CGI2022 manuscript No.
(will be inserted by the editor)

Deep Hybrid Order-Independent Transparency

Grigoris Tsopouridis · Ioannis Fudos · Andreas-Alexandros Vasilakis

Abstract Correctly compositing transparent fragments is an important and long-standing open problem in real-time computer graphics. Multifragment rendering is considered a key solution to providing high-quality order-independent transparency at interactive frame rates. To achieve that, practical implementations severely constrain the overall memory budget by adopting bounded fragment configurations such as the k-buffer. Relying on an iterative trial-and-error procedure, however, where the value of k is manually configured per case scenario, can inevitably result in bad memory utilization and view-dependent artifacts. To this end, we introduce a novel intelligent k-buffer approach that performs a non-uniform per pixel fragment allocation guided by a deep learning prediction mechanism. A hybrid scheme is further employed to facilitate the approximate blending of non-significant (remaining) fragments and thus contribute to a better overall final color estimation. An experimental evaluation substantiates that our method outperforms previous approaches when evaluating transparency in various high depth-complexity scenes.

Keywords visibility determination · multifragment rendering · k-buffer · order independent transparency · deep learning

G. Tsopouridis · I. Fudos
University of Ioannina, Dept of Computer Science and Engineering, Ioannina, Greece

A.A. Vasilakis
University of Ioannina, Dept of Computer Science and Engineering, Ioannina, Greece, and Athens University of Economics and Business, Dept of Informatics, Athens, Greece

Corresponding author: Ioannis Fudos, fudos@uoi.gr

1 Introduction

Multifragment rendering (MFR) has become a key solution to facilitate real-time support of complex effects [22] ranging from order-independent transparency (OIT) [9] to global illumination [18]. To achieve real-time performance several MFR algorithms have been used that maintain a memory space per pixel, where information regarding *all* fragments for this pixel is stored. This rendering pipeline is known as A-buffer [4] and can be implemented either by allocating a predetermined memory portion per pixel that can accommodate all fragments [7] which is fast but memory demanding, or by creating per pixel linked-lists [25, 19], which is much slower but needs exactly the required memory space.

In practical scenarios, only a small portion of the fragment pool is enough for achieving plausible simulation effects, such as OIT rendering. The k-buffer [2, 20] can objectively be considered as the most preferred framework using a k-subset selection of all generated fragments, especially when low graphics memory requirements are of the utmost importance. Unfortunately, the standard practice of using a fixed value for k across the entire image can lead to various issues [23]. Firstly, setting k to a small number, can result in view-dependent artifacts as more than k fragments might be required for some pixels at a particular viewing configuration to correctly simulate the desired effect. Secondly, employing a large value of k can result in bad utilization of the allocated space as unused storage is unnecessary allocated for pixels that contain less than k fragments.

In this paper we introduce the first machine learning MFR approach to dynamically adjust and unevenly distribute k values across the image according to the available storage space. Our method utilizes a deep learning mechanism that aims at optimizing the fragment distribution without exceeding the allocated storage, so

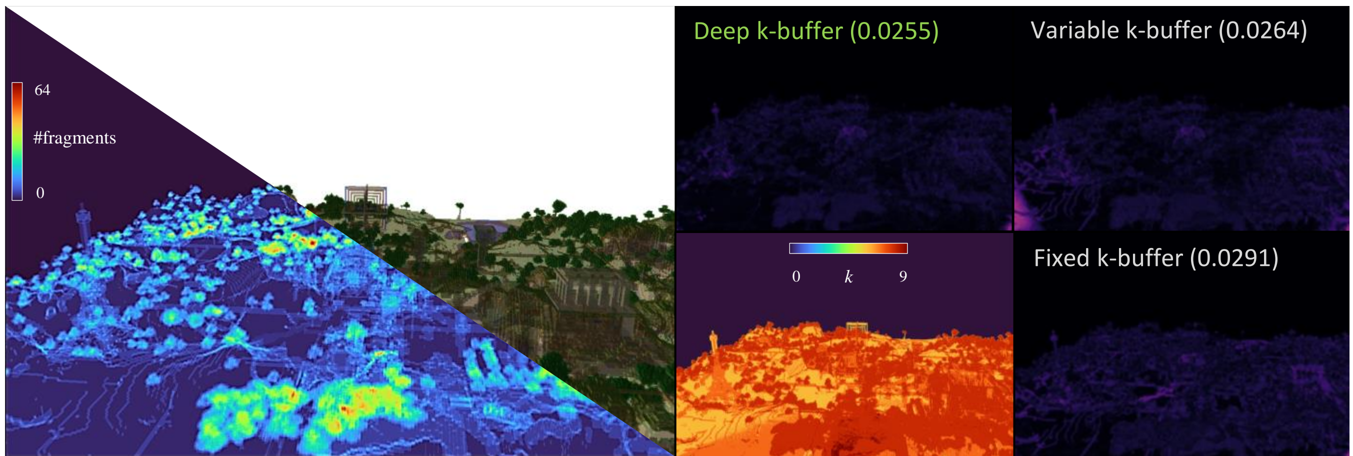

Fig. 1 Our deep multifragment rendering improves rendering quality (as illustrated using ꟻLIP tool [1]) of transparent primitives in high depth-complexity scenes (left) when compared to modern memory-bounded k-buffer approaches (right). Our method distributes wisely a predefined memory budget using a novel neural network predictor that assigns fragment space to pixels in a non-uniform manner (middle).

as to achieve an OIT rendering effect that is as close as possible to the optimal image output (produced by any A-buffer implementation [22]). The per-pixel k distribution is computed by a fast neural network, that is then passed on to the rendering process before the actual fragment data generation. A hybrid rendering approach [8] is applied to perform accurate compositing of the k captured fragments per pixel while applying a quick approximation for the remaining, if any, discarded ones. After training on a sufficient number of high depth-complexity scenes, we show that our network has the ability to effectively approximate OIT for different scene configurations that it has not encountered during training. Although we have used hybrid transparency as our testing scenario, we can apply the same methodology for simulating any OIT variant techniques or for capturing rendering effects that can benefit from using relatively shallow k-buffers [22].

The structure of the rest of this paper is as follows. Section 2 provides a survey of related work. Section 3 describes the core method that is built around a deep learning method for predicting the optimal allocation of fragments per pixel based on hybrid transparency. Section 4 provides a thorough comparative experimental evaluation of our method as compared to previous competent k-buffer techniques that have been presented in the literature. Finally, Section 5 offers conclusions and future research directions.

