IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 29, 2019, accepted April 30, 2019, date of publication May 20, 2019, date of current version June 6, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2917721

GPU Sparse Ray-Traced Segmentation

LUCIAN PETRESCU, ANCA MORAR, FLORICA MOLDOVEANU, (Member, IEEE), AND ALIN MOLDOVEANU

Department of Computer Science, Faculty of Automatic Control and Computers, Politehnica University of Bucharest, RO-060042 Bucharest, Romania

Corresponding authors: Lucian Petrescu (lucian.petrescu.24@gmail.com) and Anca Morar (anca.morar@cs.pub.ro)

This work was supported by the European Union's Horizon 2020 Research and Innovation Program under Agreement 643636 (Sound of Vision project – www.soundofvision.net).

ABSTRACT This paper introduces a real-time region growing segmentation algorithm, designed for graphics processing units (GPUs), which labels only a fraction of the input elements. Instead of searching locally around each element for strong similarity, like state-of-the-art segmentation and pre-segmentation methods do, the proposed algorithm searches both locally and remotely, using a unique ray tracing-based search strategy, which quickly covers the segmentation search space. The presented algorithm fully exploits the parallelism of the GPUs, sparsely segmenting high-resolution images (4K) in real-time on low range laptops and other mobile devices, approximately 5× times faster than the state-of-the-art simple linear iterative clustering (SLIC). While this paper demonstrates the results with images, the algorithm is trivially modifiable to work with input sets of any dimension. In contrast to the state-of-the-art real-time GPU methods, this algorithm doesn't require additional merging steps, as pre-segmentation methods do, and it produces complete segmentation. Additionally, post-segmentation optional stages for complete labeling and region merging on the GPU are also provided, although they are not always necessary.

INDEX TERMS GPU, image segmentation, parallel processing, ray tracing search strategy, sparse segmentation.

I. INTRODUCTION

Segmentation algorithms play an important role in computer vision, partitioning sets into unique non-overlapping subsets, where each sub-set has similar elements. Segmentation methods are common building blocks for a large variety of algorithms and data processing pipelines, being used in many industries.

Due to segmentation being a critical step in a large number of problems, there is now a vast amount of research on this subject, with many different approaches: clustering and dual clustering [1]–[3], histogram [11], edge detection [7], [11], region growing [3]–[5], graph partitioning [8]–[11], watershed [11], [26], adaptive thresholds [6], split and merge [7], [11], mean shift and mode seeking [10], [13], [24], [31], hierarchical [11], [15] and active contours [11], [17], [18].

Recent research has focused on trainable segmentation methods [11], [19]–[22] and co-segmentation [11], [16] using machine learning techniques to produce highly accurate results, with weak supervision. Nevertheless, these methods do not present themselves as solutions to real-time segmentation, as their execution costs dwarf the computational budgets of non-specialized hardware, especially on consumer laptops and mobile hardware.

Since the computational cost is a critical aspect of algorithm usefulness, another recent trend is to exploit hardware parallelism in order to maximize speed. This has led to the development of quick iterative GPU segmentation methods [1], [5], [10], [13], [23]–[27] and pre-segmentation techniques. Pre-segmentation methods [2], [6], [7], [13], [28]–[33] create precise local micro clusters, based on local similarity, but afterwards require an expensive clustering process. Without the costly final clustering process, they produce the fastest local results out of all the segmentation methods. Still, even these fast methods require iterative processes.

It is worth noting that the aim of segmentation is to help in extracting information from the input set. In such processes a good enough solution is sufficient for a large majority of the encountered cases. In many industrial applications of real-time segmentation, an acceptable quality at a very high speed is particularly more useful than superior quality at

The associate editor coordinating the review of this manuscript and approving it for publication was Jiachen Yang.

a comparatively much lower speed. Low cost segmentation is essential for object tracking and recognition, navigation and perception for autonomous robots, perception enhancing devices or computer controlled medical tools.

FIGURE 1. Results of sparse ray-traced segmentation with different sparsity settings.

In this paper a novel segmentation algorithm is introduced, which sparsely labels input sets by growing regions with ray tracing, starting from initial seeds. It is designed to fully harness the parallelism of GPUs, and benefit from many ray tracing optimizations (packet tracing [32], Digital Differential Analyzer (DDA) [33]). Figure 1 shows how sparse segmentation looks like in 2D, on a classic segmentation test image, and how different levels of precision can be obtained with different sparsity settings. The sparsity, or the seed density, is parameter controlled. Even with a high sparsity the resulting segmentation is of sufficient quality for many real-time applications. In this paper, the algorithm is mostly discussed in 2D, but it can be implemented in any dimension.

Our sparse ray-traced segmentation includes the following contributions:

1) ray-traced based region growing, leading to sparse segmentation, prioritizing the fast space coverage of the set over exact local coverage;
2) a constant number of GPU passes, conducting to a constant execution speed, without affecting quality;
3) low bandwidth usage due to sparsity;
4) a complexity of $O(N/tsize)$, where N is the number of elements from the input set (pixels) and $tsize$ is the tile size for which a seed is generated (e.g. 16×16). The state-of-the-art SLIC [28], [29] and Really Quick Shift [13] methods have complexities of $O(N)$ and $O(d \cdot N^2)$ where d is the dimensionality of the input set.

The purpose of the algorithm described in this paper is to provide a very fast low-level image segmentation on low power GPUs. This algorithm is suitable for applications which require a real-time processing of the environment images (navigation systems, robotics, automotive, surveillance), especially of depth images.

Our GPU sparse ray-traced segmentation has proven its usefulness in the Sound of Vision project [34], where it was included in a real-time processing pipeline of an assistive system for visually impaired people.

The Sound of Vision system's goal is to scan the environment, extract information of interest from RGBD or depth images and send it to the user, codified through haptics and sounds. The codification of the information from the environment requires object recognition. In our system, the object recognition uses the result of the low-level segmentation described in this paper, which produces regions representing planar surfaces. The regions are merged into objects, which are then assigned to different semantic classes (ground, walls, ceiling, stairs, doors, generic static and dynamic objects), using regions adjacency, geometric heuristics and inter-frame consistency. Each detected object is described by its semantic label, 3D position and bounding box relative to the camera. The Sound of Vision processing pipeline is detailed in [35] and [38]. In the initial prototypes of the Sound of Vision system, a scan-line planar segmentation described in [36] was included in the pipeline. However, in the final prototype this segmentation was replaced with our GPU sparse ray-traced segmentation, leading to faster and more accurate results in detecting objects of interest in indoor environments.

II. RELATED WORK

Segmentation on GPUs is generally achieved through iterative methods, where a sizeable number of iterations is necessary to converge the labeling to a final state. GPU pre-segmentation methods follow a similar strategy, which leads to over-segmentation with high quality local labeling.

