Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on CEIG 2021

Feature-based clustered geometry for interpolated Ray-casting

Francisco González García[1], Ignacio Martin, Gustavo Patow[*]

ViRVIG - Universitat de Girona, Spain

A R T I C L E I N F O

Article history:
Received 14 July 2021
Received in revised form 29 August 2021
Accepted 30 August 2021
Available online 3 September 2021

Keywords:
Accelerated Rendering
Geometry clustering
Ray Casting

A B S T R A C T

Acceleration techniques for Rendering in general, and Ray-Casting in particular, have been the subject of much research in Computer Graphics. Most efforts have been focused on new data structures for efficient ray/scene traversal and intersection. In this paper, we propose an acceleration technique that approximates rendering and that is built around a new feature-based clustering approach. The technique starts preprocessing the scene by grouping elements according to their features using a set of channels based on an information theory-based approach. Then, at run-time, a rendering strategy uses that clustering information to reconstruct the final image, by deciding which areas could take advantage of the coherence in the features and thus, could be interpolated; and which areas require more involved calculations. This process starts with a low-resolution render that is iteratively refined up to the desired resolution by reusing previously computed pixels. Our experimental results show a significant speedup of an order of magnitude, depending on the complexity of the per-pixel calculations, the screen size of the objects, and the number of clusters. Rendering quality and speed directly depend on the number of clusters and the number of steps performed during the reconstruction procedure, and both can easily be set by the user. Our findings show that feature-based clustering can significantly impact rendering speed if samples are chosen to enable interpolation of smooth regions. Our technique, thus, accelerates a range of popular and costly techniques, ranging from texture mapping up to complex ambient occlusion, soft and hard shadow calculations, and it can even be used in conjunction with more traditional acceleration methods.

1. Introduction

The problem of efficient image generation has been a cornerstone of research since the earliest days in Computer Graphics [1]. Ray Tracing is one of the most popular techniques when generality, quality and ease of implementation comes into account, being able to handle most optical effects [2]. Thus, it is logical that most efforts have been devoted to increase the speed of these calculations [3–6].

However, besides tracing the rays themselves, complex shading operations (e.g., complex brdfs, sub-surface scattering, etc.) can be expensive to compute, considerably hindering rendering performance. As a consequence, sometimes rough approximations, simplified calculations or other trade-offs are used to accelerate computations [7,8]. A promising avenue for optimization is to exploit different kinds of coherence inherent to the rendered scenes [9–11], but the complexity of these CPU-oriented approaches has precluded their use in modern hardware-based ray tracing approaches.

In this paper our aim is to investigate ways to exploit the coherence, when projected into screen space, of the low-variability regions in the scene. First, a pre-processing stage uses information theory-based tools to define a channel for each feature, allowing to cluster the scene geometry according to them (e.g. visibility, orientation, or texturing). Then, the runtime step approximately computes a series of passes with increasing resolution images up to the final resolution. At each pass, samples of the same cluster (similar samples) are reused in order to obtain, wherever possible, an interpolated value for the new samples. If not possible, for instance at cluster boundaries, the samples are evaluated in the current pass and reused in future passes. One of the main benefits of the proposed technique is its ability to increase rendering performance by reusing previous calculations, whenever possible, done in lower resolution passes to generate the final image. We also introduce an automatic control of the number of passes such that performance has a lower bound in the performance of traditional Ray Casting, resulting in a win-win situation. From the point of view of quality, the user is able to have a fine-grained control of the final rendering by selecting the channels involved in the clustering stage and controlling their thresholds. Finally, memory usage requirements are of the same level as traditional Ray Casting techniques. This paper presents a novel technique that provides a number of contributions:

* Corresponding author.
E-mail addresses: gonzalezgarciafran@gmail.com (F.G. García), ignacio.martin@udg.edu (I. Martin), gustavo.patow@udg.edu (G. Patow).

1 Currently at Pixar Animation Studios.

https://doi.org/10.1016/j.cag.2021.08.019

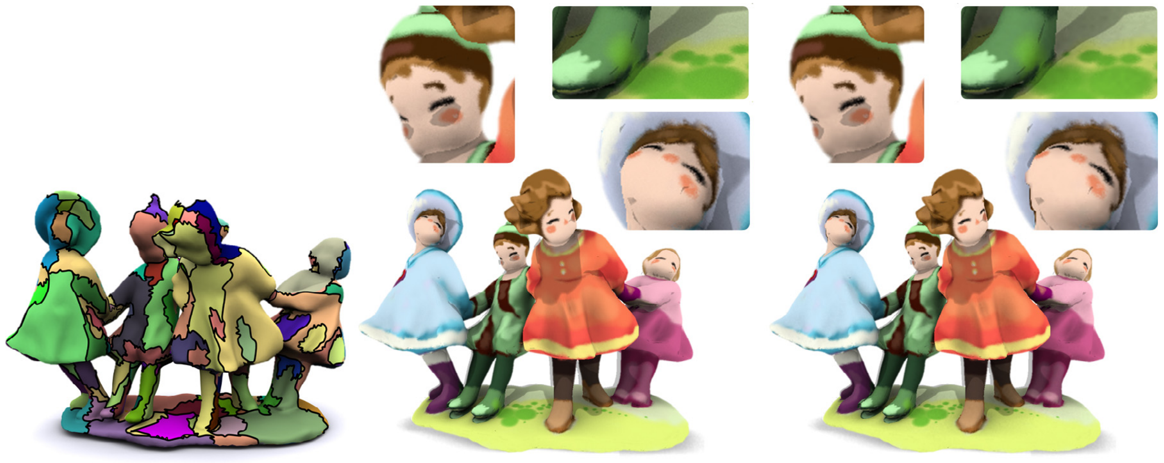

Fig. 1. Our system first performs a feature-based clustering of the object (left) and then reconstructs the final image by an approximate interpolated approach (right). Quality is comparable with the same model rendered by using Ray-casting (middle), observe the texture details, shadow boundaries and correct visibility. Our technique can render the scene up to 12 times faster than Ray-casting.

- A mesh-clustering framework based on tools from Information Theory to exploit feature similarity. In particular, we define clustering criteria based on geometry visibility, orientation, texture stretching and other user-definable parameters, decoupling them from the actual rendering process.
- A simple to implement, multi-pass progressive rendering strategy based on the reuse and interpolation of previously-computed results to reconstruct the final image.
- A controlling mechanism to guarantee traditional Ray-casting as a lower bound to the rendering speed.
- Our technique can accommodate both static and animated scenes as well.

As limitations of our technique, the pre-processing requires knowing in advance the properties of the scene to be rendered, and the interpolating nature of the algorithm prevents extremely high-frequency objects, such as the hairball, to take advantage of this technique, forcing the system to resort to traditional pixel-by-pixel rendering.

2. Previous work

The technique we present in this paper is related to different areas in the Computer Graphics literature. Here we are going to review only the most relevant related works, providing pointers for further reading when possible. For a general discussion of the developments in Computer Graphics about continuity and interpolation techniques, please refer to the survey by Gonzalez and Patow [12].

Acceleration structures for ray tracing. State of the art Ray Tracing methods rely on acceleration structures. We can classify these techniques as inter- and intra-frame techniques. Inter-frame acceleration techniques try to use information from previous frames for the current one. Data reprojection [13,14] exploits the temporal coherence by caching the expensive intermediate shading calculations performed at each frame. However, these methods are unable to avoid unexpected performance drops in fast movements or drastic view changes, because of the cache misses and shading recomputations in the new view. On the other hand, intra-frame techniques usually rely on acceleration structures to speed up the traversal of primary and secondary rays. While this problem has been well studied for CPUs [3], only a few approaches provided GPU-efficient dynamic ray-traversal acceleration data structures [4–6]. These object-space data structures are seamlessly compatible with our scheme.

Image-space techniques. These techniques can separate algorithmic and scene complexities, avoiding wasted computations on off-screen portions of the scene. Szirmay-Kalos et al. [11] introduced an image-based structure to avoid complex ray-traversal evaluations, by looking up an approximated result from an environment map. More recently, Novák et al. [8] suggested to accelerate Ray Tracing by converting complex meshes into a set of rasterized height fields intersected by simple ray marching, relying on replacing many ray traversals of complex dense geometries. On the contrary, our acceleration strategy proves effective for all kinds of scenes. Yang et al. [15] proposed accelerating rendering by using a subsampled image and using an edge-preserving upsampling approach to obtain the final resolution, requiring to process the scene twice. For Ray Tracing this requirement would decrease the possible gain when upsampling the shading evaluations. Our approach reuses samples, effectively computing only the ones in the final image, thus processing the scene once.