2 Related Work

Multifragment Rendering. A rapidly increasing number of applications rely on MFR solutions to develop visually convincing graphics applications with dynamic

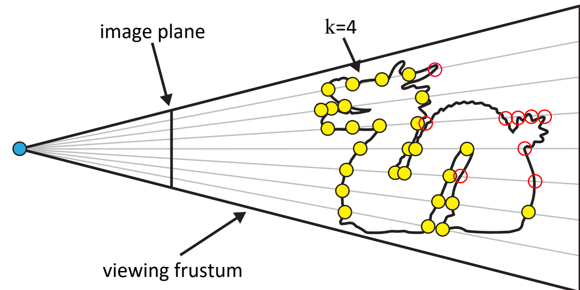

Fig. 2 Hybrid transparency [8] combines an accurate composition of the "core" fragment samples captured in a k-buffer (yellow color), with a quick approximation for the remaining "tail" ones (red color).

content [22]. The main advantage of these approaches is that they encompass additional rasterized geometry, by retaining more information from the fragment sampling domain, thus augmenting the primary visibility determination stage - commonly performed via the Z-buffer [5]. Historically, the most prominent example of a method that went beyond the single layer of rasterized geometry was A-buffer by Carpenter [4]; a software implementation of multiple per-pixel linked lists for transparency and antialiasing purposes. Despite the popularity of the modern, GPU-implemented, A-buffer [25] and its variants [7,19], this class of methods suffers from memory overflows as a result of the unpredictable memory space needed to store all generated fragments making it not appropriate for interactive applications with low memory demands.

***k*-buffer** - Limited per pixel number of fragments. k-buffer can be considered as the most preferred framework for optimal fragment subset selection, especially when low graphics memory requirements are of the ut-

most importance [2,20]. It reduces the computation cost by capturing the best k-subset of all generated fragments, usually the closest to the camera (Fig. 2). The idea is to bound fragment storage (per pixel) to allocate less memory than the A-buffer variants by sacrificing quality to a small extent. k-buffer assumes a preassigned, and global, fixed value of k fragments across the entire image. From a development and production standpoint, the process of finding the optimal value of k that correctly captures the user intent, while keeping memory budget low, can become very challenging. To cope with this issue, the dynamic k-buffer [21] automatically determines the value of k by performing a depth complexity histogram analysis. However, it is clear that using a single value for k for all pixels results in bad utilization of the allocated space. To this end, Vasilakis et al. altered the classic k-buffer construction procedure from a fixed-k to a variable-k per-pixel fragment allocation strategy [23]. Given a fixed memory budget, the idea is to dynamically distribute more fragments in pixels that contribute more to the visual result according to importance-driven metrics. Our work advances this line of research by performing a non-uniform per-pixel fragment allocation guided by a machine learning prediction mechanism that approximates the optimal fragment distribution under a fixed memory budget. Despite the recent trends of neural rendering [16], and its application to enhance screen-space global illumination [11,17,26], the use of machine learning for predicting multifragment storage has not been addressed so far.

Order-Independent Transparency. MFR is actually the tool that spurred research on order-independent transparency [9,24]. Transparent surfaces require multiple per-pixel fragments to be captured in sorted back-to-front order and then evaluated using a compositing operator [12]. To avoid GPU resource exhaustion from an A-buffer, a k-buffer scheme is commonly applied to capturing only the most important information and compactly representing the resulting transmittance as a function of depth [14,8,15]. Without loss of generality, we train our artificial network using the hybrid transparency approach [8] where the transmittance function is accurately computed for the closest to the viewer fragments and approximated with a fast weighted average blending [3] for the remaining ones.

3 Deep k-buffer Method

In this section, we analytically describe the two phases of our deep hybrid OIT technique; i) the *training* phase, and ii) the *runtime* phase. We introduce a “one network for one effect” approach for training the network to learn the per-pixel relation between features and the number of fragments necessary for a particular rendering effect, specifically the Hybrid Transparency method [8]. Without loss of generality, we can apply the same training methodology with minor feature changes for simulating rendering effects, even outsize the OIT spectrum [24], that can benefit from using relatively shallow k-buffers [22].

For the training phase (Sec. 3.1), we first explain how we acquire the necessary screen-space feature data for the supervised learning process (Sec. 3.1.1), along with a heuristic process that generates the optimal k-buffer for each case which we use as our desired ground truth output (Sec. 3.1.2). We then describe the architecture of the neural network (Sec. 3.1.3) used for learning how to distribute memory along the image space. During the runtime phase (Sec. 3.2), the network generates k per-pixel values for new scene configurations at interactive rates, with a similar quality to the optimal k-buffer used for training, and using only image space buffer information (Sec. 3.2.2). Finally, we discuss how the predictor is practically integrated into a modern MFR pipeline (Sec. 3.2.1) that simulates hybrid transparency (Sec. 3.3).

3.1 Training Phase

3.1.1 Data Acquisition

To train the neural network, we created a dataset that contains pairs of per-pixel inputs and the target pixel importance. To create the dataset, input features from 8 fully-transparent scenes with varying depth complexity (10-30 fragments), each containing a single 3D object, are exported from our renderer. Each scene has eight different camera views, which were manually set and differ in terms of position and direction, and are shared amongst all scenes. Finally, each camera view has five different memory allocation variations ranging from having the bare minimum of one fragment per pixel to having all scene fragments, to include all possible user-defined memory budget setups. The neural network expects the following 12 float features as input:

- Per-pixel p
  - Number of pixel fragments divided by the total number of fragments in the specific scene/view, $\frac{n(p)}{\sum_p n(p)}$.

  - Nearest fragment depth, $d_n(p)$.
  - Farthest fragment depth, $d_f(p)$.
  - Diffuse color of the nearest fragment, $C_n(p)$.
  - Average diffuse color of pixel fragments, $C_{avg}(p)$.
- Image-based
  - Number of allocated fragments (based on the available memory budget M) divided by all generated fragments, $\frac{F(M)}{\sum_p n(p)}$.
  - Nearest fragment depth, $d_n = \min_p d_n(p)$.
  - Farthest fragment depth, $d_f = \max_p d_f(p)$.

We have tried out several other features as well, but they were rejected because they had no correlation with the target optimal distribution. In this paper, we focus on predicting correctly the optimal distribution of Section 3.1.2. Therefore, geometry information (e.g. normal vector) does not help in predicting the optimal distribution. However, one could seek to predict the optimal A-buffer color per pixel. This is a complex problem that requires further investigation.