Most of the early GPU segmentation work was done with medical applications in mind, motivated by large tridimensional datasets which benefitted most from GPU acceleration. Schenke *et al.* [5] investigated early opportunities presented by parallel computation hardware and introduced a hybrid CPU-GPU segmentation algorithm, based on seeded region growing through dilate and erode operations. Hagan and Zhao [1] used an extended Lattice Boltzmann Model (LBM) to solve the level set equation, in an iterative approach which generates high quality labeling at the expense of a large number of iterations with CPU-GPU synchronization.

Vineet and Narayanan [10] adapted the maxflow/mincut algorithm to CUDA, in which graph cuts are used to partition an initially single labeled set into a multitude of independently non-overlapping labeled sets. A costly synchronization for the graph re-labeling stage is used between each of the many graph cut iterations. Roberts *et al.* [24] used a level-set based iterative algorithm with $O(N \cdot \log(N))$ complexity.

Körbes *et al.* [26] introduced an iterative parallel watershed algorithm, where an image is divided into 16 × 16 tiles, and each tile computes a small local watershed transformation, for each iteration of the algorithm. Collins *et al.* [25] mapped the co-segmentation problem to linear algebra operations, which were then solved with CUDA, offering a high-quality solution at a high computational cost. Ramírez *et al.* [27] segmented volumes with a GPU adaptation of GrabCut, a flow network algorithm designed to partition images. Like [10], the push-relabel algorithm is implemented in CUDA, requiring costly synchronizations.

Recent GPU segmentation investigations are based on trainable methods [11], [19]–[22], in which various weakly supervised machine learning algorithms are used to learn and detect information in datasets. Segmentation is thus performed with machine learning methods such as Support Vector Machines (SVM), Markov Random Fields (MRF), Conditional Random Fields (CRF) or Fully Convoluted Networks (FCN). However, most of these methods are not adequate for real-time segmentation yet.

Pre-segmentation algorithms solve the segmentation problem only locally, usually with gradient ascent strategies, where seeds are moved iteratively in local vicinities, labeling the image at a local level, in small clusters named superpixels. Pre-segmentation algorithms have the highest performance levels out of all segmentation methods. Because of their speed, the pre-segmentation methods are excellent candidates for real-time processing pipelines, as pre-segmentation methods can be combined with cheap region merging strategies. Thus, they are preferred over complete segmentation algorithms in performance critical applications. Fulkerson *et al.* [31] used conservative over-estimation of small regions to produce superpixels, which are not fixed in approximate size or number. Levinshtein *et al.* [30] introduced TurboPixels, which are superpixels computed with geometric flows, where initial seeds are iteratively perturbed to cover local vicinities. The algorithm has a complexity of $O(N)$, and it works as a series of dilation operations.

Fulkerson and Soatto [13] provide a CUDA compatible alteration of the quick shift algorithm, in which a five-dimensional feature-space is used to establish the connection strength between pixels and clusters. The implementation has $O(d \cdot N^2)$ complexity, but it is very fast in practice as it is non-iterative, its only obstruction being the weak control over the size and compactness of the resulting superpixels.

Achanta *et al.* [28] introduced SLIC superpixels, clustering pixels by color and similarity in an iterative fashion but limiting the search space to a region proportional to the superpixel size. The complexity of the algorithm is $O(N)$. Achanta *et al.* [29] improved upon [28] with a zero-parameter variant of the SLIC algorithm. Li *et al.* [7] improved the quality of SLIC segmentation by using an iterative split-and-merge strategy. For each iteration the superpixels are split with an edge-map and then are merged with the adjacent superpixel with the shortest Bhattacharyya distance. This method trades off execution speed for better quality. Li and Chen [2] used linear spectral clustering to further increase the segmentation accuracy of SLIC, albeit at a large increase in computational costs.

Out of all segmentation and pre-segmentation methods discussed in this section, the ones that are the most pertinent towards real-time segmentation are Really Quick Shift [13] and SLIC [29]. Both methods use a local gradient ascent clustering strategy and they also offer good enough segmentation quality and the best time performance.

III. METHOD

A. REGION GROWING STRATEGIES

K-dimensional segmentation is a partitioning problem in which a set S containing N K-dimensional elements needs to be partitioned into non-overlapping sub-sets, based on both local and global similarity between elements. This problem is solved by searching S for the non-overlapping sub-sets and picking the best solution, given a fitness function.

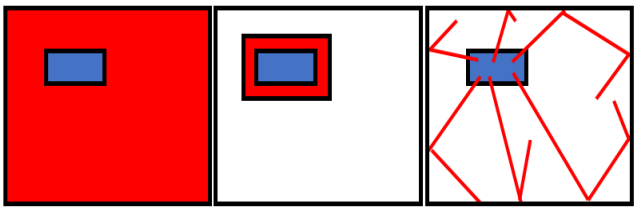

FIGURE 2. Segmentation search space strategies, in 2D: k-means (left), superpixel methods (center), sparse ray-traced segmentation (right).

Region growing or clustering methods determine this partitioning by iteratively trying to link elements outside the cluster, as shown in Fig. 2. The blue object represents a current cluster and the red pixels represent the search space for potential new cluster elements. Traditional methods such as k-means [3] explore the entirety of the set, producing extremely large search spaces, which are expensive to explore. Pre-segmentation region growing methods like SLIC or other superpixel methods [28], [29] limit the search space to only a small neighboring vicinity of the cluster, guaranteeing a quick search space exploration but also not being able to entirely detect very large segments. The strategy proposed by sparse ray-traced segmentation combines the best of both approaches: the small number of elements explored by the SLIC kernel approach and the large search space of the traditional methods, resulting in both fast execution speed and set-wide connection. Given the search space strategy, the proposed method leads to both small and large segments, in accordance to the particularities of the segmented space.

While the sparse ray-traced segmentation algorithm works identically on sets of any dimension, the algorithm is presented in the following sections in its 2D form. The output of the algorithm is a matrix with the resolution of the input image, containing the label for each pixel.

