EG UK Computer Graphics & Visual Computing (2021)
K. Xu and M. Turner (Editors)

RECCS : Real-Time Camera Control for Particle Systems

M. Köster[1], J. Groß[1] and A. Krüger[1]

[1] German Research Center for Artificial Intelligence (DFKI)
Saarland Informatics Campus
Saarbrücken, Germany

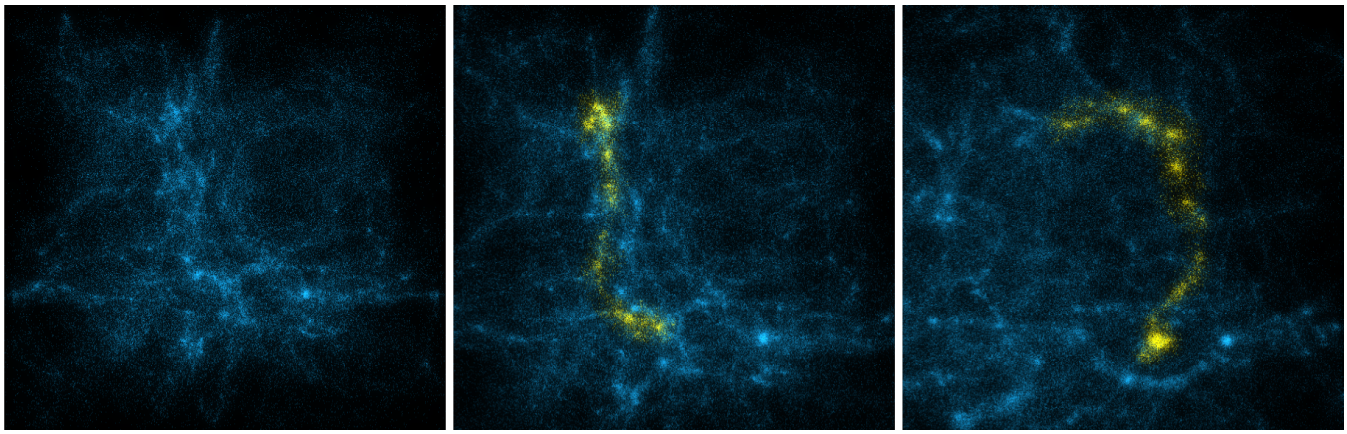

Figure 1: *The left image shows an overview over a subset of the Millenium 2 dataset consisting of* 442000 *particles.* 6100 *particles are selected with the help an existing particle-set selection method using the same camera perspective (center image). However, this perspective does not reveal the actual 3D shape of the selection. The right image displays our automatically determined camera perspective that was computed in* ≈ 28*ms using a developed evaluation heuristic (see Section 6).*

Abstract

Interactive exploration and analysis of large 3D particle systems, consisting of hundreds of thousands of particles, are common tasks in the field of scientific and information visualization. These steps typically involve selection and camera-update operations in order to reveal patterns or to focus on subsets. Moreover, if a certain region is known to be potentially interesting or a selection has been made, the user will have to choose a proper camera setup to investigate further. However, such a setup is typically chosen in a way that the interesting region is in the center of the screen. Unfortunately, it still needs to show important characteristics of the selected subset and has the least amount of occlusions with respect to other particles but shows enough context information in terms of non-selected particles. In this paper, we propose a novel method for real-time camera control in 3D particle systems that fulfills these requirements. It is based on a user and/or domain-specific evaluation heuristic and parallel optimization algorithm that is designed for Graphics-Processing Units (GPUs). In addition, our approach takes only several milliseconds to complete, even on the aforementioned large datasets while consuming only a few megabytes in global GPU memory in general. This is due the fact that we are able to reduce the processing complexity significantly using screen-space information and the excessive use of fast on-chip shared memory. This allows it to be seamlessly integrated into modern visualization systems that need real-time feedback while processing large particle-based datasets.