As the desired output, we compute the optimal k-buffer distribution under fixed memory budget for each scene/view variation (Sec. 3.1.2) using Hybrid Transparency (Sec. 3.3) and we divide it by the total number of available fragment spots to derive the desired per-pixel importance. We perform uniform random sampling and keep only 20% of the exported input-output pairs for the training process in an effort to avoid overfitting. All input-output pairs in the dataset were exported using a resolution of 1430x960 pixels but any other resolution can be used without re-training, as prediction works on a per-pixel basis.

3.1.2 Optimal Fragment Distribution

We introduce a heuristic method for deriving the optimal fragment distribution under fixed overall memory budget for the case of Hybrid Transparency [8]. The optimal fragment construction, called *optimal k-buffer* in the context of this paper, is the distribution of a fixed number of fragments (based on the given memory resources) to pixels that minimizes the mean square error of hybrid transparency as compared to the correct result derived from an unbounded memory MFR method for the specific view/scene that takes into account all fragments [25,19].

Let P be the set of pixels in an image of resolution $w \times h$ derived from a scene under a specific view. Therefore, $|P| = w\,h$. For each pixel p, we assign a depth-sorted list of all fragments available for this pixel with unbounded memory. We denote this list by $FL(p) = [f_1, \ldots, f_{n(p)}]$ and its length as $|FL(p)| = n(p)$.

Let $K_u : P \to \mathbb{N}$ be the mapping that represents the number of fragments per pixel with unbounded memory for a specific scene and view. Note that $K_u(p) = n(p)$. Let $K_M : P \to \mathbb{N}$ be the mapping that represents the distribution of fragments to pixels under a fixed memory budget M. Note that when $K_M(p) < K_u(p)$ this means that under the fixed memory budget we shall compute for this pixel the hybrid OIT by considering the first $K_M(p)$ elements of $FL(p)$ as core and the rest of the $FL(p)$ elements as tail (Sec. 3.3). For the two mappings the following hold: $K_u(p) = 0 \Rightarrow K_M(p) = 0$ and $K_u(p) > 0 \Rightarrow K_M(p) > 0$. Then $\sum_{p \in P} K_M(p) = F(M)$, where $F(M)$ is the number of fragments available under a memory budget M. $F(M)$ is computed by the following formula:

$$F(M) = \left\lfloor \frac{M - |P|\,S_p - S_i}{S_f} \right\rfloor \quad (1)$$

where S_p and S_i is the total memory required by our deep OIT method for each pixel and for storing information about the entire image respectively (Sec. 3.2.1). Finally, S_f is the memory required for storing information for each fragment. For 32-bit GPUs this is 8 bytes (two single precision floats).

The optimal distribution of fragments K_M is the distribution that minimizes the sum of differences of two factors:

1. the color $C(p, K_M)$ of each pixel p computed using hybrid OIT with fragment distribution K_M on $FL(p)$ by considering the first $K_M(p)$ elements of $FL(p)$ as core and the rest as tail (more details in Sec. 3.3).
2. the color $C_u(p)$ of each pixel p using OIT on all generated fragments (unbounded memory budget) of $FL(p)$.

Therefore the objective function is computed using this formula:

$$\sum_{p \in P} MSE(p, K_M) \quad (2)$$

where $MSE(p, K_M) = (C_u(p) - C(p, K_M))^2$.

The optimal distribution of fragments K_M is computed by a backward greedy algorithm that starts with the exact unbounded OIT scheme and removes at each step the farthest from the viewer fragment that causes the minimum increase in Equation 2 (Fig. 3). To this end, we present Algorithm 1 that uses a min-heap initialized by the increase in error caused by removing the largest depth fragments of all pixels. Then the fragment from the top of the heap is removed from the scheme and the fragment that has the next largest depth for that pixel is inserted in the heap (except if this is the

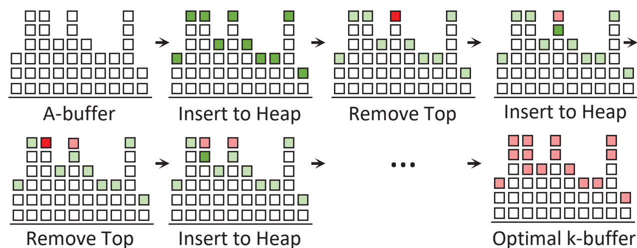

Fig. 3 Optimal k-buffer construction: Starting from an A-buffer construction [19] with iteratively remove (move from core to tail [8]) the farthest fragment from the viewer that contribute less to the final hybrid OIT result.

last remaining fragment for this pixel). The algorithm stops when we have reached the overall number of available fragment spots $F(M)$. Please note that the computation of the optimal fragment distribution is conducted offline for training the neural network that predicts the pixel importance (Sec. 3.1.1).

Lemma 1 *Alg. 1 derives the optimal fragment distribution* K_M.

Proof Since the list of fragments for each pixel is independent from all other pixels, and the fragment distribution determines how many of the first depth-sorted fragments we will use from each list as core fragments, it is straight forward to prove the correctness of the algorithm by induction on the number of removed fragments.

Step 1: When removing the first fragment we choose to remove a fragment from a pixel that has the minimal error $MSE(p, K_M)$ among all candidate pixels (with two or more fragments) when $K_M(p)$ is decreased by 1.

Step 2: If we have removed so far n fragments from various pixels and this is the optimal configuration, then at the next step we should remove a fragment from a pixel that has the minimal $MSE(p, K_M)$ among all pixels when $K_M(p)$ is decreased by 1. The element at the top of the list has the minimal such value among all eligible candidates (i.e. with two or more core fragments).

Input: All fragments of a specific (scene, view) instance for resolution $w \times h$, P, M, K_u, C_u, C, F, FL

```
1  K_M = K_u;
2  f_M = F(M), via Eq. 1
3  f_c = Σ_{p∈P} K_u(p);
4  Initialize min-heap H;
5  for every p ∈ P do
6  |  HeapInsert(H,K_M,p);
7  end
8  while f_c > f_M do
9  |  extract top node from H: (MSE(p,K_M),p);
10 |  K_M(p) = K_M(p) − 1;
11 |  f_c = f_c − 1;
12 |  HeapInsert(H,K_M,p);
13 end
   Output: Return K_M
14
   Function HeapInsert(H, K, p):
15 |  if K(p) ≥ 2 then
16 |  |  K(p) = K(p) − 1;
17 |  |  insert (MSE(p,K),p) to H with key
   |  |    MSE(p,K), (with Eq. 2)
18 |  |  K(p) = K(p) + 1;
19 |  end
20 End Function
```

Algorithm 1: Algorithm for obtaining the optimal distribution of fragments K_M under a fixed memory budget M.