B. RAY TRACED REGION GROWING

The complete proposed algorithm is summarily depicted in Fig. 3. This section covers all the stages besides the optional

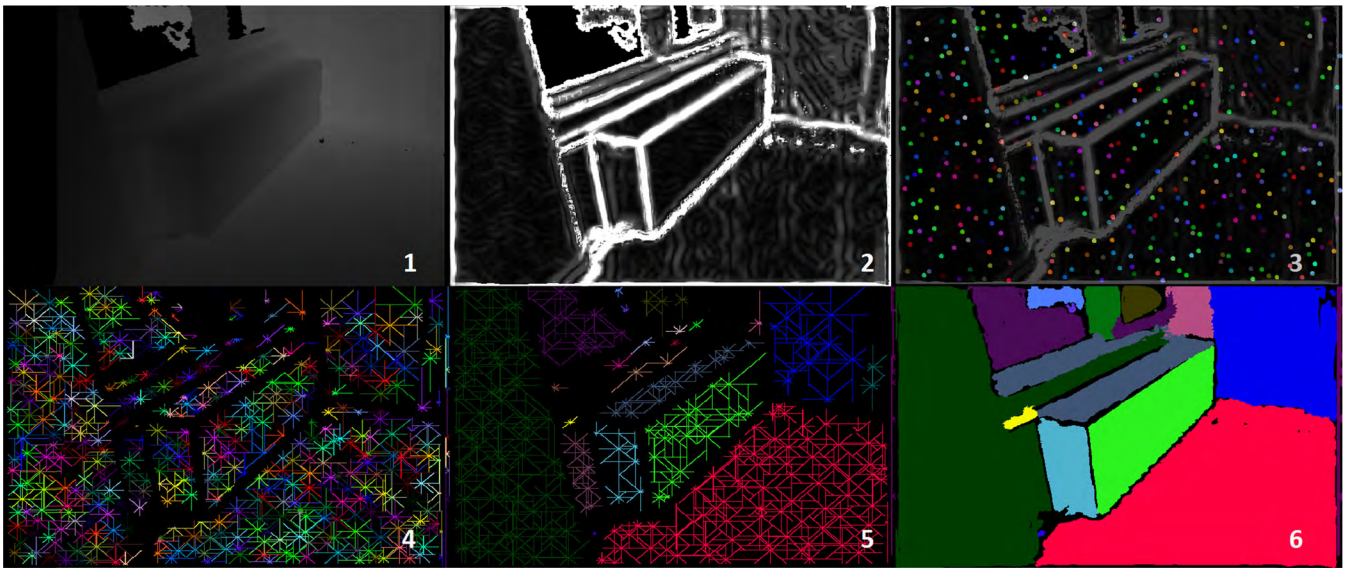

FIGURE 3. The sparse ray-traced segmentation pipeline, shown on a 2D depth image: Input depth image (1), flux computed based on input depth image and normal map (2), region seeds (3), net created by each seed (4), connected nets (5), result of complete labeling and region merging (6).

complete labeling and region merging: flux computation, seed generation, ray-traced search space exploration and connection of traced regions. All stages are parallelized on the GPU, using compute shaders (GLSL version 4.50). For each computation that runs in parallel, unidimensional or bi-dimensional workgroups of different size are used.

1) FLUX COMPUTATION

The flux represents the rate of change of the input set. It can be a simple adaptive gradient for color images, and it is thus extremely cheap to compute on the GPU.

In the Sound of Vision project, for depth images an exponential decay adaptive gradient (adjusted with the error of the acquisition device, Structure Sensor) was used. Depth images allow the reconstruction of the environment into 3D point clouds. Indoor environments are usually characterized by man-made planar surfaces, i.e., regions with the same normal in all their points. Therefore, for an indoor planar segmentation of depth images, the normal map [36] was also used in computing the flux. An example of flux generated based on a depth image is seen in Figure 3. 2.

In the Sound of Vision project [34], we used $((W + 7)/8, (H + 7)/8, 1)$ workgroups of size 8×8 for the normal estimation and flux computation, where W and H represent the width and the height of the image, respectively.

2) SEED GENERATION

The positions of the clustering seeds are first generated in a pseudo-random pattern across the entire set. For instance, any of the Sobol, van der Corput or Hammersly series can be used for an image.

After seed generation, the seeds use a gradient ascent strategy in order to distance themselves from high flux areas, using Pseudocode 1.

Pseudocode 1 Seed Positioning

```
seedPos ← getPseudoRandomPosition()
iterations ← 0
flux ← getFlux(seedPos)
while flux > Threshold and iterations < MaxIterations
    fluxDirection ← getFluxDirection(seedPos)
    seedPos ← seedPos + fluxDirection
    flux ← getFlux(seedPos)
    iterations ← iterations + 1
return seedPos
```

Each seed is processed in parallel (in our implementation we used $((S + 31)/32, 1, 1)$ workgroups, each workgroup containing 32 threads, where S represents the number of seeds).

An example of the resulting seed distribution can be seen in Fig. 3.3, colored with a Jenkins hash function applied over the seed index.

3) RAY-TRACED EXPLORATION AND CONNECTION OF SIMILAR REGIONS

After the seed positions have been generated, each seed traces multiple rays through the image in order to quickly cover the entire image, as illustrated in Figure 4: on the left, the input image and its pixels are illustrated, with an object shown in gray. In the middle, the result of applying the algorithm to the input image is illustrated (for simplicity, we use two rays per seed, with maximum 7 iterations per ray and without conservative rasterization). On the right, the sparse segmentation result on a high-resolution input image is shown (after seeds of rays with similar properties were connected).

Each ray is processed in parallel with a thread. In our implementation, we used 8 rays per seed, leading

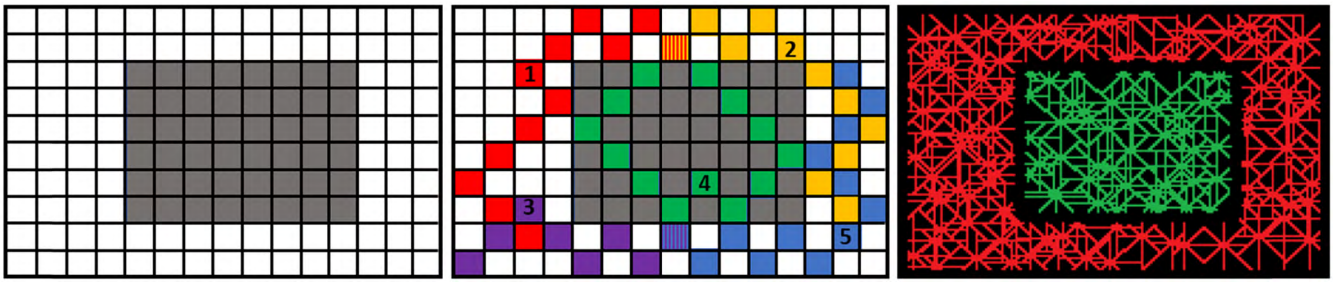

FIGURE 4. Input simplified image (left), result of applying the proposed method on the simplified image (middle), sparse segmentation result of a high-resolution image after seeds of rays with similar properties were connected (right).

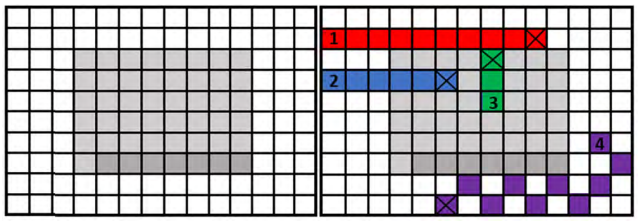

FIGURE 5. Input image with a low-density flux (left) and different ray tracing scenarios (right): Rays 1 and 4 end after a maximum number of steps, ray 2 ends because of accumulated flux and ray 3 ends on contact with another ray with different properties.