Sample re-utilization. Our approach belongs to the family of methods that reuse and interpolate samples at homogeneous regions. Akimoto et al. [9] proposed a method that exploits the similarity between adjacent pixels to reduce the number of evaluations. Although they use a similar sampling pattern to ours, their procedure requires several verifications at run-time, which introduces a considerable overhead compared to our simple cluster ID verifications. Moreover, they use pixel intensities to determine similarity between samples, resulting in texture interpolation problems. Instead, our technique smoothly interpolates texture coordinates allowing high frequency texture details. Bala et al. [10] proposed a CPU-based system that uses per-surface interpolants to approximate and accelerate radiance computations. They decouple the acceleration of visibility and shading operations by exploiting temporal coherence and interpolating radiance samples. A hierarchical data structure called *linetrees* is used at run-time, being its maintenance one of the main drawbacks of this technique, as it is complex and costly. In contrast, our technique can be easily adapted by both CPU and GPU ray-tracers, without needing such structure, as our interpolation decision is extremely fast and simple (a simple cluster ID check). Adamson et al. [16] presented an acceleration method for intersectable models exploiting spatial coherence by adjusting the sampling resolution, but it lacks support for popular techniques such as shadows and ambient occlusion, which involve the computation of secondary rays. Moreover, as they shift the ray-scene intersections computations

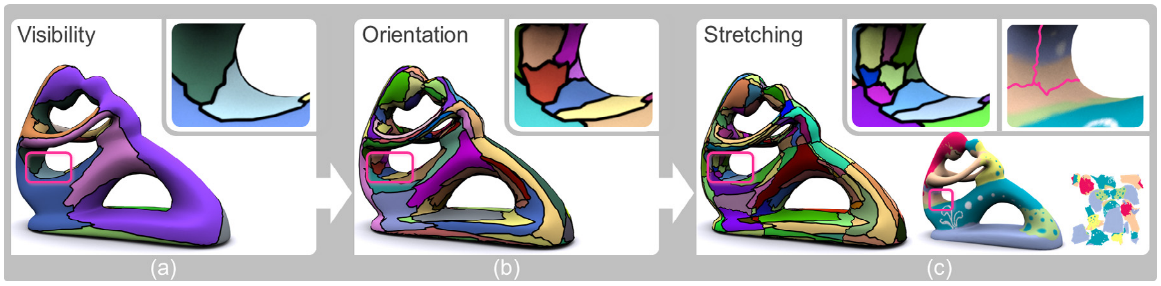

Fig. 2. Iterative clustering: first, visibility, then, orientation, and finally, texture stretching channels. The pink lines represent the boundaries of the multi-chart parameterization used.

to the CPU, GPU-based ray-tracers performance may drop due to data transfer operations. A more recent approach by Bitterli et al. [17] reuses samples from neighboring pixels and/or frames to improve quality for real-time ray-tracing with dynamic direct lighting. Crespo et al. [18] generate a data structure in which global illumination is piecewise approximated with polynomials, which, ultimately, in screen space becomes interpolation, which is related to the technique presented in this paper.

Recently, Bako et al. [19] took a deep learning approach for Monte Carlo-rendered images using a denoising technique that produces high-quality results suitable for film production, by training a convolutional neural network to learn the complex relationship between noisy and reference data across a large set of frames with varying distributed effects from movies. Chaitanya et al. [20] introduced a variant of deep convolutional networks better suited to the kind of noise present in Monte Carlo rendering, allowing for much larger pixel neighborhoods to be taken into account, and at the same time improving execution speed by an order of magnitude. Vogels et al. [21,22] introduced a modular architecture based on kernel-predicting neural networks that performs multi-scale, temporal denoising of rendered sequences. Gharbi et al. [23] presented a splatting approach that works with the samples directly, and is trained using deep learning. This allows to handle in an appropriate way various components of the illumination, such as specular reflection, indirect lighting, motion blur, and depth of field more effectively. We recommend the interested reader to refer to the survey by Zwicker et al. [24] for an in depth review of the advances up to 2015 in adaptive sampling and the associated reconstruction techniques for Monte Carlo-based rendering. Although our technique can be considered to be somewhat related with denoising techniques, its purpose is completely different: here we do not want to remove noisy artifacts produced by stochastic sampling the scene, but to optimize the number of samples needed to generate a quality image, interpolating the rest.

Mesh clustering. Mesh clustering techniques, also known as mesh segmentation, group triangles by their similarity and are characterized by both, their objective (surface-type or part-type segmentation) and approach (region growing, hierarchical clustering, iterative clustering, etc.) used to segment the mesh. One important aspect in all mesh segmentations methods is the similarity criteria used to group triangles, which is usually based on a set of attributes (curvature, geodesic distances, parameterization distortion, etc.) obtained from the mesh [25–31]. Here, our interest is on clustering but from the homogeneity while rendering point of view. Thus, although our work might be related to other mesh clustering approaches, the techniques and metrics involved are completely different.

Information theory. The concepts of information theory have been previously applied in many areas of Computer Graphics, introducing measures and relationships with important properties for different scenarios such as radiosity, shape descriptors, and viewpoint selection [32,33]. To the best of our knowledge, our approach is the first to combine information theoretic measures to define a cluster-based acceleration data structure suited for Ray Tracing. Here we have chosen this approach for its large success in many similarity-detection problems, such as viewpoint selection [32], selection of automatic transfer functions [33], facade element recognition [34] and mesh saliency [35]. For a more detailed information on mesh segmentation techniques and information-theoretic tools in Computer Graphics, we refer the interested reader to the excellent survey of Shamir [36] or the book from Sbert et al. [37].

3. Clustering

At the heart of our technique lies a pre-processing stage, which has the objective of grouping the input geometry according to a user-defined set of features. Its input consists of a triangle soup with connectivity information (i.e., we do not require any special structuring). For each triangle we may have any number of associated attributes (e.g., color, normal, etc.).

We first define a number of information theoretic channels, each one representing a user-defined feature. See Section 3.1. For each channel, the algorithm uses a clustering strategy to group triangles into homogeneous clusters of similar elements. Observe that this "*homogeneity*" is only with respect to a given feature/channel, and that the resulting triangle groups might not be homogeneous with respect to a different feature. See Section 3.2.

Then, the algorithm performs an iterative hierarchical processing over the mesh, progressively obtaining finer clusters after each step: the clusters generated at the previous step with respect to a channel/feature are fed as independent meshes to the clustering algorithm for the next channel/feature (See Fig. 2). See Algorithm 1. It is easy to realize that, after each iteration, the input geometry is partitioned into a finer set of clusters, and each cluster is, at most, as large as the input. Each iteration operates over smaller sets of triangles, resulting in a reduced cost and faster pre-processing computations. Finally, as the resulting clusters present jaggy edges, we apply a boundary-smoothing algorithm respecting the criteria, improving rendering at shorter distances. See Section 3.3.

3.1. Information theoretic channels

As mentioned above, our clustering strategy consists of detecting homogeneous areas with respect to some user-defined criteria. In order to evaluate the similarity among triangles, we use information theoretic tools. Let X be a discrete random variable with probability distribution $\{p(x)\}$, where $p(x) = Pr\{X = x\}$ (also denoted by $p(X)$) and $x \in X$. Let Y be another random variable with its respective probability distributions. Let $p(y|x) =$

Algorithm 1: Clustering algorithm.

```
Input: $Scene\_mesh$
$clusters := [mesh]$
for each channel ch in channels do do
    $newClusters := []$
    for each cluster cl in clusters do
        ch.init(cl)
        ch.cluster()
        ch.smoothBoundaries()
        $newClusters += $ ch.getClusters()
    end
    $clusters := newClusters$
end
return clusters
```

$Pr[Y = y|X = x]$ be the conditional probability of y given x. The conditional entropy can be thought of in terms of an information channel $X \to Y$ [37]. The basic elements of this channel are:

- An input distribution $p(X)$, which represents the probability of selecting each element in X, which also can be considered as the importance given to each x.
- Transition probability matrix $p(Y|X)$. Conditional probabilities fulfill $\sum_{y \in Y} p(y|x) = 1$, representing the probability of an element from Y once a given one from X has happened.
- The output distribution $p(Y)$, given by

$$p(y) = \sum_{x \in X} p(x)p(y|x) \quad (1)$$

represents the average probability of each element in Y.

In some cases, it could be more practical to compute $Y \to X$ and then invert the channel to get $X \to Y$. This can be easily done with the help of Bayes' theorem: $p(x, y) = p(x)p(x|y) = p(y)p(y|x)$. We classify channels that require an inversion as *indirect channels*, while the ones that do not as *direct channels*. The mutual information $I(X; Y)$ between two random variables X and Y is defined by $I(X; Y) = H(X)-H(X|Y) = H(Y)-H(Y|X)$, being H and $H(X|Y)$ the entropy and conditional entropy, respectively.