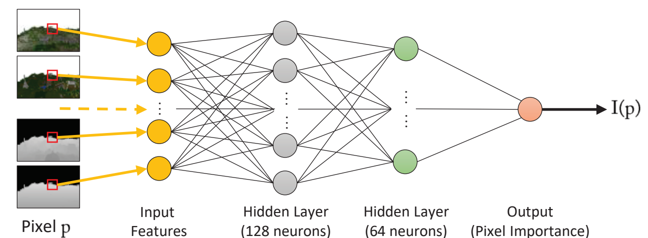

Fig. 4 A schematic of the neural network exploited to learn a mapping from image-space buffers to target per-pixel importance.

3.1.3 Network Architecture & Training

We have selected a simple fully connected multilayer perceptron (Fig. 4) to predict an unnormalized per-pixel importance value that helps us compute the final per-pixel k value allocation. The neural network is composed of an input, an output, and two hidden layers. The two hidden layers are composed of 128 and 64 neurons respectively and use ReLU as their activation function. Furthermore, to reduce over-fitting, L1 and L2 regularization are applied on the first and second hidden layers respectively. The output layer returns a per-pixel importance value $I(p) \in [0, 1]$ by using the sigmoid activation function. In the training step of the network, Mean-Squared Error is used as a loss function and Stochastic Gradient Descent with a learning rate set to 0.001 as an optimizer. Finally, a batch size of 512 is used.

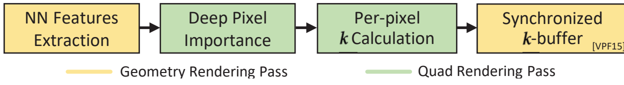

Fig. 5 The rendering pipeline of the proposed deep k-buffer.

3.2 Runtime Phase

3.2.1 MFR Pipeline

Our deep MFR method operates in four main steps similar to the variable k-buffer pipeline [23] as illustrated in Figure 5. We consider the memory fixed, pre-allocated and linearly organized into variable contiguous regions of length $k(p)$ for each pixel p. The method comprises two geometry rendering passes and two quad rendering passes. A fast, geometry rendering pass (*NN Features Extraction*) is responsible for computing and storing the neural network input buffer features (Sec. 3.1.1), such as the number of fragments per pixel which could be accumulated using atomic operations, as they do not require any complex order-dependent operations (similar to how deferred shading works). Then, the pixel importance $I(p)$ and $k(p)$ are calculated in parallel via two consecutive full-screen quad passes (*Deep Pixel Importance* and *Per-pixel k Calculation*), stored in an 2D texture per se, and passed on to the final geometry rendering stage. Finally, the dynamic k^+-buffer implementation [21] is adapted for capturing the $k(p)$-closest fragments (*Synchronized k-buffer*) while ensuring correct hybrid transparency computation for the discarded fragments (Sec. 3.3).

3.2.2 Prediction of per-pixel k

The k value of each pixel p, $k(p)$, can be dynamically assigned on a per-pixel basis according to the pixel importance, $I(p)$, predicted by the trained neural network without exceeding the pre-allocated memory budget. Since these values are computed locally (per pixel), we have to adjust them in image space by performing a normalization step. We accumulate the unnormalized, predicted importance values in an atomic value $I_{sum} = \sum_p I(p)$ to get the importance-based probability $IP(p) = I(p)/I_{sum}$ of each pixel p. If $F(M)$ is the number of fragments available under a memory budget M (Eq. 1), then the final value $k(p)$ is computed using this simple formula:

$$k(p) = \lfloor F(M)\, IP(p) \rfloor \tag{3}$$

3.3 (Deep) Hybrid Transparency

Hybrid transparency [8] divides the transparent fragments into two subsets depending on the importance of their contribution to the final pixel color. One subset consists of fragments with major importance to the final color (*core*) and are blended with accurate, multi-fragment rendering, algorithms that require sorting and have higher memory and computational demands. The remaining transparent fragments constitute the second subset (*tail*) that is deemed less important to the final color and can be blended using less accurate but faster, approximate OIT algorithms. The two subsets are finally combined with the opaque background to produce the final pixel color.

The core is the subset of transparent fragments of a pixel that has the main contribution to the pixel color and is extracted using any k-buffer approach [22]. In our work, the core is consisted of $k(p)$ fragments per pixel, as allocated by our neural network prediction mechanism. If a pixel has $n(p)$ fragments to combine, all fragments not in the core, $n(p) - k(p)$, compose the tail subset. The core and tail colors are calculated similarly to the original paper [8].

The visibility v_i of core fragments is defined using the opacities α of the depth-sorted fragments by using the over operator [12].

$$v_i = \begin{cases} \alpha_1 & \text{if i} = 1 \\ \alpha_i(1 - \sum_{j=1}^{i-1} v_j) & \text{if } i > 1 \end{cases}$$

For the core subset, each color (C) of the $k(p)$ core predicted fragments is weighted by the visibility and summed into a new core color layer (C_{core}). The opacity of the core layer (α_{core}) is the sum of all visibilities of all the $k(p)$ core fragments.

$$C_{core} = \sum_{i=1}^{k(p)} (C_i\, v_i), \quad \alpha_{core} = \sum_{i=1}^{k(p)} v_i$$

The tail fragment color is calculated with a quick approximation. Weighted Average [3] is used to accumulate tail fragment colors (C_i) weighted by their own opacities (α_i) and accumulates their opacities as well.

$$C_{accum} = \sum_{i=k(p)+1}^{n} (C_i\, \alpha_i), \quad \alpha_{accum} = \sum_{i=k(p)+1}^{n} \alpha_i$$

Then, the accumulated color is divided by the accumulated opacity, producing the tail color (C_{tail}). Weighting average uses the number of tail fragments $t = n(p) - k(p)$ to distribute the accumulated visibility equally among all the tail fragments of the pixel.

$$C_{tail} = \frac{C_{accum}}{\alpha_{accum}}, \quad \alpha_{avg} = \frac{\alpha_{accum}}{t}$$

The transmittance τ_{tail} of the tail composition (how much can be seen through the composition) is used to calculate the tail opacity.

$$\tau_{tail} = (1 - \alpha_{avg})^t, \quad \alpha_{tail} = 1 - \tau_{tail}$$

Finally, the final pixel color (C_{final}) is blended from the core color, the tail color dimmed by the core opacity, and the background color, weighted by the remaining transmittance.

$$C_{final} = C_{core} + (1 - \alpha_{core})\,(C_{tail} + (1 - \alpha_{tail})\,C_{background})$$

The core correctly estimates the visibility and color of its fragments while the tail weights the omitted fragments by their own opacities, approximating their transmittance and color.