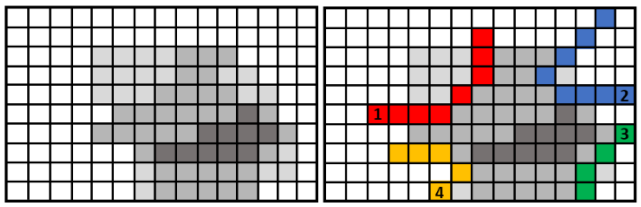

FIGURE 6. Input image with a flux of varying density (left), ray scattering depending on the strength of the flux (right).

to $((S+31)/32 \cdot 8, 1, 1)$ workgroups, each workgroup having 32 threads.

The rays have multiple termination cases, including: encountering a pixel where another ray with different properties was traced, encountering a high flux (edge) pixel or accumulating enough flux, as shown in Figure 5.

The rays also support multiple reflection on the edges of the image and on high flux (edge) pixels, making them efficient explorers even for complicated tracing cases. The many traced rays are easily distributed over all available compute cores. The result of the traced but unconnected rays can be observed in Fig. 3.4. Furthermore, direction scattering is used if the ray accumulates sufficient flux, in order to guide the ray from high flux to low flux areas, on the same principles as the watershed transform. The stochastic reflectance of rays towards low local flux deters the rays from choosing risky connections in high flux areas, as illustrated in Fig. 6.

A forest structure is created over the seeds, linking all the starting seeds over the entire image. Initially all seeds start as separate trees, having themselves as parents, and with each ray connection the number of single sized trees in the forest decreases. In order to minimize the height of the trees, when two rays connect, only the greatest ancestor

Pseudocode 2 Connecting Two Seeds

```
getGreatestParent(initial_seedId)
1: seedId ← initial_seedId
2:while seedId ≠ getParentId(seedId)
3:      seedId ← getParentId(seedId)
4: return seedId

ConnectSeeds (seedId1,seedId2)
1: parent1.id ← getGreatestParent(seedId1)
2: parent2.id ← getGreatestParent(seedId2)
3: if parent1.id < parent2.id
4:      atomicExchange(parent2.parentId, parent1.id)
5: else
6:      atomicExchange(parent1.parentId, parent2.id)
```

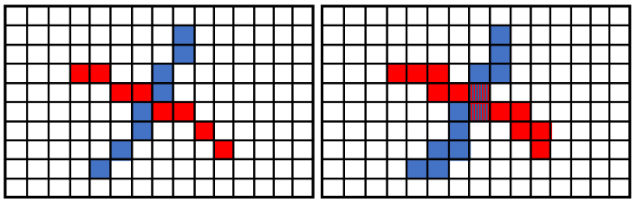

FIGURE 7. The difference between not using (left) and using (right) conservative rasterization for tracing.

seeds are connected. Because GPU data synchronization can only be efficiently implemented with lockless programming (atomic operations), a relationship order needs to be defined between seeds on a successful connection. The proposed method uses the minimum of the indices of the potentially connecting seeds. The algorithm for connecting two seeds is described in Pseudocode 2.

Conservative staircase tracing (conservative rasterization [37]) is used if the ray direction does not perfectly align to the image axes in order to guarantee the detection of ray connections even in aliased tracing scenarios, such as the one showcased in Fig. 7. Without conservative rasterization of rays some contact scenarios are lost (the two rays from the left image do not intersect themselves in any pixel).

Each ray maintains a set of properties about the traced input, including accumulated flux, average input value, variance and the average value for the last T traced pixels. The traced properties are written in an auxiliary buffer (the traced properties matrix *TProps* from Pseudocode 3).

Pseudocode 3 Ray tracing

```
Initialization Pass:
initialize label matrix Labels to empty
initialize traced properties matrix TProps to empty

Ray Tracing Pass (1 thread per each ray per seed):
seed ← getSeed (ray.id)
rayProperties ← initializeRayProperties()
iterations ← 0
pos ← seed.pos
while iterations < MaxIterations
  flux ← readFlux (pos)                    //from flux matrix
  if flux > MaxFlux
    end                                    //high flux (edge) termination case
  input ← readInput (pos)                  //from input image
  updateRayPropertiesInputFlux (rayProperties,
input,
                                           flux)
  if rayPropertiesDeviateFromCluster(rayProperties)
    end                                    //average value of the last T traced pixels
                                           //deviates from average value of the ray
  if (rayProperties.accumulatedFlux > AccMaxFlux)
    end                                    //accumulating enough flux termination
case
  scatterDir ← getScatteringBasedOnAccumFlux()
                                           //see Figure 6
  id ← readLabel (pos)
  if id is undefined
                                           //the pixel has not been traversed by other
rays
    writeLabel (pos, seed.id)
    writeTraceProperties(pos, rayProperties)
  else
    if alreadyConnectedNets(seed.id,id)
      end                                  //the seeds are already part of the
same
                                           //forest (they have the same greatest parent)
    posProperties ← readProperties(pos)
    if rayProperties not_similar posProperties
      end
    updateRayProper-
ties(rayProperties,posProperties)
//update with the properties of the other ray traced
through pos
    writeTraceProperties(pos, rayProperties)
    ConnectSeeds(seed.id, id)
  if ( ray.dir + scatterDir) aligned to image axes
    pos ← pos + ray.dir + scatterDir
  else
    pos ← StaircaseTracing (ray.dir + scatter-
Dir)
```

A potential seed connection occurs when a ray is about to be traced over another. The connection is successful only if there are both local and global matches between the potentially connected seeds. The following pseudocode describes this process:

Finally, after all the rays are traced, there remains a small possibility that two rays end near one another but neither initiates a connection case, therefore an extra pass is needed to enforce connectivity. The algorithm for the enforced connectivity is presented in Pseudocode 4.

Pseudocode 4 Enforce Connectivity Pass (per Each Pixel)

```
pos ← getPos(pixel)
seedId ← readLabel (pos)
properties ← readProperties(pos)
flux ← readFlux(pos)
foreach posN neighbor of pos
  seedNId ← readLabel(posN)
  propertiesN ← readProperties(posN)
  fluxN ← readFlux(posN)
  if properties similar propertiesN
    if max(flux, fluxN) < MaxFlux
      ConnectSeeds (seedId, seedNId)
```

The final pass updates each pixel to its greatest parent. The final parentless seeds, acting as roots in the forest constructed over the image, represent the unique sparse segmentation ids, as illustrated in Fig. 3.5.

In our implementation, this final pass is done in parallel using $((W + 31)/32, (H + 31)/32, 1)$ workgroups of size 32×32.