Finally, the Jensen–Shannon (JS) divergence is defined as:

$$JS(\pi_1, \ldots, \pi_n; p_1, \ldots, p_n) = H\left(\sum_{i=1}^{n} \pi_i p_i\right) - \sum_{i=1}^{n} \pi_i H(p_i)$$

with p_i as the probability distributions defined with normalized weights π_i, $i \in [1, n]$. We use this divergence as a measure of the similarity between the triangles to cluster in a specific information theoretic channel. For practical reasons, we introduce the following shorthand notation for the case of two probability distributions a and b and an output variable Y:

$$JS(Y|a, b) = JS(\pi_1 = p(a), \pi_2 = p(b); p(Y|a), p(Y|b))$$

Once the information channel is computed, we cluster the mesh according to the Jensen–Shannon divergence, placing the triangles with a similarity below a user-defined threshold Th in the same cluster. See following sections.

Throughout this paper, we have chosen three different features to be used to extract similarity information to speed up the rendering performance: visibility, orientation and texture stretching. We believe that those features are important, but at the same time, they can be considered as an example on how the proposed approach could very well fit other user-defined properties.

In the case of the orientation and stretching channels (*direct channels*), X will represent the triangles of a model. Otherwise, in the case of the visibility channel (*indirect channel*), X will represent a set of viewpoints.

3.1.1. Visibility channel

For visibility, we define an information channel between the set of input viewpoints V and the set of output mesh polygons T, as $V \to T$. Then, we invert it ($T \to V$) as it is simpler to compute the visibility from every viewpoint using rasterization than doing so from each triangle. This example of an *indirect* channel is called the viewpoint channel and was introduced in [32]. Conceptually, the channel expresses, for any given triangle t_j, how all viewpoints v_i see it. The probability for each viewpoint is $p(v_i) = 1/N$, with N the number of viewpoints, all having the same "preference". We define the *occlusion ratio* as $OR(t_j, v_i) = a_{v_i}(t_j)/u_{v_i}(t_j)$, where $a_{v_i}(t_j)$ is the projected area (in pixels) of triangle t_j at viewpoint v_i, and $u_{v_i}(t_j)$ is the same projected area without taking into account occlusions. Observe that $OR(t_j, v_i) <= 1$, with the equality meaning that t_j is fully visible from all v_i, and 0 that it is fully occluded. We define the normalized transition probability matrix $p(T|V)$ as $p(t_j|v_i) = OR(t_j, v_i)/\sum_t OR(t, v_i)$. The output distribution $p(t_j)$, is computed as $p(t_j) = \sum_{v_i \in V} p(v_i)p(t_j|v_i)$, which represents the average occluded area of t_j. This definition differs from previous ones in that the probability is independent of the model triangulation. To the best of our knowledge, this is the first time this is proposed. Also, note that we do not consider orientation in this channel as we do it on a separate channel. This allows us to have a finer degree of control over the clustering process. To accelerate computations, we have implemented this channel entirely on the GPU, being an order of magnitude faster than previous approaches [32].

3.1.2. Orientation channel

This *direct channel* accounts for strong differences in the polygon dihedral angles (orientations), like sharp edges or strong curvatures. It goes from the 3D triangles $t \in T$ to the triangles themselves ($T \to T$). Here the input probabilities $p(t)$ are set to the constant value $1/M$, with M the total number of triangles to be processed, $p(t_i|t_j) = (1-n(t_i)\cdot n(t_j))/\sum_t(1-n(t_i)\cdot n(t))$, with $n(t)$ the normal of triangle t, and we$p(t_j)$ is computed using Eq. (1).

3.1.3. Texture stretching channel

This *direct channel* is designed to account for stretching in the textures applied to objects, which should be preserved if we are going to do an interpolated up-sampling process of the texture coordinates (see Fig. 12). Conceptually, given a triangle t_i in 3D space, it focuses on its stretching with respect to all other triangles t_j in T. It is a channel $T \to T$. Again, we set $p(i_j) = 1/M$, $p(t_j|t_i) = (1-|S(t_i)-S(t_j)|)/\sum_t(1-|S(t_i)-S(t)|)$ and we compute $p(t_j)$ using Eq. (1), and normalized. Here, $S(t)$ is the stretching of a given triangle t calculated as described by Degener et al. [38].

3.1.4. Other possible channels

Although we have not implemented them, it is easy to think of other clustering criteria. Our previous channels already took into account visibility, orientation and texture stretching effects. Complementary channels could be reflections, refractions and other optical effects. However, we do not expect these channels to introduce large changes with respect to our current implementation. For instance, soft shadows or ambient occlusion are somewhat already included in the visibility channel, as the quality of our results shows. For instance, Fig. 1 contains both, relying only on the above defined channels.

3.2. Clustering a single channel

In general, each cluster represents an homogeneous and continuous area in the model, and a cluster boundary reflects an abrupt change in any relevant feature (e.g., visibility). For any channel, the clustering process follows four basic steps. First, the seeds of the initial clusters are selected. Second, a parallel clustering approach grows those clusters trying to make them as large as possible. Third, a sequential stage assigns possible unclustered triangles to one of the existing clusters and finally, a merging step group small clusters to avoid over clusterization (see Section 3.2.4).

3.2.1. Clustering initialization

Initializing a cluster means computing all the elements of the information channel, as previously described. This implies computing the probabilities $p(x)$, $p(y)$ and $p(Y|X)$. Also, some channel evaluations can be the result of a possible noisy evaluation, like visibility: here, measuring means rendering the triangles with OpenGL, which provides a good but not exact estimate. To prevent future problems, we *denoise* it by replacing the value for each probability $p(Y|t_i)$ by the average of the probability over the immediate similar triangle neighbors, i.e.,

$$p(Y|t_i) \leftarrow \|SN(t_i)\|^{-1} \sum_{j \in SN(t_i)} p(Y|t_j)$$

Where the neighbors $SN(t_i)$ are those that are similar enough:

$$JS(Y|t_i, t_j) \leq Th$$

where Th is a threshold specifically set for this channel.

3.2.2. Selecting seeds

To select the seeds of our clusters, we start by adding the most representative triangle, that is the triangle with the minimum $I(x, y)$ value. Then, while we are not set and there are candidate triangles, we select the triangle M with the maximum difference with respect to all other seeds,

$$M = max_{t \in T} \sum_{s \in seeds} JS(Y|t, s)$$

such that, for every already selected seed s, M verifies that it is different enough from s, i.e., $JS(Y|M, s) > Th$, adding M to the set. When there is no such M, we stop the selection process.

Actually, for large meshes, this can be a bottleneck: we need to compare all the triangles between themselves. We decided to do a GPU-based implementation that distributes the calculation of the needed JS divergences of a given seed with respect to all the other triangles in parallel. Observe this is done lazily, only for the requested seeds.

3.2.3. Parallel and sequential clustering

We implemented two clustering strategies that we apply sequentially, both based on the same basic ideas. First, we grow the clusters from the seeds in parallel. A triangle can be in a given cluster if its Jensen–Shannon divergence with respect to the seed is smaller than the user-provided threshold Th. The seeds grow in a greedy way, trying to grab one ring of triangles at each iteration. When finished, there might be small groups of triangles that were not added to any cluster, mainly because another cluster "barred" the "right" one to grow in their direction. Then, the sequential clustering selects a new seed, growing the cluster until no more triangles can be added. This process is iterated until no unclustered triangles remain. This may result in some small clusters, which may slow down the runtime rendering stage, so we correct this in the following step. Fig. 3 shows the effect of selecting different thresholds, which results in somewhat different clusters being built: first of all, a tight threshold produces a set of seeds that is a subset of the set produced by a more relaxed one. Second, because the clustering process itself will stop growing sooner if the threshold is tighter.

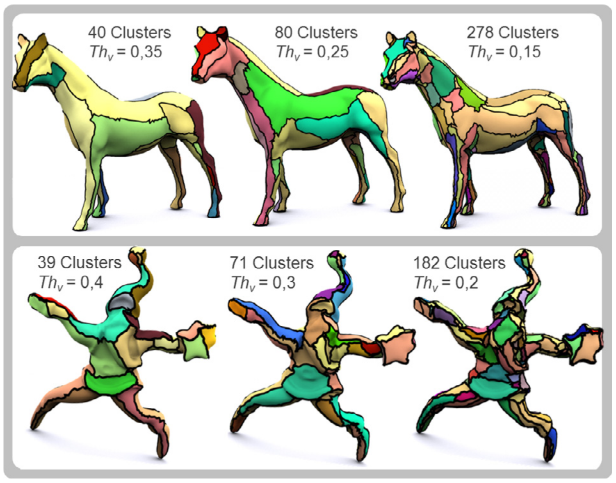

Fig. 3. Results on the clustering when varying the threshold *Th*. In this case the Visibility Channel was used to cluster the triangles.