4 Experimental Evaluation

We present an experimental analysis of our deep k-buffer method (DKB) when compared against the fixed k-buffer (FKB) [2,20] and the variable k-buffer (VKB) [23] when simulating hybrid transparency [8] with regards to estimated performance (Sec. 4.1) and image quality (Sec. 4.2) under strict memory budgets $M \in \{20MB, 40MB\}$. In every possible memory scenario and resolution, the amount of fragments that can be distributed across the image depends on the size of each fragment node and the fragment allocation strategy of each method. In our implementation, each node stores the fragment color, packed into an unsigned int, (4 bytes) and a depth value (4 bytes). All experiments were conducted on a 1430×960 viewport on an NVIDIA RTX 2080 Super. We have implemented all methods in OpenGL 4.6. The forward pass of the network was implemented using plain OpenGL fragment shaders. We evaluated our method on several high depth complexity scenes (Fig. 6), which were not included in the training data and exhibit higher depth complexity (30-120 fragments) compared to the scenes in the training data.

4.1 Quantitative Results

Performance.We implemented all methods in modern OpenGL[1]. As illustrated in Table 1, our method performs similarly to variable k-buffer, with a performance overhead depending on the inference time of the neural network and the neural network feature extraction pass, offering an interactive performance. Both methods offer the same performance during the *Synchronized k-buffer* stage but differ in the *Feature Extraction* and *Importance Computation* steps. Our method has an additional overhead during the *NN Feature Extraction* stage as it needs to compute and store the neural networks input features, and during the *Deep Pixel Importance* stage (Fig. 5) due to the neural network inference time.

As our method works on a per-pixel basis, its performance is based the number of per-pixel predictions that have to be performed. Thus, our method scales properly in larger or smaller viewport sizes.

Table 1 Performance comparison between our DKB and VKB. Our method with a small performance overhead offers an interactive performance under a fixed memory budget M, in scenes with maximum depth complexity D.

Method	DKB	VKB
Vokselia Spawn, $M = 20MB$, $D = 64$		
Feature Extraction	1.95	0.9
Importance Computation	1.48	0.45
Synchronized k-buffer	18.97	18.97
Total Time (ms)	22.40	20.32
Crytek Sponza, $M = 40MB, D = 30$		
Feature Extraction	1.6	0.45
Importance Computation	3.25	0.39
Synchronized k-buffer	10.38	10.38
Total Time (ms)	15.23	11.22

Memory. The FKB assumes a fixed k value of extracted fragment layers across the entire image. This can result in bad memory utilization, as a large and potentially unused storage space can remain empty in pixels that contain less than k depth complexity. Both DKB and VKB consider the memory fixed, pre-allocated and linearly organized into variable contiguous regions per pixel. In this manner, storage space is exactly allocated and properly utilized due to the ability to pre-allocate storage for a variable, pre-estimated number of fragments per pixel (see Table 2). FKB manages to handle a small percentage of the available fragments while VKB and DKB are able to process almost twice as much fragments.

Table 2 The captured fragment space of the three tested k-buffer approaches in two different scenarios with maximum D depth layers under a fixed storage budget M.

Method	FKB	VKB	DKB
Vokselia Spawn, $M = 20MB$, $D = 64$			
$k(p)$	2	1-9	1-5
Fragments (Perc.)	1.6M (24%)	2.7M (40%)	2.7M (40%)
Crytek Sponza, $M = 40MB$, $D = 30$			
$k(p)$	4	2-25	4-25
Fragments (Perc.)	2.5M (39%)	5.4M (83%)	5.4M (83%)

[1] Shader source code available at: `https://github.com/gtsopus/dhoit`

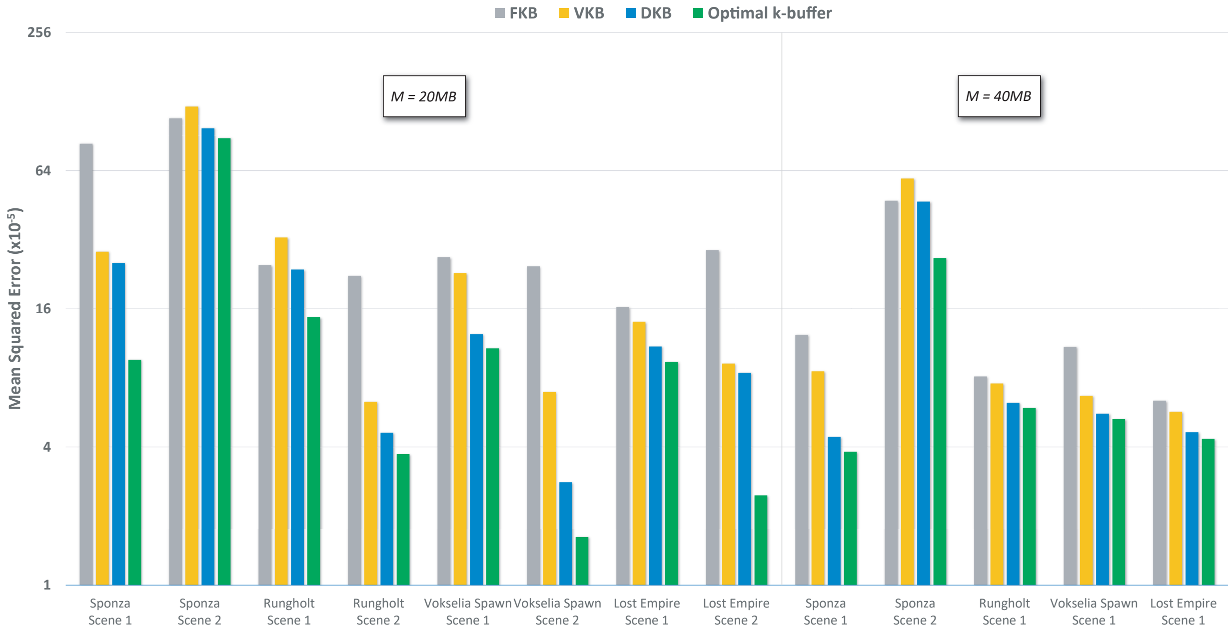

Fig. 6 Mean squared error measurements of the three k-buffer approaches and optimal k-buffer on several scenes with varying memory budgets (log4 scale). Our method outperforms the competing methods, offering a better image quality while still having room to improve (Optimal k-buffer).