CCS Concepts

• ***Human-centered computing*** → *Visualization systems and tools; Scientific visualization; Interaction techniques;* • ***Computing methodologies*** → *Shared memory algorithms; Massively parallel algorithms; Graphics processors;*

DOI: 10.2312/cgvc.20211320

1. Introduction

Data analysis is an omnipresent challenge in everyday scientific and information visualization. Important tasks in this scope are often data selection operations that help differentiating possibly interesting subsets from other regions [Tuk77; ONI05]. This specifically applies to the domain of scientific visualization and large unstructured particle-based data sets [KK16] which might be the immediate outcome of particle simulations [MCG03; Kel06; MMCK14; KK16]. There have been many newly invented algorithms in this context that provide helpful methodologies and tools to realize particle-selection operations in an efficient way [HWVF12; YEII12; YEII16; KK16]. However, these papers do not cover further camera-control related operations that might be needed to get a deeper understanding of a selected sub volume.

Camera-control concepts in general deal with the problem of adjusting a certain camera position/orientation setup to get a *better* view onto a single or multiple objects of interest [Hai09]. Analytical methods work well in the scope of a few objects, around which a camera needs to be positioned [LST04]. This is particularly useful for minimizing occlusions in computer games or other interactive computer graphics applications [CO09]. In our domain, we deal with thousands of occluding objects that cannot be handled directly by analytical equations. Even if we consider a semantic distinction between particle clusters, this does not apply to arbitrary datasets where cluster structures cannot be easily determined (see Figure 1). Moreover, a major challenge related to automated camera control are the degrees of freedom. In theory, we can position the virtual camera at any point in the 3D space and adjust its look-at point freely. However, in practice, it is important to limit the degrees of freedom in order to reduce the search space.

In this paper, we introduce a new approach to realize real-time camera control for particle systems called *RECCS*. We consider a particle selection step to be completed in the beginning, yielding a subset of selected particles. In terms of limiting the search space, we consider possible camera positions to be on a sphere constructed around a bounding box including all selected particles. Additionally, we restrict the camera orientation to look at the center of the sphere. This dramatically reduces the number of possibilities to consider and is inspired by arcball cameras which rotate around a centered object in 3D space [Sho92; ZSS11].

Our idea is based on decoupling the dataset from the actual processing pipeline by reducing the amount of considered particles in the first step. It is inspired by the approach of *Screen Space Particle Selection* created by Köster et al. [KK18], who used screen-space operations to separate the complexity of the dataset and the actual selection operations they perform. In contrast to purely screen-space-based operations, we use cube maps [PF05] to detect visible particles from all six cube-map-face perspectives. Those particles then form the reduced dataset on which our further analyses work on (Section 3.1). Afterwards, we evaluate a set of uniformly distributed and randomly chosen points on the initially determined sphere to get the "best" perspective (Section 3.2). Here, best refers to the minimization of a cost function given by the user, which is a fundamental input to our method, similar to other prominent heuristic optimization approaches [MLBT11; KGK19a]. This domain-specific function enables us to differentiate between "good" and "bad" camera setups (see also Section 6).

We follow the general approach of particle swarm optimization [EEB07], which works by parallelizing over a large set of possible solution candidates. Similarly, we evaluate a large set of camera-perspective candidates in a massively parallel way motivated by recent work presented in [KGK19a; KGK19b]. However, we do not perform an iterative optimization step, as we evaluate all potentially interesting candidates at the same time. In order to reduce the memory consumption and improve the runtime performance, we use fast on-chip *shared memory* to cache temporary evaluation results in analogy to work by Groß et al. [GKK19; GKK20] (Section 3.2, Section 4). This allows us to complete a full camera-control step in a few milliseconds which only requires a few megabytes of auxiliary space (Section 6).