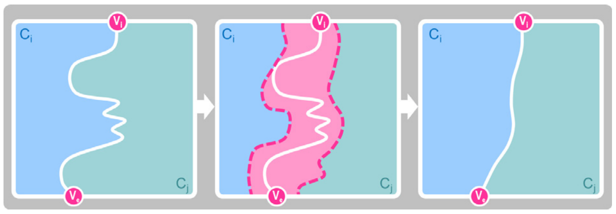

Fig. 4. Around the boundary between two clusters, C_i and C_j, we define a band from vertex V_i to vertex V_e., which will be used to redistribute triangles to get a smoother boundary to improve interpolation possibilities. See Fig. 5.

3.2.4. Merging clusters

As mentioned, sometimes small clusters can be created. To avoid over clustering the model, we merge those small clusters with other neighboring clusters. A cluster is selected for merging if it has a number of triangles smaller than a given fraction of the input triangles. The cluster chosen to merge the small cluster with is the one whose seed has the smallest JS divergence with the small cluster seed. That is, we merge small clusters with the most similar neighboring cluster.

3.3. Smoothing

The resulting clusters can have rough boundaries, which can result in a poor interpolation at certain views. See Fig. 5, left column. Observe that, in the boundary regions, there are triangles that could have been clustered with the seeds of both clusters sharing the boundary, and that were assigned to one given cluster just because the growing process made this cluster to "arrive first". We call the set of all these triangles the boundary *band*. Actually, defining the band this way can produce too broad bands, so we restricted the definition to those triangles that were clustered with a seed but that are more similar (in the Jensen–Shannon sense) to the other seed. See Fig. 4. The actual boundary between two given neighbor clusters actually can be any line inside (and along) the band. To compute this smooth new boundary, we follow a procedure inspired in the work by Sander et al. [25].

First, we generate the actual band, initialized with all the edges of the current boundary. Then, we identify all triangles that could have been more likely clustered with the seed at the other side of the boundary. Once identified, we add all their edges whose incident triangles are completely inside one of the clusters

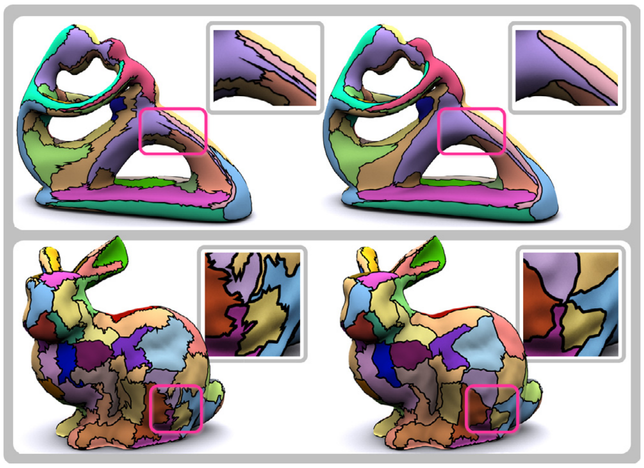

Fig. 5. Clusters before and after smoothing. See Fig. 4.

we are considering. This last condition was introduced to avoid problems by accidentally modifying other clusters.

Once all edges have been selected, we simply compute the shortest path through the band from the starting boundary vertex to the end vertex. In our implementation, we use the well known A^* algorithm. Once the new boundary has been computed, we simply reallocate the triangles to their new clusters. As a positive side-effect of this reordering, the resulting clusters have, in general, better shapes. See Fig. 5.

3.4. Animated scenes

We account for animated scenes by repeating the clustering for each keyframe of the model animations. Basically, an animation consists of a series of keyframes which are interpolated to get the final postures of the characters for in between frames. We use these keyframes to iteratively perform the clustering for each one. Starting from the initial keyframe, we repeat the clustering pipeline for each successive keyframe, every iteration working after the result of the previous one. This way, we ensure having a consistent model which can be reused for every frame of the animation. Observe that our current implementation only takes the keyframes, not any interpolated position for the computations, which might result in some situations where visibility problems might occur. In such cases, simply taking further frames in between for the calculations would suffice to solve this problem. In Fig. 6 we can see the result applied to the Panda model.

3.5. Threshold selection

Although the only parameters that the user is expected to select are the channel thresholds, selecting a good threshold can be challenging for an inexperienced user. In addition, the selection of a given threshold can also produce different results that could positively or negatively impact the rendering, later on. In Fig. 3 we can see the effect of selecting different thresholds, which results in somewhat different clusters being built: first of all, a tight threshold produces a set of seeds that is a subset of the set produced by a more relaxed threshold. Second, the clustering process itself will stop growing sooner if the threshold is tighter.

To provide an intuitive interface, we use the upper and lower bounds to the JS divergence, which can be demonstrated to be $JS \in [0, 1]$ [39]. This way, the user can have a simple slider to control the clustering by setting any value between $Th = 0$, resulting in a cluster for each triangle; and $Th = 1$, resulting in a single cluster for the whole model. This way, threshold selection becomes a simple and easily configurable task.

4. Rendering by upsampling

Here we focus on the rendering stage, which consists of a multi-pass approximate strategy that progressively computes the final image from the information gathered at previous passes. Based on this information, the runtime shaders decide whether to reuse the previously computed values and interpolate a new value from them, or simply to perform a full computation to guarantee the final image quality. For each sample, the information we need to store is its position, texture coordinates, cluster ID, normal, and some extra information depending on the chosen channels. In our case, we add the ratio of occluded samples for soft shadowing and ambient occlusion. All this information is stored in a buffer we call the *I-Buffer* (*I* from interpolation).

The first pass has no previous reference image, so all samples must be fully evaluated, as shown in Fig. 7-left. Then, from our sampling pattern, the coincident samples at the new resolution can be directly copied, as shown in Fig. 7-middle with the samples marked as "R" (for *Re*-used). On the other hand, if the cluster IDs of the previous samples are equal, new samples in-between can be interpolated (marked with "I" in the figure). If they are different, they need a full evaluation (marked with "E"). The arrows in the image show which previous samples are used for each new sample. Fig. 7-right shows the third pass in the process.

The key to our up-sampling algorithm is that the samples evaluated at each pass are the same, and thus could be reused at following passes. This, together with the reconstructing pattern we use, means that each pass should have a size of $(2N-1) \times (2M-1)$ if the previous one was $N \times M$. However, we can render any resolution by just starting from correctly chosen N and M, at most discarding a thin border of unused pixels at the end.

To efficiently implement this technique in a GPU-based renderer, and to avoid incoherent memory accesses, each pass should work on a different target image, effectively resulting in an *I-Buffer* pyramid. This way, considerable speedup is obtained thanks to the increased coherent access patterns to access to GPU memory. Note that memory requirements are the same as for Mip Mapping, about one-third of the final image size.

4.1. Hard shadows, textures and higher-frequency signals

The described interpolation scheme provides good results for low-frequency signals in a natural way. However, higher-frequency signals, like hard shadows or textures, require some extra work. To avoid missing details, we add an extra shadow check at the pixel program together with the cluster ID verification. For a single light, a simple boolean suffices to know whether a sample is illuminated or not, and from there the check for the shadow boundary is immediate. More lights would require multiple booleans, but they can be "compacted" in a single integer value, thus requiring a single equality verification in the shader. The results can be seen in Fig. 11. Other high frequency signals, such as textures, can be guaranteed in a similar way, as long as their frequency is not as high as to require a per-pixel evaluation, which would render our technique useless (e.g., hair, grass, fibers).

Here we should mention that the geometry itself may pose problems when large triangles are used. In this case, our approach has been to simply tessellate these polygons until they had a ratio of polygon area to on-screen-projected area below a user-provided threshold, which prevented our system to have problems in these situations.

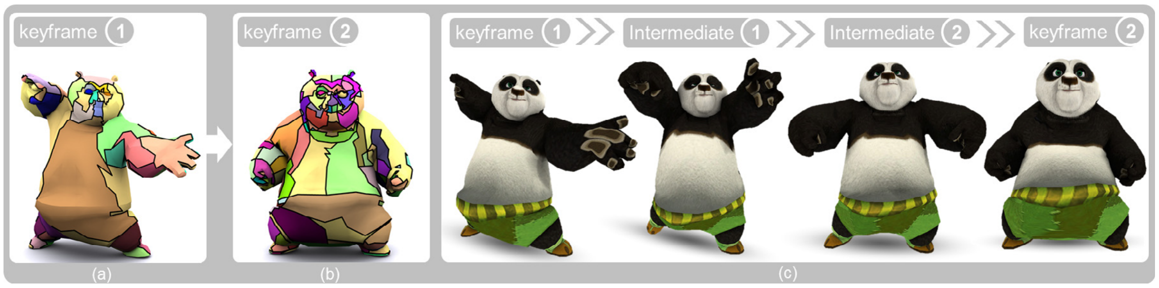

Fig. 6. To take into account animations (c), we iteratively refine the clustering for the different model keyframes (a and b).