4.2 Qualitative Results

Our method provides a better image quality both in terms of mean squared error (Fig. 6), and ꟻLIP mean error [1] (Fig. 7) in comparison to the aforementioned k-buffer approaches, under a fixed memory budget.

FKB produces the lowest quality images due to its k value allocation strategy and potentially wasteful memory allocation (Table 2), which can be noticed especially in scenes with uneven fragment allocation and pixels without any fragments. FBK allocates a fixed, per-pixel memory to store up to k fragments, even if there are pixels with lower depth complexity than the selected k value in the image, which can lead to bad memory utilization. Furthermore, FKB treats all pixels equally by using a fixed, global k value, thus ignoring fragments that may potentially have a higher impact on the final result. In cases where the fixed k value is not large enough to approximate OIT, FKB exhibits visual artifacts due to incorrect blending or missing detail (Fig. 4.2).

The VKB provides a better image quality due to its ability to assign a variable per-pixel value of k to regions deemed more important. The k value allocation is determined using a per-pixel importance value computed according to several importance maps (Fresnel, Periphery, Depth Complexity heuristics). It solves the FKB problem of wasteful memory utilization due to its exact memory allocation strategy and the variable per-pixel k value. VKB provides better visual results compared to FKB but may still incorrectly approximate transparency (especially further away from the center) due to the heuristic rules used for determining pixel importance.

Finally, our DKB outperforms the aforementioned approaches due to its ability to determine a more sophisticated k-value allocation compared to VKB (Fig. 4.2), allowing higher depth complexity in regions that are deemed more important and contribute more to the final result. In a similar way to VKB, our method avoids the memory utilization problems of FKB offering exact memory allocation and variable per-pixel value of k which facilitates the full exploitation of a fixed memory budget.

Our experimental evaluation demonstrates that our method outperforms the competing methods in a plethora of testing scenarios with a variety of scene views and memory budgets in terms of RMSE and ꟻLIP mean error as shown in Fig. 6 and Fig. 7. While our method offers better quality than the competing methods, in many cases, it still lacks in comparison to the *optimal k-buffer* scheme (Sec. 3.1.2), indicating that there is still room for improvement by either improving the neural network architecture, augmenting or enhancing the training dataset or by choosing more or different of

input features, such as neighboring or geometric information.

Finally, we have provided a video as supplementary material that compares our method to the unbounded A-buffer. The video shows that there are no artifacts caused by camera movement. There are occasional differences as compared to the unbounded A-buffer, since due to limited memory resources some pixels take only a few fragments and may differ from neighboring pixels. Such differences from unbounded A-buffer occur in all fixed memory methods, even in the results obtained by the optimal fragment distribution.

5 Conclusions

We have introduced an intelligent approach for approximating the optimal distribution of fragments per pixel under fixed memory budget. The training of the artificial neural network has been conducted with several scenes and views and provides impressive approximation results on all unseen scenes/views that we have considered. Therefore, we have outperformed previous competent methods with the trade off of a small performance overhead which is less than 10% for large scenes. The implementation that we have offered is not optimized and we think that it can be improved to reduce the performance overhead even further.

For some pairs of scenes/views the average MSE is better than previous approaches but still quite large as compared to the optimal k-buffer. This may be alleviated by using features that take into consideration the neighborhood of the pixel or predict the fragment requirements per tile. However, with such approaches one needs to be cautious with performance issues.

The derivation of the optimal per pixel fragment allocation works for any multifragment rendering application and metric, with very few restrictions. Therefore, we can use this approach to train networks for OIT with different metrics [6, 1]. Finally, this methodology may have some merit when applied for fragment allocation prediction for global illumination [18] or for direct rendering of Boolean operations [13].

Acknowledgements

Lost Empire, Vokselia Spawn, Rungholt and Crytek Sponza were downloaded from Morgan McGuire's Computer Graphics Archive [10].

6 Compliance with Ethical Standards

Conflict of interest

All authors declare that they have no conflict of interest.

Funding

This research was supported by project “Dioni: Computing Infrastructure for Big-Data Processing and Analysis” (MIS No. 5047222) co-funded by European Union (ERDF) and Greece through Operational Program “Competitiveness, Entrepreneurship and Innovation”, NSRF 2014-2020.
This work has been co-financed by the European Union (European Regional Development Fund- ERDF) and Greek national funds through the Interreg Greece Albania 2014-2020 Program (project VirtuaLand).

