fox Film Corporation for allowing us the use of imagery from the movie *Abraham Lincoln: Vampire Hunter*. Teaser image courtesy of Weta Digital Ltd 

## References

AILA, T., AND LAINE, S. 2004. Alias-free shadow maps. In *Proc. Eurographics Symposium on Rendering 2004*, 161–166.

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, 145–149.

BARAN, I., CHEN, J., RAGAN-KELLEY, J., DURAND, F., AND LEHTINEN, J. 2010. A hierarchical volumetric shadow algorithm for single scattering. In *ACM SIGGRAPH Asia 2010*, 178:1–178:10.

CHEN, J., BARAN, I., DURAND, F., AND JAROSZ, W. 2011. Real-time volumetric shadows using 1d min-max mipmaps. In *Proceedings of the 2011 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D 2011.

DIMITROV, R., 2007. Cascaded shadow maps. `http://developer.download.nvidia.com/SDK/10.5/opengl/src/cascaded_shadow_maps/doc/cascaded_shadow_maps.pdf`.

EISEMANN, E., SCHWARZ, M., ASSARSSON, U., AND WIMMER, M. 2011. *Real-Time Shadows*. A.K. Peters.

ENGELHARDT, T., AND DACHSBACHER, C. 2010. Epipolar sampling for shadows and crepuscular rays in participating media with single scattering. In *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, I3D '10, 119–125.

ERNST, M., AND WOOP, S., 2011. Embree. `http://software.intel.com/en-us/articles/embree-photo-realistic-ray-tracing-kernels/`.

FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. 2001. Adaptive shadow maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, 387–390.

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

HILLMAN, P., WINQUIST, E., AND WELFORD, M., 2010. Compositing "Avatar". SIGGRAPH 2010 Talks.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer: Hardware acceleration for irregular data structures. *ACM Trans. Graph. 24*, 4, 1462–1482.

KAINZ, F., AND BOGART, R., 2009. A technical introduction to openexr. `http://www.openexr.com/TechnicalIntroduction.pdf`.

LAURITZEN, A., SALVI, M., AND LEFOHN, A. 2011. Sample distribution shadow maps. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, 97–102.

LOKOVIC, T., AND VEACH, E. 2000. Deep shadow maps. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, 385–392.

PANTALEONI, J., FASCIONE, L., HILL, M., AND AILA, T. 2010. Pantaray: fast ray-traced occlusion caching of massive scenes. *ACM Transactions on Graphics (Proc. SIGGRAPH 2010) 29* (July), 37:1–37:10.

RITSCHEL, T., GROSCH, T., KIM, M. H., SEIDEL, H.-P., DACHSBACHER, C., AND KAUTZ, J. 2008. Imperfect shadow maps for efficient computation of indirect illumination. In *ACM SIGGRAPH Asia 2008*, SIGGRAPH Asia '08, 129:1–129:8.

STAMMINGER, M., AND DRETTAKIS, G. 2002. Perspective shadow maps. In *Proceedings of ACM SIGGRAPH*, Annual Conference Series, 557 – 562.

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23*, 6, 343–349.

WILLIAMS, L. 1978. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '78, 270–274.

XIE, F., TABELLION, E., AND PEARCE, A. 2007. Soft shadows by ray tracing multilayer transparent shadow maps. In *Proc. Eurographics Symposium on Rendering 2007*.

ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In *Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, VRCIA '06, 311–318.