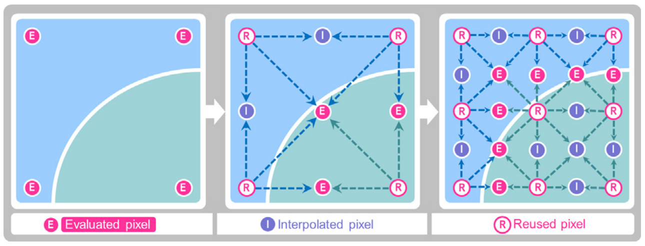

Fig. 7. We apply a rendering pattern for each of the three passes, starting at a 2 × 2-pixel image. Re-used (R), interpolated (I) and evaluated (E) samples in the successive passes. Blue and green regions represent two different, neighboring clusters. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

4.2. Automatic pass-controlling mechanism

One of the main issues that affect the performance is the number of passes (see Section 5). Optimizing it is a complex task, as it strongly depends on the scene itself, the distance from the observer, the complexity of the shading, among other factors. To avoid cumbersome manual trial and error tests, we introduce an automatic controlling mechanism that selects a number of passes trying to guarantee user-defined upper and a lower framerate bounds. If the current average framerate is smaller than the lower bound, the number of passes is increased. If it is above the upper bound, the number of passes is reduced. In practice, we have observed that, for medium and far distances, more than 5 passes provide too blurry results and some noticeable "swimmering" effects, so we kept it to be 4 at most. Observe that this approach does not strictly guarantee the framerate: if the user selects an extremely complex shading plus real-time bounds, the system will increase the number of passes up to the maximum allowed, never achieving the required bound.

5. Results

We integrated Aila and Laine's [40] GPU ray tracer into our application. For each triangle, we store the cluster ID as new texture coordinate for the model in addition of the existing ones, smoothly integrating this information to be used in runtime. All renderings in the paper are 1025 × 1025 pixels, and preprocessing times (i.e., clustering) range from 2 up to 5 min for all our models in our unoptimized code.

In Fig. 1 we can appreciate how our technique is able to reconstruct complex images involving complex features like visibility, illumination, and textures. The results are clearly comparable to those of traditional Ray-casting, at a fraction of the computational cost (see below). Also, our technique can handle soft and hard shadows (Figs. 11 and 10). textures (Fig. 12), hard shadow boundaries (Fig. 11), and even animation (Fig. 6).

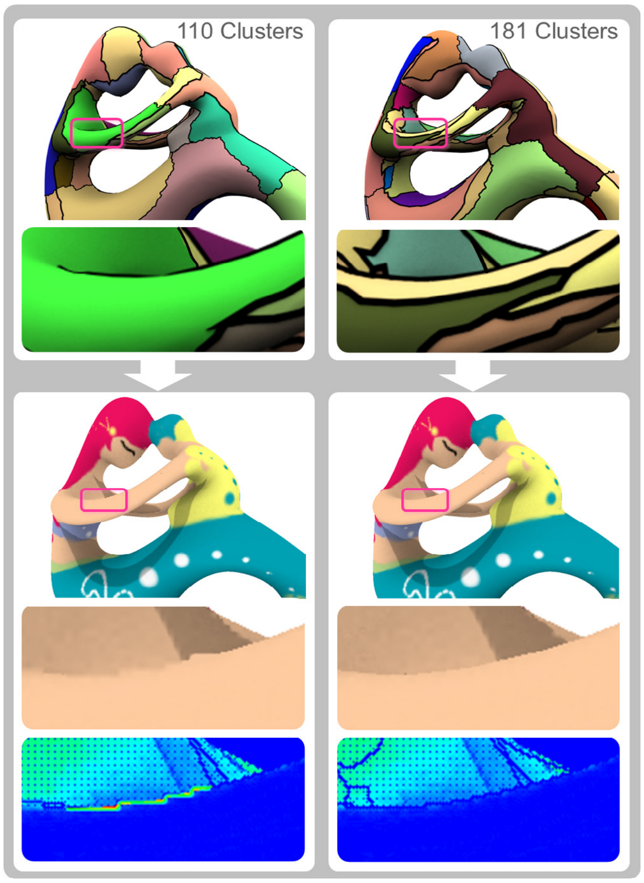

Fig. 8. Image quality depending on the number of clusters. Top: our clustering, Bottom: renders showing details (inset) and error values (bottom, in false color). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

As can be seen in Figs. 8 and 9, the two main factors that affect quality and performance are the number of clusters and, more significantly, the number of passes. In Fig. 8 we can see that the more clusters, the higher quality the final image will have. Textures and their mapping/stretching are well preserved (insets), but a correct visibility interpolation requires more clusters in this case, as can be seen in the error values shown at the bottom (colder colors mean lower error, while warmer areas mean higher error with respect to the ground truth render). In addition, in both cases frame-rates are similar: the model with 110 clusters requires 427.4 msec to be rendered (thresholds: Visibility = 0.54, Orientation = 0.3, Stretching = 0.1), while the one with 181 clusters requires 500.81 msec (thresholds: Visibility = 0.48, Orientation = 0.02, Stretching = 0.1). In any case, our experience shows that the more channels are added to the preprocessing, the

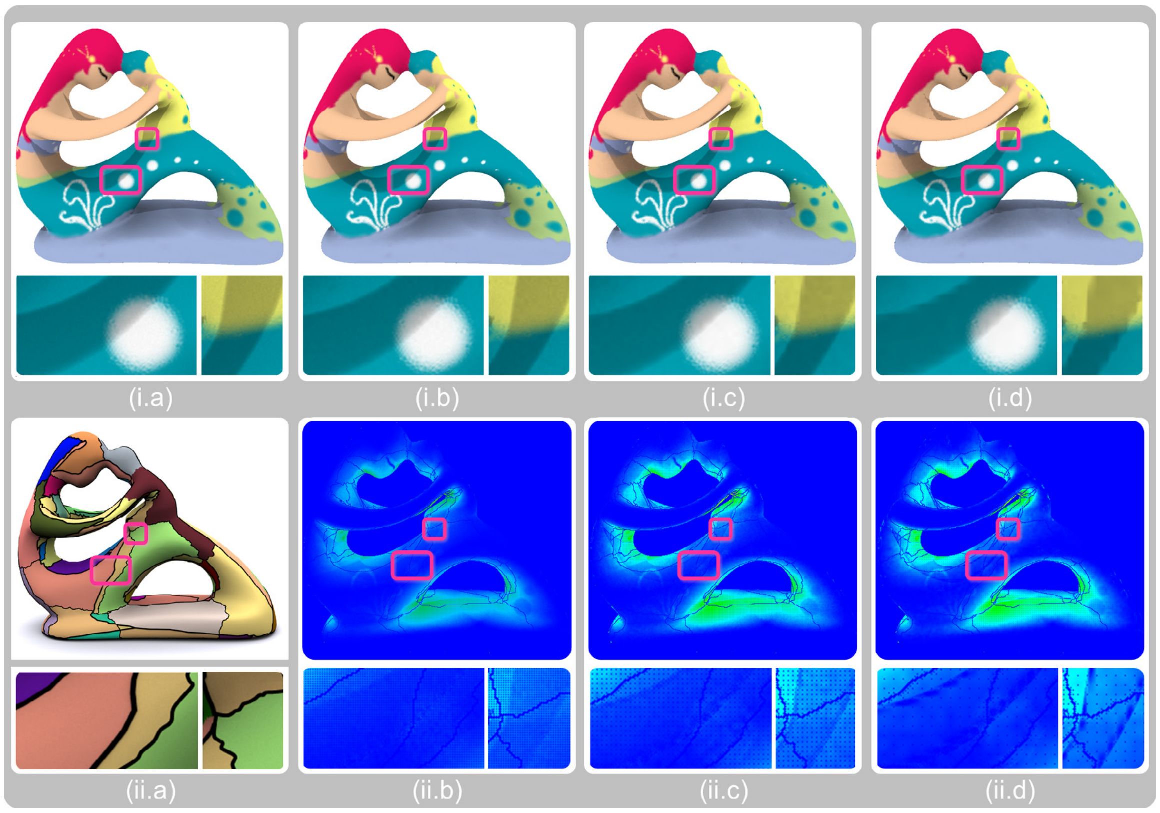

Fig. 9. Quality depending on the number of passes. Top row, from left to right: Ray-Casting reference and 2, 3 and 4 passes respectively. Bottom row: the clusters and the respective error images, computed with the L^2 metric, which show the increased error with the number of passes. The insets show details of the texturing and soft shadows. Timings are shown in Fig. 15.