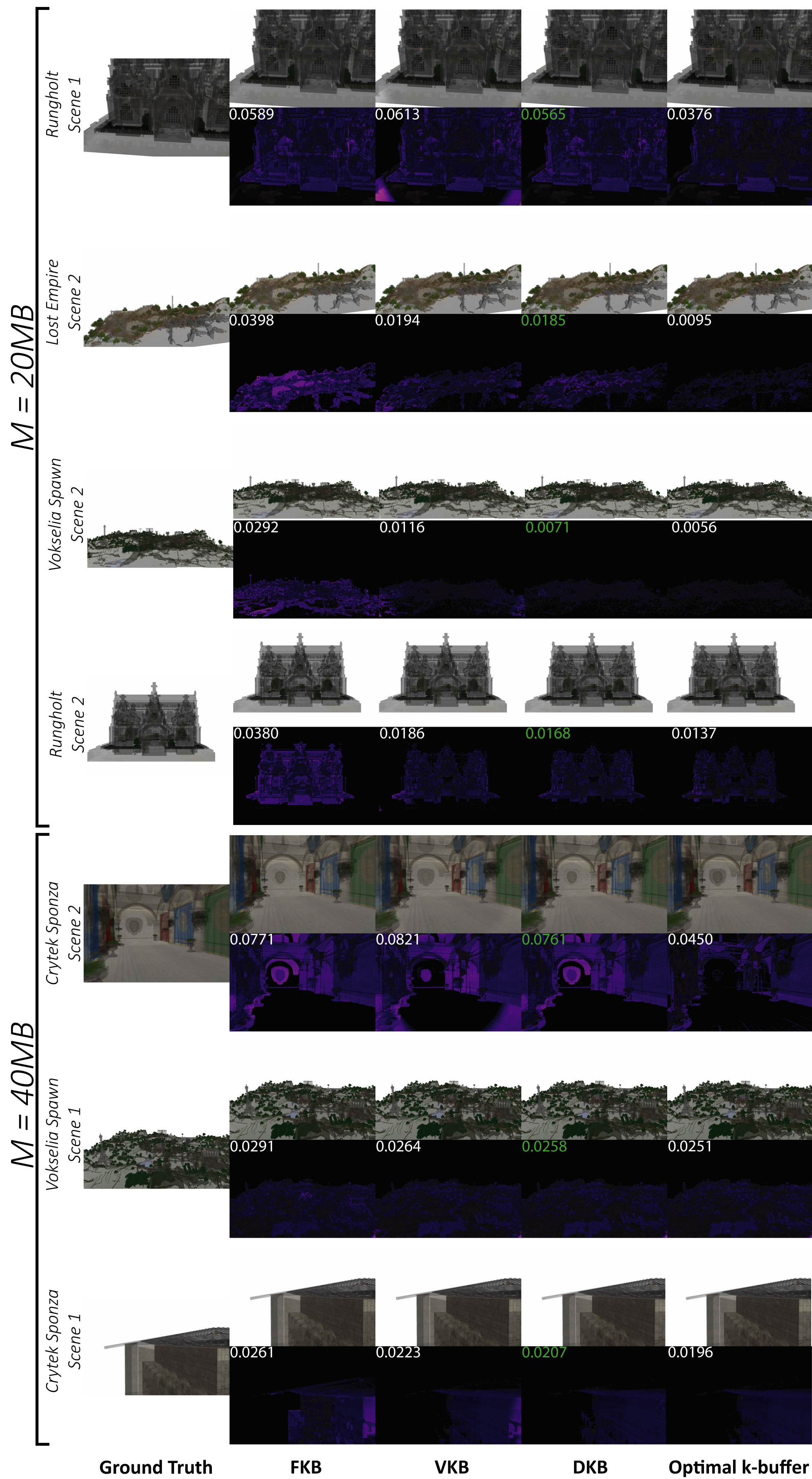

Fig. 7 A quality comparison between the three approaches and the optimal k-buffer. An error map (generated using the ꟻLIP tool [1]) is computed between each method and the ground truth. On the top left corner of each image the ꟻLIP mean error verifies that our method exhibits better results compared to the other two approaches.

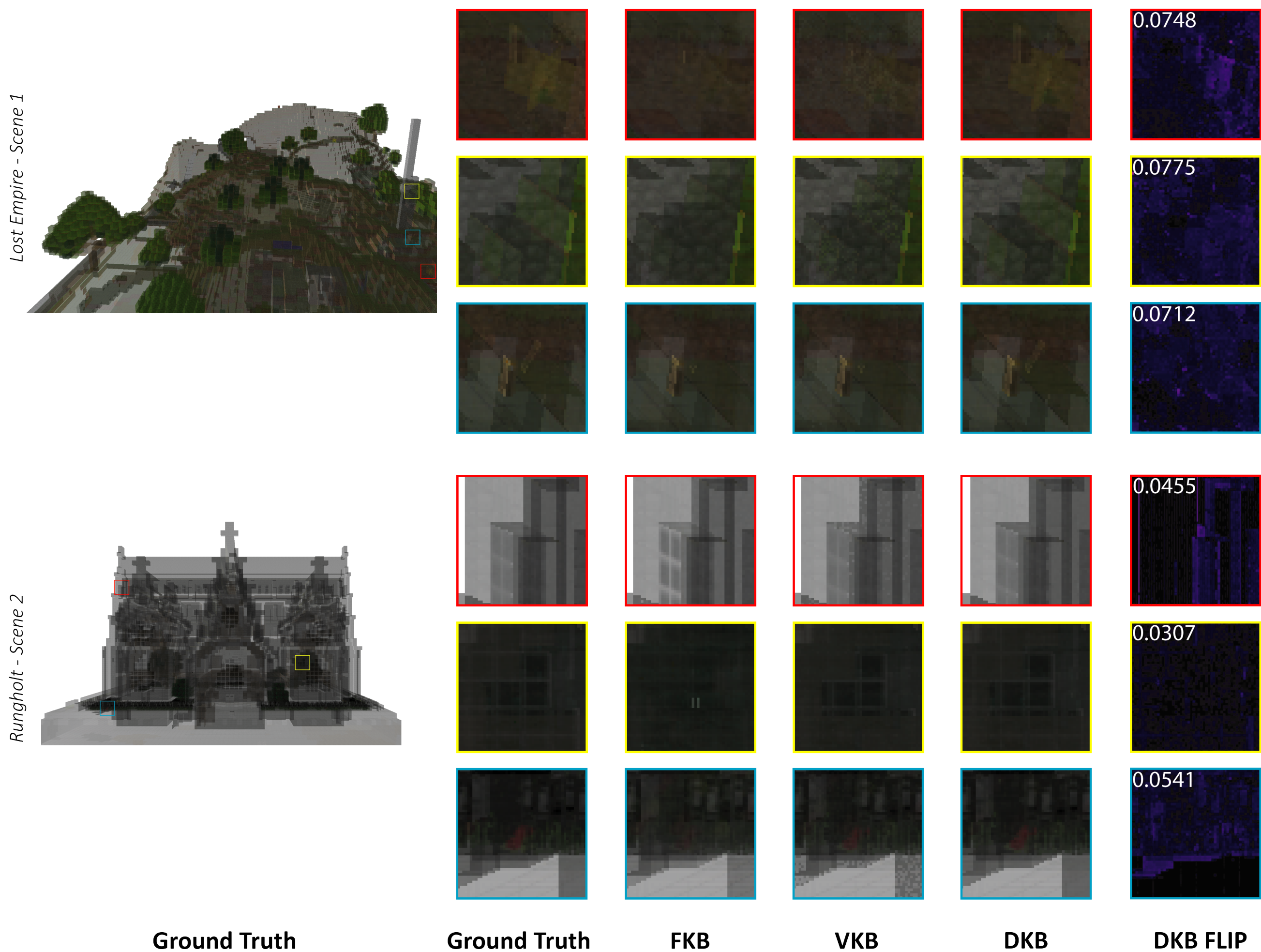

Fig. 8 A quality evaluation between the three k-buffer approaches. Our method achieves better quality results and retains more detail under a strict memory budget of $M = 20MB$ compared to the other two competing methods.