The bandwidth of the presented method is low, as the number of memory operations is small. Packet tracing [32] can be used on the rays. Packing by seed leads to good results as each seed normally traces rays with comparable length and similar connections. The staircase tracing pattern can be improved upon by modifying the classic DDA [33] rasterization algorithm and making it conservative. This eliminates a sizeable number of unnecessary memory operations. Using predefined directions can also further decrease the tracing computational costs.

Many of the constants from Pseudocode 3 (MaxFlux, AccMaxFlux, etc.), were established experimentally in the Sound of Vision project [34], for optimal results in segmenting depth maps of indoor environments. These constants represent a weakness of the approach, and we are currently researching ways to automatically parametrize the boundary exploration.

C. COMPLETE LABELING AND REGION MERGING

Sparse segmentation is sufficient for a large number of practical applications, but sometimes a full segmentation is desired. The complete labeling and region growing optional steps are presented for this purpose.

1) COMPLETE LABELING

After the initial sparse labeling described in the previous section, the unique segmentation ids can be considered as a sparse net over the image. Full labeling is implemented by

connecting the pixels lacking labels to this net, in two stages. In the first stage the pixels are connected under the same principles of ray tracing connection, as used in the previous section. In the second stage the pixels are connected based on a similarity search inside a kernel.

The first stage of the complete labeling is described in Pseudocode 5.

Pseudocode 5 Full labeling – first stage (per each pixel)

```
pos ← getPos(pixel)
if readLabel(pos) not undefined
    end
steps ← 0
for rayId ← 1, rayId < NumRays, rayId ++
    alive[rayId] ← true
    rayProperties[rayId] ← empty
for steps ← 1, steps < MaxSteps, steps ++
  for rayId ← 1, rayId < NumRays, rayId ++
      if not alive[rayId]
          continue
      ray ← getRay(rayId)
      if ray.dir aligned to image axes
          rpos ← pos + steps · ray.dir
      else
          rpos ← StaircaseTracing(ray.dir)
      flux ← readFlux(rpos)
      if flux > MaxFlux
          alive[rayId] ← false
          continue
      input ← readInput(rpos)
      updateRayProperties(properties[rayId],
input)
      seedId ← readLabel(rpos)
      if seedId not undefined
          posProperties ← readProperties(rpos)
          if properties[rayId] similar posProp-
erties
              writeLabel(pos, seedId)
              end
          else
              alive[rayId] ← false
```

After the first stage, most of the pixels will be labeled. In the second stage, the unlabeled pixels which are not positioned on an edge (high flux) are processed in the following manner: for each pixel, the neighboring area is sampled with rays which intersect the regions inside the vicinity. The pixel will be assigned to the region with the most similar properties (with the minimum difference between the pixel's intensity and the average intensity inside the region).

The complete labeling process is applied iteratively, until all the pixels are labeled. Experimentally we have reached the conclusion that a very small number of iterations (∼2-3) is required to fully label most segmentation cases.

2) REGION MERGING

Region merging is another optional process, which can lead to significant quality improvements, as showcased in Fig. 8. It is used to combine similar regions which could not be united because of high flux variations. Like full labeling, it is an iterative multi-stage process.

In the first stage the fully labeled image is taken and each initial seed traces short rays to gather data and average properties of the local labeling.

FIGURE 8. Before (left) and after (right) the region merging step.

Then, in the second stage, the data that characterizes each initial seed (gathered in the first stage) is accumulated in the greatest parent of each initial seed, using atomic operations to synchronize the writes. Since this is an accumulation method, special care must be taken to protect against overflows. In the third stage a per-pixel method is ran which compares all potential neighbors in a kernel to the center pixel. If the properties match but the labeling is different, then the two labels are merged.

Similarly to the complete labeling process, we have determined experimentally that two or three iterations of region merging are enough for obtaining a high-quality segmentation, as can be observed in Fig. 8.

Both full labeling and region merging increase the quality of the labeling, and the extra costs still lead to comparable overall processing times to those of the state-of-the-art (pre)segmentation methods, as shown in Table 1.

IV. EVALUATION AND RESULTS

The evaluation methodology is to compare sparse ray-traced segmentation both qualitatively and speed-wise to the fastest state-of-the-art GPU segmentation methods, that is, with Really Quick Shift [13] and SLIC [29]. While these methods are pre-segmentation algorithms and require additional processing to produce full segmentation, they represent the only category of methods which are comparable in execution speed to the presented algorithm. We have also chosen these segmentation algorithms since they produce visual results which are comparable to our algorithm. Sparse ray-traced segmentation could also be used as a pre-segmenter, but it would produce super-pixel sized segments only in very textured areas and large segments in homogenous areas.

From a speed standpoint, sparse ray-traced segmentation has an $O(N/tsize)$ complexity which makes it much faster than any full GPU segmentation algorithm, the fastest being $O(N)$.

TABLE 1. Speed results. This image compares various execution speeds. all measurements (besides image size) are in milliseconds. The highlighted columns show the total execution speed for sparse ray-traced segmentation with one seed per 4 × 4 pixels (4 × 4) and one seed per 16 × 16 pixels (16 × 16). Both timings are clearly superior to the state-of-the-art results, SLIC without connectivity, SLIC with connectivity [29] and really quick shift [13].

size (pixels)	flux	sparse net (4x4)	opt. complete labeling (4x4)	opt. merging (4x4)	FULL (4x4)	SPARSE (4x4)	sparse net (16x16)	opt. complete labeling (16x16)	opt. merging (16x16)	FULL (16x16)	SPARSE (16x16)	SLIC without connectivity	SLIC with connectivity	Really Quick Shift
256^2	0.182	0.523	0.505	0.803	2.013	0.705	0.172	4.682	2.323	7.359	0.354	2.183	5.001	5.075
512^2	0.659	1.955	1.853	2.415	6.882	2.614	0.998	4.772	1.785	8.204	1.647	5.135	20.175	21.788
758^2	0.909	3.551	5.531	3.689	13.68	4.46	1.293	10.741	2.693	15.636	2.202	13.38	36.159	62.422
1024^2	1.677	7.519	4.854	5.677	19.727	9.196	2.329	11.932	4.153	20.091	4.006	18.876	62.148	123.643
1536^2	3.731	17.102	10.752	12.522	44.107	20.833	4.887	26.001	9.033	43.642	8.618	53.188	123.268	307.163
2048^2	6.583	31.083	18.501	22.166	78.333	37.666	8.215	46.027	15.756	76.581	14.798	71.998	196.794	524.439
3072^2	15.001	73.141	41.004	49.809	178.955	88.142	20.777	120.555	35.111	191.444	35.778	203.093	481.791	1272.256
4096^2	26.375	134.754	68.761	87.251	317.141	161.129	37.115	215.502	62.437	341.429	63.49	292.329	770.729	2055.832
6144^2	59.714	307.857	164.714	193.002	725.287	367.571	85.723	486.909	140.363	772.709	145.437	762.215	1842.969	3673.915
8192^2	106.051	437.751	416.503	341.875	1302.18	543.802	135.447	897.222	267.256	1405.976	241.498	1151.41	2990.072	6941.628