2. Related Work

Camera control in general is a well known topic covering many years of research. Therefore, this chapter provides an overview over selected papers that involve different fields of applications. To begin with, a well known application domain of camera-control methods are cinematography systems. For instance, the work by Kneafsey et al. [KM05] focusses on using virtual cameras attached to NPCs in computer games. Driven by a high-level cinematography module, switches between those cameras are possible on-the-fly. This approach implicitly limits the search space by considering camera perspectives to already known candidate locations and directions. A follow up work [MK06] provides the opportunity to improve the engagement in computer games to attach the virtual camera to different game objects.

As mentioned in the introduction, there have been different approaches using analytical equations to formulate optimization problems. A prominent paper in this area is the work by Bodor et al. [BSP05]. They define and solve a general formulation for multi-camera-control problems to optimize the placement of cameras in terms of improved observability regarding specific tasks in their setting. However, extending this approach to be suitable for large-scale particle systems is not feasible solution.

There has also been work in the field of minimizing occlusions with respect to camera control [CON08]. In this paper, they used a depth-buffer based visibility analysis to be able to compute occlusions for a large number of objects. The general idea is related to our method, since we also use kind of screen-space projections. However, our method does not primarily focus on minimizing occlusions by design and on differentiating between specific objects in scene. Instead, the domain-specific heuristic has to decide whether some regions should not be occluded. Furthermore, they construct visibility volumes to minimize the occlusions by sampling to support even multiple objects, whereas we rely on evaluating completely distinct camera perspectives in a massively parallel way.

As outlined in the introduction, we rely on a user-provided evaluation heuristic. This is also performed by many related papers focusing on camera control. For instance, Yannakakis et al. [YMJ10] presented a detailed user study to determine appealing camera setups for users of a marble game. They used the gathered results to train a neural network in order to perform the intended camera adjustments. Such a method can also be applied to our generic approach, which allows the use of arbitrary heuristic functions.