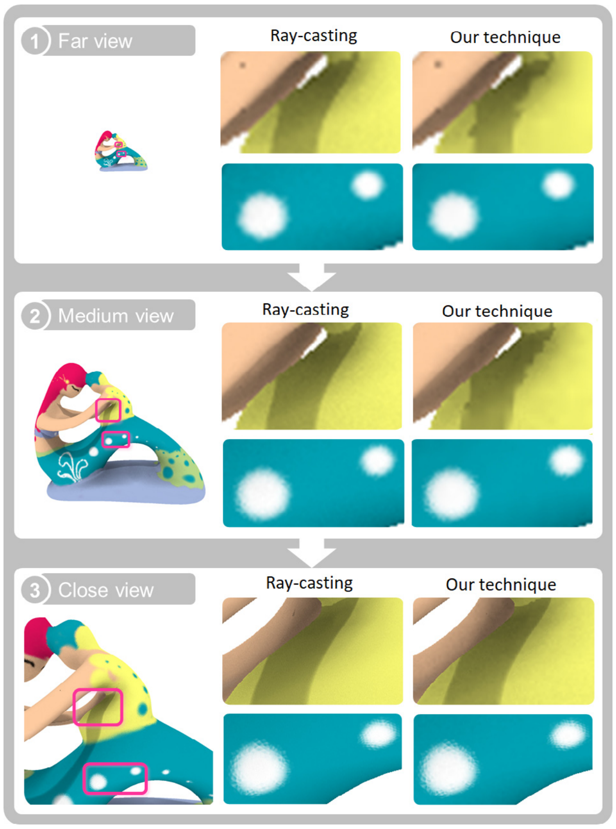

Fig. 10. Quality depending on the observer distance. Timings in Fig. 13.

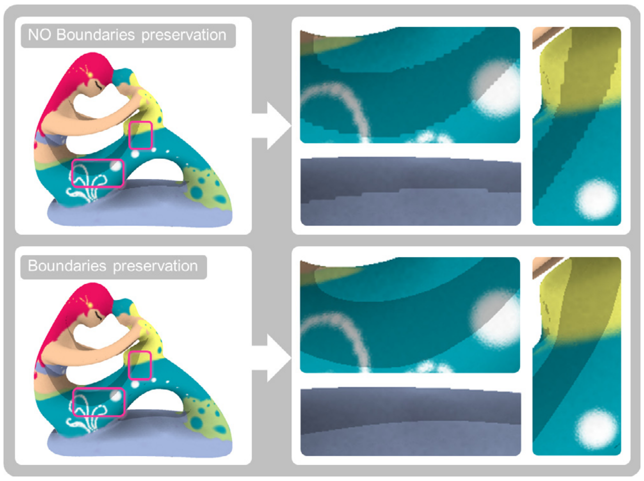

Fig. 11. Hard shadow boundary preservation. Left-center: Render with no extra boundary check. Right-center: Render with boundary preservation.

more relaxed the thresholds can be, still obtaining good quality results. On the other hand, in Fig. 9 we can see the dependence of the quality on the number of passes. It is important to remember this algorithm *approximates* the evaluation of some pixels by a much simpler interpolation scheme. By definition, this replacement must degrade the quality of the image, and the more pixels are interpolated instead of being computed, the worse the quality. Thus, as expected, we can see that the more passes, the more approximated the rendering, but the faster the model can be rendered (see below). This is only noticeable in the zoomed views

Table 1
Table showing the differences between the cost of evaluated and interpolated pixels for Ray-Casting and our technique for several models shown in the paper.

Models	Method	Evaluation	Interpolation
Dancing Kids	Ray Casting	21.7969	0
	our	22.3699	0.0142
Fertility	Ray Casting	20.85121	0
	our	24.6957	0.0140
Chinese Vase	Ray Casting	6.3608	0
	our	6.4115	0.0184

(top row), which show a good quality for the textures but some artifacts in the shadow details, which is reflected in the error map (bottom insets). Also, observe that the lowest error areas concentrate around boundaries and initial samples, and that these samples are more separated as we increase the number of passes.

In Fig. 10 we can see the dependency of the quality with respect to the distance to the observer: for a fixed number of passes, at short distances, more samples are evaluated over the model projection, resulting in a better quality affecting mainly the high-frequency shadows. However, as the model size on screen is smaller, the overall perceived quality is not reduced, even when less samples are needed. Also, it is important to notice that the texture signal is correctly preserved at any distance.

In Fig. 12 we can appreciate that our technique is fully compatible with the use of textures, as we use it to interpolate *texture coordinates* and not the textures themselves. As texture coordinates (lower insets) have a lower frequency than the textures themselves, their interpolation results in high quality interpolated renderings (upper insets), even with high frequency details. All the models used in the paper has been textured using multichart parameterizations. When texture coordinates are interpolated with our technique, it may happen that, for some convex charts, we could obtain interpolated points that fall outside. To avoid such situations we have used padding, but in extreme cases the technique described by González and Patow [41] could be used to sample the correct texture coordinates avoiding possible artifacts, i.e., color from outside the charts or from other charts leaking into the atlas.

As one would expect, the time needed to interpolate a single pixel is several orders of magnitude smaller than the time needed to fully evaluate it. To asses this claim we have rendered several models from a range of different viewing angles and distances. In our case, the model shown in Fig. 1 shows an average cost of 0.0142 ms for interpolated pixels vs 21.7969 ms for evaluated pixels. The Chinese Vase shown in Fig. 12 presents a similar difference, being 0.0184 ms the average cost of interpolation and 6.4115 ms the average cost of evaluation.

In Table 1 we can see a comparison of the times needed to evaluate and to interpolate a single pixel (in ms), averaged over a number of viewing angles and distance ranges. As we can see, the evaluation cost is several orders of magnitude slower than interpolation. In Fig. 13 we can see a graph showing the relation between the rendering time and the distance to the observer for the Fertility model (25K triangles). Observe that, at large distances, our pass-controlling mechanism guarantees at least the performance of Ray Casting.

Rendering with Ray-casting is, in general, $O = n^2E$, with n the side of an $n \times n$ image and E the cost of evaluating an average sample. Our technique changes this to be $O = nE + n^2I$, where we assume that the cost of reused samples (R) is roughly equal to the cost of the interpolated ones (I). The reason of this behavior is that cluster boundaries represent a 1D space embedded in the screen 2D space (actually, they are embedded in the 2D projection of the 3D object space), so they grow linearly while screen resolution grows quadratically. This can be appreciated in Fig. 15. Of course, as every pixel must have a value in the end, the interpolated (or reused) samples also need to be considered, resulting in an asymptotic quadratic behavior. However, the multiplicative constant of interpolated samples can be orders of magnitude smaller than the one for evaluated samples. As a result, the more complex the computation to perform, the better for our algorithm. Of course, this difference is completely dependent on the specific elements involved in computing a particular sample. If the exact evaluation of a sample is very cheap, then probably plain Ray-casting is a better option. However, in the case of a GPU-based implementation, as in our case, there are additional overheads introduced by the device. On one side, there is a constant overhead produced by the multiple kernel calls, although we found this to be negligible for our scenes. On the other, as threads in a GPU are executed in scheduling units called *warps* that execute in SIMD, there is an overhead that we can easily identify with the control divergence between evaluated and interpolated paths. As we have seen, evaluation is the dominant part of the computations, and the control divergence overhead can be associated to the samples that need to be evaluated (even if only one sample in a warp needs evaluation, this cost will drive the warp's total cost), thus resulting in an effective behavior of $O = nE' + n^2I$, with $E' = E + W$, and W is the overhead introduced by the divergence between the threads. Fig. 14 illustrates the costs of each term for a number of scenes and averaged over a number of different viewpoints, ranging from close to far distances. It is important to remark that this extra W cost does not change the overall behavior of the system, as Fig. 15 shows. In addition and to aim for fair comparisons, throughout all the paper our figures and data do not include background computations, which could bias the results.

This is the reason of our pass-controlling mechanism, explained before. On the other hand, if too many passes are selected, the constant C can grow large enough to remove any advantage obtained with our reusing mechanism. Fig. 15 we can see the behavior for a fixed medium-distance view, different resolutions and different number of passes.