TABLE 2. Ratios between computing times for sparse ray-traced segmentation with one seed per 4 × 4 pixels (4 × 4), one seed per 16 × 16 pixels (16 × 16), full segmentation with our algorithm (4 × 4 and 16 × 16), SLIC with/without connectivity and really quick shift.

size (pixels)	Ratio SPARSE (16x16) / SPARSE (4x4)	Ratio FULL (16x16) / FULL (4x4)	Ratio SLIC without connect. / SPARSE (16x16)	Ratio SLIC with connect. / SPARSE (16x16)	Ratio Really Quick Shift / SPARSE (16x16)	Ratio SLIC without connect. / FULL (16x16)
256^2	1.99	0.27	6.17	14.13	14.34	0.30
512^2	1.59	0.84	3.12	12.25	13.23	0.63
758^2	2.03	0.87	6.08	16.42	28.35	0.86
1024^2	2.30	0.98	4.71	15.51	30.86	0.94
1536^2	2.42	1.01	6.17	14.30	35.64	1.22
2048^2	2.55	1.02	4.87	13.30	35.44	0.94
3072^2	2.46	0.93	5.68	13.47	35.56	1.06
4096^2	2.54	0.93	4.60	12.14	32.38	0.86
6144^2	2.53	0.94	5.24	12.67	25.26	0.99
8192^2	2.25	0.93	4.77	12.38	28.74	0.82

Results show that the proposed algorithm has good local quality results, as shown in Figure 9, and exceptional speed results, as illustrated in Table 1, making it ideal for real-time segmentation pipelines. Furthermore, the results demonstrate that the new method finds complete segmentations in less time than the state-of-the-art methods need to finish their local segmentations. Table 1 shows the computing times for the flux computation, the creation of the sparse net, the complete labeling step and the merging step, our full segmentation and our sparse segmentation, in comparison with the results of SLIC [29] and Really Quick Shift [13].

All measurements were made on an NVIDIA GTX 960M graphics card. Table 2 simplifies the performance comparison between our sparse segmentation and the other two methods, as well as the performance assessment of our algorithm for different seed densities. We chose the most important computing times from Table 1 (full segmentation, sparse segmentation, SLIC and Really Quick Shift) and compared them by computing several ratios, which can be observed in Table 2.

From the Ratio SLIC without connectivity/SPARSE (16 × 16) and the Ratio Really Quick Shift/SPARSE (16 × 16) we can conclude that our sparse ray-traced segmentation consistently has running times approximately 5× faster than SLIC and approximately 25× faster than Really Quick Shift.

The obtained results indicate that even the full variant of our segmentation method has a running time approximately equal to the computation of the fastest algorithm, SLIC without connectivity, which does not produce a complete segmentation, but only super-pixels (Table 2, Ratio SLIC without connectivity / Full (16 × 16)).

Table 1 and 2 also show that the presented algorithm has a performance degradation which is sub-linear in the number of pixels, making it ideal for large and very large images.

Even if the algorithm is highly parallel, the overall complexity of the sparse ray-traced segmentation method is influenced by the number of threads, since the connection of seeds, described in Pseudocode 2, assumes that all the threads access the same array when updating the parent seed id. However, due to the spatial distribution of the rays, there are few cases when the atomicExchange() function delays the threads writing operations. Therefore, from the Ratio SPARSE (16 × 16)/ SPARSE (4 × 4) column in Table 2 we can observe that 16× more seeds only double the computing time. However, from the Ratio FULL (16×16) – FULL (4×4) we can conclude that the full segmentation is not affected by the number of seeds, since the gain obtained by the sparse segmentation with one seed per 16 × 16 pixels over the sparse segmentation with one seed per 4 × 4 pixels is lost by the additional time spent for the complete labeling and the merging steps.

Figure 9 shows the qualitative results of sparse ray-traced segmentation, where different types of images are segmented and compared to the state-of-the-art results.

The qualitative testing includes both sparse and complete labeling variants of the presented algorithm, as well as different tile sizes, which directly affect the number of seeds. Therefore, the fourth column in Fig. 9 might require zooming, as it has results obtained with a very low sparsity (seed density) value.

Sparse ray-traced segmentation shows comparable perception-based quality to the state-of-the-art methods. Furthermore, the presented algorithm outputs complete segmentations, while both SLIC and Really Quick Shift need further region merging processing due to their over-segmentation strategies.

FIGURE 9. Qualitative results: test image (1st column), sparse ray-traced segmentation with one seed each 16 × 16 pixels (sparse – 2nd column – and complete – 3rd column), sparse ray-traced segmentation with one seed each 4 × 4 pixels (sparse – 4th column and complete – 5th column), Really Quick Shift [13] (6th column), SLIC [29] (7th column).

In the first and third rows from Fig. 9, the result of the segmentation with Really Quick Shift and SLIC is almost identical to the initial image, meaning that the algorithms produce an extremely large number of very small regions. Comparatively, our method produces a small number of homogenous regions, that can be observed in the third and fifth columns.

On the fourth row from Figure 9, we can observe (also illustrated in the upper part of Figure 10) that, unlike sparse ray tracing segmentation, Really Quick Shift can produce false regions. On the fourth and fifth rows from Figure 9 and in Figure 10 we can also observe that besides over-segmentation, the Really Quick Shift algorithm produces regions that cover two separate objects (under-segmentation). Our algorithm can also lead to over-segmentation in some areas, due to light reflection or textured surfaces, as can be observed in the lower part of Figure 10, but it separates correctly the surfaces of different objects (it does not lead to under-segmentation).

Sparse ray-traced segmentation is a flexible algorithm, where the tradeoff between quality and speed is controlled by a single sparsity parameter. It can even be used as an enhanced pre-segmentation method, as shown in Figure 11. The resulting pre-segmentation preserves a large part of the local structure but still completely discovers and links the large segments, in comparison to the state-of-the-art pre-segmentation methods which partition such segments.

One limitation of our approach is that the flux required as input by the presented method can be parameter heavy, especially if a non-adaptive solution is used.

Because the algorithm sparsely explores space, it is especially prone to noise related errors, thus the flux needs to be conservative. Another limitation is that the algorithm was designed for usage in common scenarios (video tracking, important features detection, real-time usage) and is not the best choice for specialized segmentation cases where quality is much more important than speed, for example OCRs.

As previously mentioned, sparse ray-traced segmentation is already used in practical applications in the Sound of Vision project [34], where it segments RGBD input in real-time, with the help of an adaptive exponential flux detector.