References

1. Andersson, P., Nilsson, J., Akenine-Möller, T., Oskarsson, M., Åström, K., Fairchild, M.D.: *FLIP:* A difference evaluator for alternating images. Proceedings of the ACM on Computer Graphics and Interactive Techniques **3**(2), 15:1–15:23 (2020). DOI 10.1145/3406183
2. Bavoil, L., Callahan, S.P., Lefohn, A., Comba, J.a.L.D., Silva, C.T.: Multi-fragment effects on the gpu using the k-buffer. In: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games, I3D '07, p. 97–104. Association for Computing Machinery, New York, NY, USA (2007). DOI 10.1145/1230100.1230117
3. Bavoil, L., Myers, K.: Order independent transparency with dual depth peeling (2008)
4. Carpenter, L.: The A-buffer, an antialiased hidden surface method. SIGGRAPH Comput. Graph. **18**(3), 103–108 (1984). DOI 10.1145/964965.808585
5. Catmull, E.E.: A subdivision algorithm for computer display of curved surfaces. Ph.D. thesis, The University of Utah (1974)
6. Lavoué, G., Larabi, M.C., Váša, L.: On the efficiency of image metrics for evaluating the visual quality of 3d models. IEEE Transactions on Visualization and Computer Graphics **22**(8), 1987–1999 (2016). DOI 10.1109/TVCG.2015.2480079
7. Liu, F., Huang, M.C., Liu, X.H., Wu, E.H.: Freepipe: A programmable parallel rendering architecture for efficient multi-fragment effects. In: Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '10, p. 75–82. Association for Computing Machinery, New York, NY, USA (2010). DOI 10.1145/1730804.1730817
8. Maule, M., Comba, J.a., Torchelsen, R., Bastos, R.: Hybrid transparency. In: Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D '13, p. 103–118. Association for Computing Machinery, New York, NY, USA (2013). DOI 10.1145/2448196.2448212
9. Maule, M., Comba, J.L., Torchelsen, R.P., Bastos, R.: A survey of raster-based transparency techniques. Computers & Graphics **35**(6), 1023–1034 (2011). DOI 10.1016/j.cag.2011.07.006
10. McGuire, M.: Computer graphics archive (2017). `https://casual-effects.com/data`
11. Nalbach, O., Arabadzhiyska, E., Mehta, D., Seidel, H.P., Ritschel, T.: Deep shading: Convolutional neural net-

works for screen space shading. Computer Graphics Forum **36**(4), 65–78 (2017). DOI 10.1111/cgf.13225
12. Porter, T., Duff, T.: Compositing digital images. In: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH ’84, p. 253–259. Association for Computing Machinery, New York, NY, USA (1984). DOI 10.1145/800031.808606
13. Rossignac, J., Fudos, I., Vasilakis, A.: Direct rendering of boolean combinations of self-trimmed surfaces. Computer-Aided Design **45**(2), 288–300 (2013). DOI 10.1016/j.cad.2012.10.012
14. Salvi, M., Montgomery, J., Lefohn, A.: Adaptive Transparency. In: Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics. ACM (2011). DOI 10.1145/2018323.2018342
15. Salvi, M., Vaidyanathan, K.: Multi-layer alpha blending. In: Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D ’14, p. 151–158. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2556700.2556705
16. Tewari, A., Fried, O., Thies, J., Sitzmann, V., Lombardi, S., Sunkavalli, K., Martin-Brualla, R., Simon, T., Saragih, J., Nießner, M., Pandey, R., Fanello, S., Wetzstein, G., Zhu, J.Y., Theobalt, C., Agrawala, M., Shechtman, E., Goldman, D.B., Zollhöfer, M.: State of the art on neural rendering. Computer Graphics Forum **39**(2), 701–727 (2020). DOI 10.1111/cgf.14022
17. Thomas, M.M., Forbes, A.G.: Deep illumination: Approximating dynamic global illumination with generative adversarial network. CoRR **abs/1710.09834** (2017). DOI 10.48550/arXiv.1710.09834
18. Vardis, K., Vasilakis, A.A., Papaioannou, G.: A multiview and multilayer approach for interactive ray tracing. In: Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D ’16, p. 171–178. Association for Computing Machinery, New York, NY, USA (2016). DOI 10.1145/2856400.2856401
19. Vasilakis, A., Fudos, I.: S-buffer: Sparsity-aware multifragment rendering. In: Eurographics (Short Papers), pp. 101–104. The Eurographics Association, Cagliari, Sardinia (2012). DOI 10.2312/conf/EG2012/short/101-104
20. Vasilakis, A.A., Fudos, I.: k^+-buffer: Fragment synchronized k-buffer. In: Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, I3D ’14, p. 143–150. Association for Computing Machinery, New York, NY, USA (2014). DOI 10.1145/2556700.2556702
21. Vasilakis, A.A., Papaioannou, G., Fudos, I.: k^+-buffer: An efficient, memory-friendly and dynamic k-buffer Framework. IEEE Transactions on Visualization and Computer Graphics **21**(6), 688–700 (2015). DOI 10.1109/TVCG.2015.2417581
22. Vasilakis, A.A., Vardis, K., Papaioannou, G.: A survey of multifragment rendering. Computer Graphics Forum **39**(2), 623–642 (2020). DOI 10.1111/cgf.14019
23. Vasilakis, A.A., Vardis, K., Papaioannou, G., Moustakas, K.: Variable k-buffer using importance maps. In: Proceedings of the European Association for Computer Graphics: Short Papers, EG ’17, p. 21–24. Eurographics Association, Goslar, DEU (2017). DOI 10.2312/egsh.20171005
24. Wyman, C.: Exploring and expanding the continuum of oit algorithms. In: Proceedings of High Performance Graphics, HPG ’16, p. 1–11. Eurographics Association, Goslar, DEU (2016). DOI 10.2312/hpg.20161187
25. Yang, J.C., Hensley, J., Grün, H., Thibieroz, N.: Realtime concurrent linked list construction on the gpu. In: Proceedings of the 21st Eurographics Conference on Rendering, EGSR’10, p. 1297–1304. Eurographics Association, Goslar, DEU (2010). DOI 10.1111/j.1467-8659.2010.01725.x
26. Zhang, D., Xian, C., Luo, G., Xiong, Y., Han, C.: Deepao: Efficient screen space ambient occlusion generation via deep network. IEEE Access **8**, 64,434–64,441 (2020). DOI 10.1109/ACCESS.2020.2984771