6. Discussion and future work

We found that feature coherence at the low-variability regions in a scene, when projected in screen space, can be exploited to drastically accelerate Ray-casting-based rendering. Our findings show a dramatic acceleration in rendering speed of up to an order of magnitude, and with a quality similar to that of Ray-casting. We identify regions that share a common set of user-defined features (defined in a broad sense), and this allows us to compute the final image in a more efficient way than traditional approaches that usually compute each pixel from scratch. Our multi-pass acceleration technique approximates rendering samples at one pass by reusing values obtained at previous passes. One important feature of this technique is that it is completely independent of, and can be used in combination with any existing acceleration technique. Also, our technique is pixel-bound, which makes it independent of the tessellation of a given model. A conceptually similar screen-space strategy can be found in other works [9,10,16]. However, here we explore a new path by letting the user define clustering features with a completely general framework, incorporating even complex properties such as visibility, which has never been unified before with other features, as we do here.

Finally, it is worthwhile commenting that one could directly store, for each sample, the normals, a measure of texture stretching, and even visibility, although the latter would be more expensive to store. Then, instead of the cluster ID, one would need to

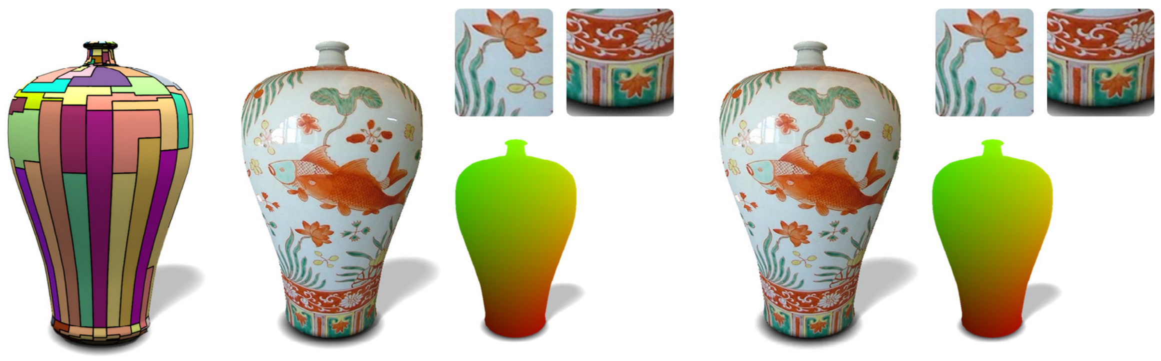

Fig. 12. Correct texture rendering. Left: clusters, Middle: Ray-casting, Right: Our technique. The lower insets show the smooth interpolation of the texture coordinates.

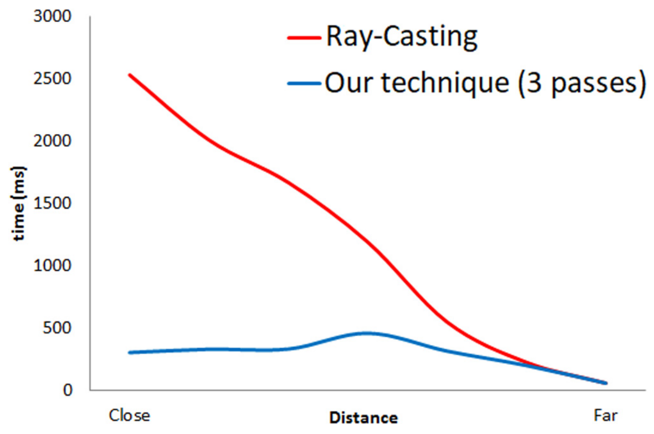

Fig. 13. Graph showing the relation between rendering time (in ms) and distance to the observer (in arbitrary units), for the model in Fig. 10.

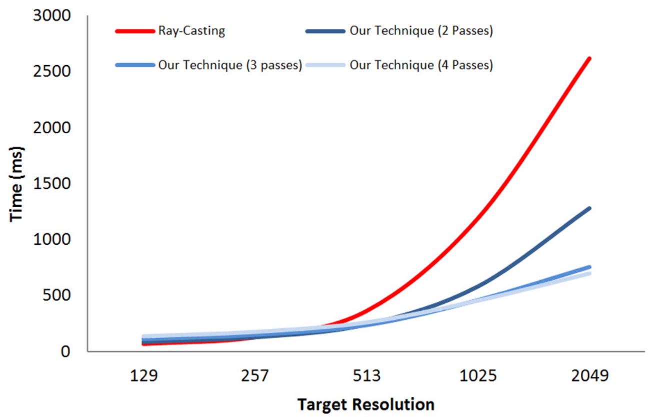

Fig. 15. Graph showing the relation between rendering time (in ms) and the number of passes for different resolutions, for the model in Fig. 9.

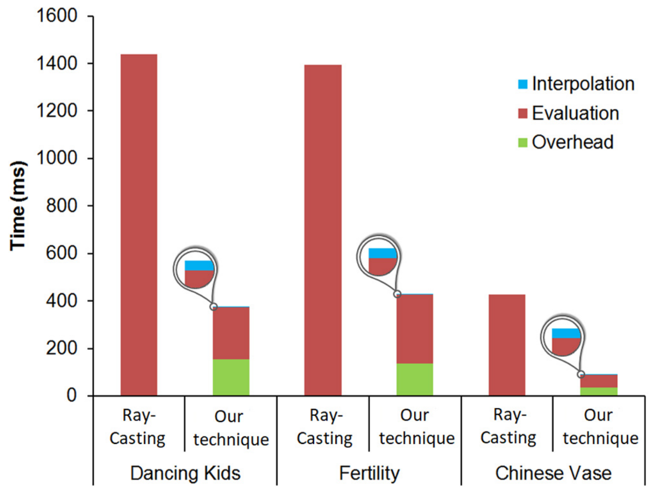

Fig. 14. Graph comparing the mean rendering time (in ms) between Ray Casting and our technique for several views and models.

define and set up new heuristics for the interpolation, probably different for each channel. Instead, a single cluster ID, as we do here, quickly answers the similarity question between two samples avoiding the need for new heuristics and a possible increase in memory requirements (e.g., visibility). In our case, two samples are similar if they have the same cluster ID, which is a quite simple comparison to do.

A drawback of our technique is that the clustering has to be done in a pre-processing stage, which requires knowing in advance, if not all, most of the properties of our scene. Future studies should explore whether this stage can be further parallelized in order to perform this in real-time. As our implementation uses a reduced-cost, lazy-evaluation approach, we foresee this as a feasible objective. On the other hand, other more standard propagation algorithms (such as Markov Random Fields while modeling the mesh as a graph) and/or clustering algorithms (k-means) would have been valid options for the clustering stage. Further testing and side-by-side comparisons would be needed to select the best among these options. Also, we use a simple linear interpolation scheme in our implementation, but more advanced forms of interpolation (bicubic as opposed to bilinear) are likely to improve results. It is quite obvious that higher-order schemes would provide smoother results, with the risk of introducing some kind of artifacts, such as banding. Again, an in-depth analysis comparing performance and quality is left as future work.

As can be seen throughout the paper, we have mostly applied the presented technique to single objects. General 3D scenes such as an office or a classroom would follow the same processing, clustering and rendering of the geometry, but some changes would be required in the computation of the *Visibitily Channel* (see Section 3.1.1): Instead of generating samples on an enclosing sphere as we currently do, we would require to uniformly sample the whole 3D scene, discarding all the samples that are inside an object or outside of the enclosing room. This situation is the same when considering specular reflections, but in this case reflections would require a more dense sampling scheme near the surfaces of the object. If no self-reflections are considered, the method presented so far can be used without changes. However, inter-reflections from the specular object itself may require computing short rays that might intersect the object surface again, thus requiring the same uniform space sampling strategy as for an office. However, these cases pose more complex problems, as in a bounding sphere the distance to the object is more or less homogeneous, while in a generic space-sampling scheme, there could be plenty of specific cases which should be carefully considered. We leave research for these kinds of cases for future work. Following the same reasoning, this technique can deal quite positively with global illumination effects, which are, in general, of low frequency. However, high frequency effects such as caustics remain as an avenue for further research.

Feature-based coherence exploitation is a powerful and promising avenue for rendering acceleration, and information theoretic tools appear as an excellent path to unleash this power. It is clear that techniques like the one presented here actually *benefit* from evaluation complexity: the more complex the computations performed for each pixel, the larger the acceleration ratio. We think that these techniques will increase in relevance as shaders continue to grow in complexity and size.

CRediT authorship contribution statement

Francisco González García: Methodology, Software, Investigation, Conceptualization, Writing – original draft, Writing – review & editing. **Ignacio Martin:** Software, Visualization, Conceptualization, Writing – original draft, Writing – review & editing. **Gustavo Patow:** Formal analysis, Investigation, Supervision, Conceptualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was partially funded by the TIN2017-88515-C2-2-R grant from Ministerio de Ciencia, Innovación y Universidades, Spain.