Because of the very small computational cost of the segmentation, it can be used in a long pipeline, together with

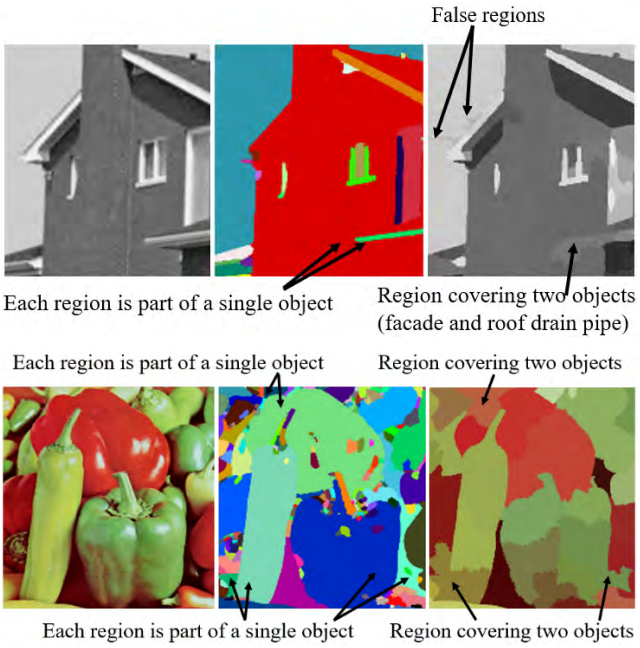

FIGURE 10. Problems of really quick shift (right), that do not appear in our segmentation (middle): Under-segmentation and false regions.

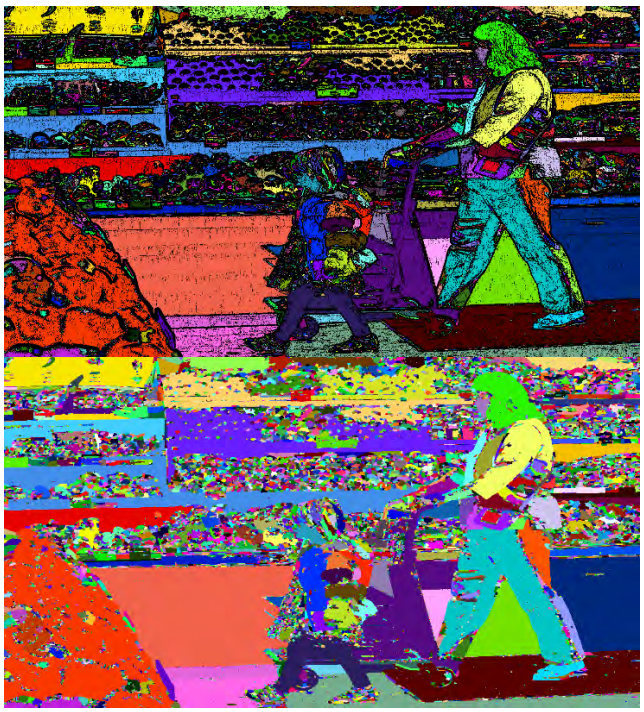

FIGURE 11. Results of running the ray-traced segmentation on a complex image with both small and large regions.

signal filtering, normal computation, best free space analysis, feature tracking, object labeling and sonification [35].

V. CONCLUSION

A GPU sparse segmentation was introduced in this article, which explores the search space with ray tracing. This strategy enables clustering at a high distance in feature space, while using the small number of searching queries specific to local cluster-size aware searches, such as those applied by pre-segmenters.

Both quality and speed results show that the presented method excels in real-time segmentation scenarios, and it has also proven its value in the Sound of Vision [34] project.

The computational load of the algorithm is handled through a ray tracing framework, which makes it easy to be efficiently implemented on GPUs, while also benefitting from numerous ray tracing specific optimizations. Because of this, the presented method can sparsely segment datasets on the GPU approximately 5× faster than the state-of-the-art pre-segmentation methods, without even considering the additional region merging costs incurred by the latter methods.

The algorithm even runs in real-time on a 7-year-old NVIDIA 460MX graphics card. Furthermore, the algorithm is platform independent, being completely implementable in OpenGL 4.3.

The presented algorithm can trade off quality and speed with a single parameter, the segmentation sparsity. The algorithm runs in real-time even at very low sparsity values, where the majority of the set elements are labeled.

Optional full labeling and region merging methods are also provided, which completely label the entire set.

ACKNOWLEDGMENT

A patent application based on this work was submitted to OSIM (Romanian patent organization), registration no. A/01176 from 29.12.2017.

REFERENCES

[1] A. Hagan and Y. Zhao, "Parallel 3D image segmentation of large data sets on a GPU cluster," in *Proc. Int. Symp. Vis. Comput.*, 2009, pp. 960–969.

[2] Z. Li and J. Chen, "Superpixel segmentation using linear spectral clustering," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 1356–1363.

[3] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient K-means clustering algorithm: Analysis and implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002.

[4] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proc. 18th Annu. ACM-SIAM Symp. Discrete Algorithms (SODA)*, 2007, pp. 1027–1035.

[5] S. Schenke, B. Wünsche, and J. Denzler, "GPU-based volume segmentation, " in *Proc. IVCNZ*, 2005, pp. 171–176.

[6] Y. Beevi and S. Natarajan, "An efficient video segmentation algorithm with real time adaptive threshold technique," *Int. J. Signal Process., Image Process. Pattern Recognit.*, vol. 2, no. 4, pp. 13–27, 2009.

[7] L. Li, J. Yao, J. Tu, X. Lu, K. Li, and Y. Liu, "Edge-based split-and-merge superpixel segmentation," in *Proc. IEEE Int. Conf. Inf. Automat.*, Aug. 2015, pp. 970–975.

[8] P. F. Felzenszwalb and D. P. Huttenlocher, "Efficient graph-based image segmentation," *Int. J. Comput. Vis.*, vol. 59, no. 2, pp. 167–181, Sep. 2004.

[9] J. Shi and J. Malik, "Normalized cuts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 8, pp. 888–905, 2000.

[10] V. Vineet and P. J. Narayanan, "CUDA cuts: Fast graph cuts on the GPU," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Workshops*, Jun. 2008, pp. 1–8.

[11] H. Zhu, F. Meng, J. Cai, and S. Lu, "Beyond pixels: A comprehensive survey from bottom-up to semantic image segmentation and cosegmentation," *J. Vis. Commun. Image Represent.*, vol. 34, pp. 12–27, Jan. 2016.