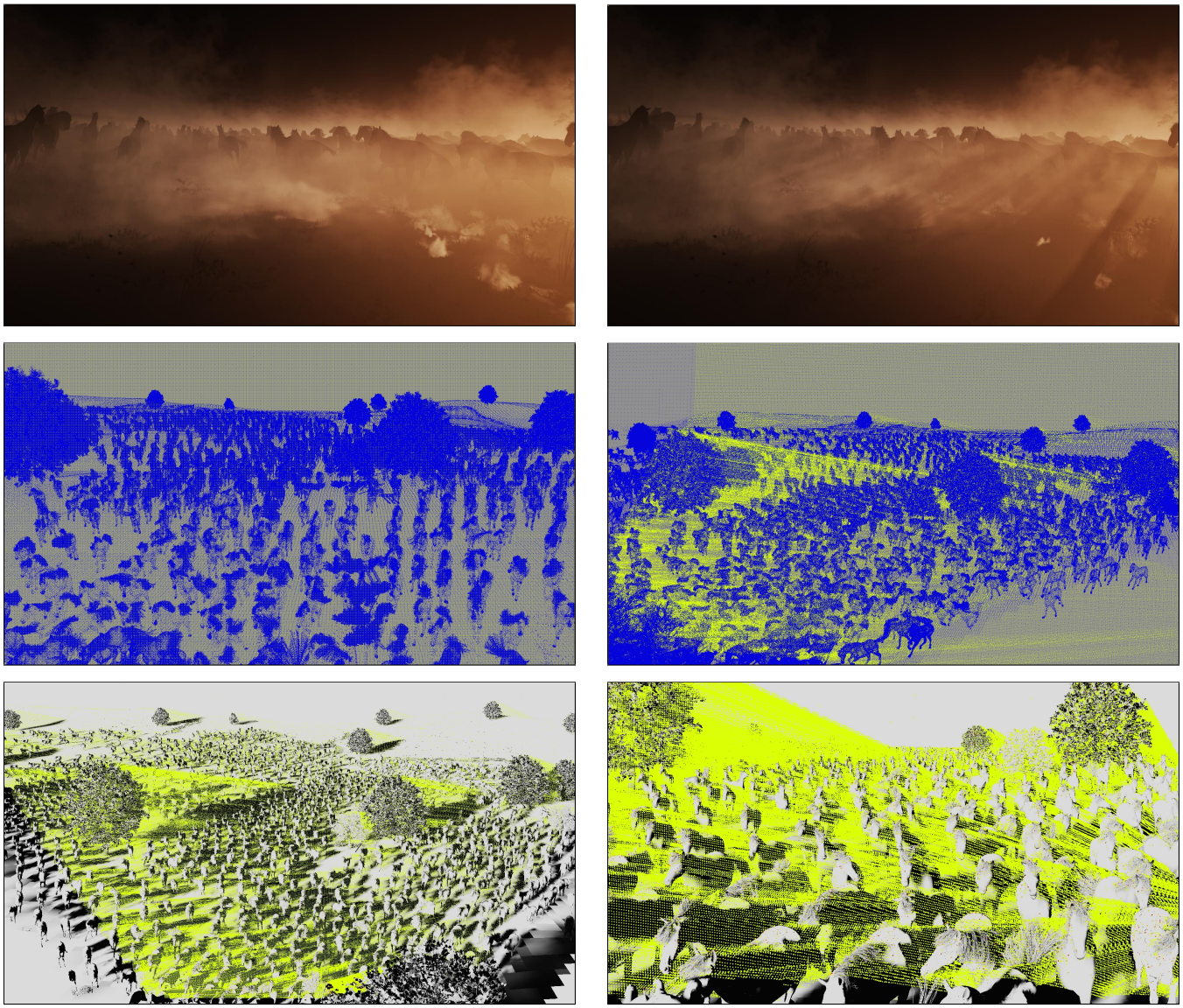



**Figure 6:** *Top row: visualization of the volume without and with shadows. Middle left: visualization of the input shadow samples (blue) and the resulting deep samples (yellow), as seen from the corresponding camera. The deep samples line up with the pixels, so the shadow volumes can't be seen. If we move the visualization camera (right) they become visible. Bottom row: the same but with shadows visualized, to illustrate how they line up with the shadow volumes. For clarity, all smaller scale plants and grass have been removed.*