References

[1] Foley JD, van Dam A, Feiner SK, Hughes JF, Phillips R. Introduction to computer graphics. Addison-Wesley; 1993.

[2] Pharr M, Humphreys G. Physically based rendering: from theory to implementation. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2004.

[3] Wald I, Mark WR, Günther J, Boulos S, Ize T, Hunt W, et al. State of the art in ray tracing animated scenes. In: Schmalstieg D, Bittner J, editors. STAR proceedings of eurographics 2007. The Eurographics Association; 2007, p. 89–116.

[4] Zhou K, Hou Q, Wang R, Guo B. Real-time KD-tree construction on graphics hardware. ACM Trans Graph 2008;27(5):126:1–126:11.

[5] Lauterbach C, Garland M, Sengupta S, Luebke D, Manocha D. Fast BVH construction on GPUs. Comput Graph Forum 2009;28(2):375–84.

[6] Garanzha K, Pantaleoni J, McAllister D. Simpler and faster HLBVH with work queues. In: Proceedings of the ACM SIGGRAPH symposium on high performance graphics. New York, NY, USA: ACM; 2011, p. 59–64.

[7] Sitthi-Amorn P, Modly N, Weimer W, Lawrence J. Genetic programming for shader simplification. ACM Trans Graph 2011;30(6):152:1–152:12.

[8] Novák J, Dachsbacher C. Rasterized bounding volume hierarchies. Comput Graph Forum 2012;31(2pt2):403–12.

[9] Akimoto T, Mase K, Suenaga Y. Pixel-selected ray tracing. IEEE Comput Graph Appl 1991;11:14–22.

[10] Bala K, Dorsey J, Teller S. Radiance interpolants for accelerated bounded-error ray tracing. ACM Trans Graph 1999;18(3):213–56.

[11] Szirmay-Kalos L, Aszódi B, Lazányi I, Premecz M. Approximate ray-tracing on the GPU with distance impostors. Comput Graph Forum 2005;24(3):695–704.

[12] Gonzalez F, Patow G. Continuity and interpolation techniques for computer graphics. Comput Graph Forum 2016;35(1):309–22. http://dx.doi.org/10.1111/cgf.12727.

[13] Nehab D, Sander PV, Lawrence J, Tatarchuk N, Isidoro JR. Accelerating real-time shading with reverse reprojection caching. In: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on graphics hardware. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association; 2007, p. 25–35.

[14] Sitthi-amorn P, Lawrence J, Yang L, Sander PV, Nehab D, Xi J. Automated reprojection-based pixel shader optimization. ACM Trans Graph 2008;27(5):127:1–127:11.

[15] Yang L, Sander PV, Lawrence J. Geometry-aware framebuffer level of detail. Comput Graph Forum 2008;27(4):1183–8.

[16] Adamson A, Alexa M, Nealen A. Adaptive sampling of intersectable models exploiting image and object-space coherence. In: Proceedings of the 2005 symposium on interactive 3D graphics and games. New York, NY, USA: ACM; 2005, p. 171–8.

[17] Bitterli B, Wyman C, Pharr M, Shirley P, Lefohn A, Jarosz W. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. ACM Trans Graph 2020;39(4). http://dx.doi.org/10.1145/3386569.3392481.

[18] Crespo M, Jarabo A, noz AM. Primary-space adaptive control variates using piecewise-polynomial approximations. ACM Trans Graph 2021;40(3):1–15. http://dx.doi.org/10.1145/3450627.

[19] Bako S, Vogels T, McWilliams B, Meyer M, Novák J, Harvill A, et al. Kernel-predicting convolutional networks for denoising Monte Carlo renderings. In: Proceedings of SIGGRAPH 2017. ACM Trans Graph 2017;36(4):97:1–97:14. http://dx.doi.org/10.1145/3072959.3073708.

[20] Chaitanya CRA, Kaplanyan AS, Schied C, Salvi M, Lefohn A, Nowrouzezahrai D, et al. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. ACM Trans Graph 2017;36(4):98:1–98:12. http://dx.doi.org/10.1145/3072959.3073601, URL: http://doi.acm.org/10.1145/3072959.3073601.

[21] Vogels T. Denoising Monte Carlo renderings with convolutional neural networks (Master's thesis), Switzerland: ETH Zurich; 2016.

[22] Vogels T, Rousselle F, McWilliams B, Röthlin G, Harvill A, Adler D, et al. Denoising with kernel prediction and asymmetric loss functions. In: Proceedings of SIGGRAPH 2018. ACM Trans Graph 2018;37(4):124:1–124:15. http://dx.doi.org/10.1145/3197517.3201388.

[23] Gharbi M, Li T-M, Aittala M, Lehtinen J, Durand F. Sample-based Monte Carlo denoising using a kernel-splatting network. ACM Trans Graph 2019;38(4):125:1–125:12. http://dx.doi.org/10.1145/3306346.3322954, URL: http://doi.acm.org/10.1145/3306346.3322954.

[24] Zwicker M, Jarosz W, Lehtinen J, Moon B, Ramamoorthi R, Rousselle F, et al. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. In: Computer graphics forum (proceedings of eurographics - state of the art reports), vol. 24. (2):2015, p. 667–81. http://dx.doi.org/10.1111/cgf.12592.

[25] Sander PV, Snyder J, Gortler SJ, Hoppe H. Texture mapping progressive meshes. In: Proceedings of the 28th annual conference on computer graphics and interactive techniques. New York, NY, USA: ACM; 2001, p. 409–16.

[26] Sander P, Wood Z, Gortler S, Snyder J, Hoppe H. Multi-chart geometry images. In: Proceedings of the 2003 eurographics/acm SIGGRAPH symposium on geometry processing. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association; 2003, p. 146–55.

[27] Katz S, Tal A. Hierarchical mesh decomposition using fuzzy clustering and cuts. ACM Trans Graph 2003;22(3):954–61. http://dx.doi.org/10.1145/882262.882369, URL: http://doi.acm.org/10.1145/882262.882369.

[28] Günther J, Friedrich H, Wald I, Seidel H-P, Slusallek P. Ray tracing animated scenes using motion decomposition. Comput Graph Forum 2006;25(3):517–25, (Proceedings of Eurographics).

[29] Chen X, Golovinskiy A, Funkhouser T. A benchmark for 3D mesh segmentation. ACM Trans Graph (Proc. SIGGRAPH) 2009;28(3).

[30] Kalogerakis E, Hertzmann A, Singh K. Learning 3D mesh segmentation and labeling. ACM Trans Graph 2010;29(3).

[31] Lucquin V, Deguy S, Boubekeur T. SeamCut: Interactive mesh segmentation for parameterization. In: SIGGRAPH Asia 2017 technical briefs. New York, NY, USA: ACM; 2017, p. 25:1–4. http://dx.doi.org/10.1145/3145749.3149435, URL: http://doi.acm.org/10.1145/3145749.3149435.

[32] Feixas M, Sbert M, González F. A unified information-theoretic framework for viewpoint selection and mesh saliency. ACM Trans Appl Percept 2009;6(1):1:1–23.

[33] Ruiz M, Bardera A, Boada I, Viola I. Automatic transfer functions based on informational divergence. IEEE Trans Vis Comput Graphics 2011;17(12):1932–41.

[34] Müller P, Zeng G, Wonka P, Van Gool L. Image-based procedural modeling of facades. ACM Trans Graph 2007;26(3). http://dx.doi.org/10.1145/1276377.1276484, URL: http://doi.acm.org/10.1145/1276377.1276484.

[35] Limper M, Kuijper A, Fellner D. Mesh saliency analysis via local curvature entropy. In: Proceedings of the 37th annual conference of the european association for computer graphics: short papers. Goslar Germany, Germany: Eurographics Association; 2016, p. 13–6. http://dx.doi.org/10.2312/egsh.20161003.
[36] Shamir A. A survey on mesh segmentation techniques. Comput Graph Forum 2008;27(6):1539–56.
[37] Sbert M, Feixas M, Rigau J, Chover M, Viola I. Information theory tools for computer graphics. Synthesis lectures on computer graphics and animation, Morgan and Claypool Publishers Colorado; 2009.
[38] Degener P, Meseth J, Klein R. An adaptable surface parameterization method. In: Proceedings of the 12th international meshing roundtable. 2003, p. 201–13.
[39] Lin J. Divergence measures based on the Shannon entropy. IEEE Trans Inf Theory 1991;37(1):145–51.
[40] Aila T, Laine S. Understanding the efficiency of ray traversal on GPUs. In: Proceedings of the conference on high performance graphics 2009. New York, NY, USA: ACM; 2009, p. 145–9.
[41] González F, Patow G. Continuity mapping for multi-chart textures. ACM Trans Graph 2009;28(5):109:1–8.