[12] D. Comaniciu and P. Meer, "Mean shift: A robust approach toward feature space analysis," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 5, pp. 603–619, May 2002.
[13] B. Fulkerson and S. Soatto, "Really Quick shift: Image segmentation on a GPU," in *Proc. 11th Eur. Conf. Trends Topics Comput. Vis. (ECCV)*, 2010, pp. 350–358.
[14] S. Paris, "Edge-preserving smoothing and mean-shift segmentation of video streams," in *Proc. 10th Eur. Conf. Comput. Vis. (ECCV)*, 2008, pp. 460–473.
[15] P. Soille, "Constrained connectivity for hierarchical image partitioning and simplification," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 7, pp. 1132–1145, Jul. 2008.
[16] R. G. Cinbis, J. Verbeek, and C. Schmid, "Multi-fold mil training for weakly supervised object localization," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 2409–2416.
[17] T. F. Chan and L. A. Vese, "Active contours without edges," *IEEE Trans. Image Process.*, vol. 10, no. 2, pp. 266–277, Feb. 2001.
[18] A. Morar, F. Moldoveanu, and E. Gröller, "Image segmentation based on active contours without edges," in *Proc. IEEE 8th Int. Conf. Intell. Comput. Commun. Process.*, Aug./Sep. 2012, pp. 213–220.
[19] A. Mustafa and A. Hilton, "Semantically coherent co-segmentation and reconstruction of dynamic scenes," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 422–431.
[20] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2015, pp. 3431–3440.
[21] G. Bertasius, L. Torresani, S. X. Yu, and J. Shi, "Convolutional random walk networks for semantic image segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 858–866.
[22] C. Lea, M. D. Flynn, R. Vidal, A. Reiter, and G. D. Hager, "Temporal convolutional networks for action segmentation and detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 156–165.
[23] A. Abramov, T. Kulvicius, F. Wörgötter, and B. Dellen, "Real-time image segmentation on a GPU," in *Facing the Multicore-Challenge*, 2010, pp. 131–142.
[24] M. Roberts, J. Packer, M. C. Sousa, and J. Mitchell, "A work-efficient GPU algorithm for level set segmentation," in *Proc. Conf. High Perform. Graph.*, 2010, pp. 123–132.
[25] M. D. Collins, J. Xu, L. Grady, and V. Singh, "Random walks based multi-image segmentation: Quasiconvexity results and gpu-based solutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2012, pp. 1656–1663.
[26] G. B. Vitor, A. Körbes, R. de Alencar Lotufo, and J. V. Ferreira, "Analysis of a step-based watershed algorithm using CUDA," *Int. J. Natural Comput. Res.*, pp. 16–28, 2010.
[27] E. Ramírez, P. Temoche, and R. Carmona, "A volume segmentation approach based on GrabCut," *CLEI Electron. J.*, vol. 16, no. 2, pp. 1–14, 2013.
[28] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk, "SLIC superpixels," School Comput. Commun. Sci., Ecole Polytech. Fédrale Lausanne, Lausanne, Switzerland, Tech. Rep. 149300, 2010.
[29] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk, "SLIC superpixels compared to state-of-the-art superpixel methods," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 34, no. 11, pp. 2274–2282, Nov. 2012.
[30] A. Levinshtein, A. Stere, K. N. Kutulakos, D. J. Fleet, S. J. Dickinson, and K. Siddiqi, "TurboPixels: Fast superpixels using geometric flows," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 12, pp. 2290–2297, Dec. 2009.
[31] B. Fulkerson, A. Vedaldi, and S. Soatto, "Class segmentation and object localization with superpixel neighborhoods," in *Proc. IEEE 12th Int. Conf. Comput. Vis.*, Sep./Oct. 2009, pp. 670–677.
[32] S. Boulos *et al.*, "Packet-based whitted and distribution ray tracing," in *Proc. Graph. Interface*, 2007, pp. 177–184.
[33] M. McGuire and M. Mara, "Efficient GPU screen-space ray tracing," *J. Comput. Graph. Techn.*, vol. 3, no. 4, pp. 73–85, 2014.
[34] *Sound of Vision Project*. Accessed: Jul. 14, 2018. [Online]. Available: https://soundofvision.net/
[35] S. Caraiman *et al.*, "Computer vision for the visually impaired: The sound of vision system," in *Proc. IEEE Int. Conf. Comput. Vis. Workshops (ICCVW)*, Oct. 2017, pp. 1480–1489.
[36] A. Morar, F. Moldoveanu, L. Petrescu, O. Balan, and A. Moldoveanu, "Time-consistent segmentation of indoor depth video frames," in *Proc. 40th Int. Conf. Telecommun. Signal Process. (TSP)*, Jul. 2017, pp. 674–677.
[37] *Conservative Rasterization*. Accessed: Jul. 14, 2018. [Online]. Available: https://docs.nvidia.com/gameworks/content/gameworkslibrary/graphicssamples/opengl_samples/conservativerasterizationsample.htm
[38] A. Morar, F. Moldoveanu, L. Petrescu, and A. Moldoveanu, "Real time indoor 3D pipeline for an advanced sensory substitution device," in *Image Analysis and Processing-ICIAP* (Lecture Notes in Computer Science), vol. 10485, S. Battiato, G. Gallo, R. Schettini, and F. Stanco, Eds. Cham, Switzerland: Springer, 2017, pp. 685–695.

LUCIAN PETRESCU received the B.S. degree in computer science, the M.S. degree in computer graphics, virtual reality, and multimedia, and the Ph.D. degree from the Politehnica University of Bucharest, in 2010, 2012 and 2015, respectively. During his Ph.D. degree, he studied the possibility of rendering massive 3D scenes in real-time. Since 2015, he collaborated with the academia and the industry, involved in projects related to computer vision and computer graphics.

ANCA MORAR received the B.S. degree in computer science from the Politehnica University of Bucharest, in 2009, and the Ph.D. degree in computer science, in 2012, in the field of medical image analysis and visualization. She is currently an Associate Professor with the Computer Science and Engineering Department, Faculty of Automatic Control and Computers, Politehnica University of Bucharest. Her research is focused on computer graphics, GPGPU, computer vision, and e-health.

FLORICA MOLDOVEANU is currently a Professor with the Department of Computer Science and Engineering with the Politehnica University of Bucharest. She coordinates the master program computer graphics, multimedia and virtual reality at the Faculty of Automatic Control and Computers. She is also the President of the Health Level 7 Romania Association. Her research and teaching activity is focused on computer graphics, computer vision, software engineering, and e-health.

ALIN MOLDOVEANU is currently a Full Professor with the Computer Science and Engineering Department, Faculty of Automatic Control and Computers (http://acs.pub.ro), Politehnica University of Bucharest, where he teaches software engineering and virtual reality. His active research areas include virtual and augmented reality (exploring and applying immersion, sensory substitution, and distorted reality), e-Health (assistive and rehabilitative solutions and prevention of hospital acquired infections), and e-Learning (3D MMO mixed-reality campuses). He is also the Director or responsible for national or European research projects in these areas, such as Sound of Vision, TRAVEE, and HAI-OPS.