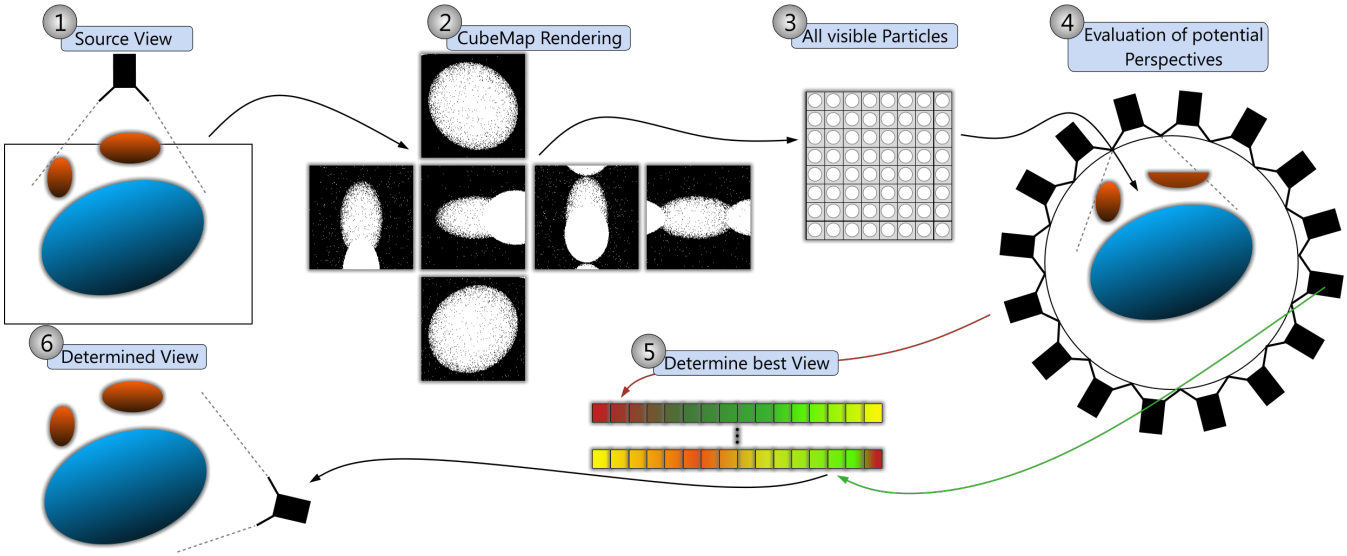

Figure 2: *High-level workflow of RECCS. Initially, we assume an arbitrary camera position looking at some particles of the dataset (1). We derive a cube map from a bounding box including all selected particles (2) to determine a set of all visible particles (without duplicates) (3). Next, we evaluate a specified number of different and uniformly distributed random camera orientations that look at the center of the derived bounding box (4). After evaluating all perspectives in parallel, we fetch all evaluation results back to the CPU and determine the best view according to a custom comparison function (5). Finally, we adjust the camera position and orientation based on the evaluation results (6).*

In analogy to the previously discussed methods, Viola et al. [VFSG06] introduce a framework for optimal estimation of viewpoints in the context of volumetric datasets. The framework they developed is capable of finding important focal points by evaluating visibility estimation metrics. As stated in the paper, it is well suited for datasets that can be segmented into multiple volumetric regions. Compared to our method, this approach focuses on multiple discrete objects that can be semantically distinguished from each other, while we reason about large sets of loosely coupled particles.

In the context of heuristically optimizing virtual/physical camera setups, a highly related paper is the one by Yi-Chun et al. [YBH11]. They use particle-swarm optimization to improve the coverage of a camera network by adjusting the field of views of all cameras. However, they do not consider moving cameras in their setting and they do not provide a domain-specific solution for particle systems. In our field of application, we need to specifically reason about huge sets of particles and their shape when they are rendered for visualization purposes. To the best of our knowledge, these methods are the ones mostly related in terms of conceptual similarities.

3. RECCS

The main processing workflow ouf our method is presented in Figure 2. As shown in this figure, we start with a particle-based dataset (step 1) and a arbitrary camera setup used for visualization purposes. Furthermore, we already assume that a particle selection step has been applied to the dataset resulting in a set of so called *target* particles (highlighted in blue). In contrast to all other particles (highlighted in red), these are the ones considered to be interesting for the end user. Therefore, the task is to find a camera perspective that focusses on the target particles while minimizing occlusions but maximizing context information about other particle-structure formations in the surrounding.

We further assume the existence of and the access to a "reasonable" bounding box including all target particles in the 3D world space. Reasonable in this context means that the distance from each side of this bounding box can be directly used to position a camera which is still able to cover all interesting properties of the selected subset (black bounding box in step 1). This bounding box is an essential input for out method since we limit the solution space to camera positions that lie on a sphere including this bounding box. We also limit the camera orientation to look at the center of this sphere rather than evaluating other possibilities. Evaluation of further degrees of freedom remains as future work (see also Section 7).

In order to solve this still highly challenging task, we perform several processing steps sequentially. This involves preparing a particle subset (see Section 3.1) in steps 2 and 3 for further processing, in order to reduce the amount of data required. Step 4 represents the actual optimization phase to determine individual rankings for all potentially interesting camera perspectives. Thereby, the perspective candidates are computed with the help of uniformly distributed points on a sphere that spans over the input bounding box. Custom domain-specific constraints, like the previously mentioned ones regarding occlusions and context information, are realized with the help of user-defined cost/evaluation functions that are plugged into the processing workflow in steps 4 and 5. Note that all computationally-expensive steps (2–4 in Figure 2) are completely executed on the GPU and do not involve any CPU to GPU data transfer (and vice versa). Steps 5 and 6 are executed on the CPU, which allows us to use the determined optimization result to adjust the camera position on the application side.

3.1. Subset Computation

An important step to evaluate potential camera perspectives is the reduction of the dataset complexity. Consider a random input dataset about which we do not have any information in the beginning. The number of particles and/or their relation to each other might be extremely complex. This would cause an algorithm working on the underlying structures of the dataset directly to be highly affected in terms of memory consumption and runtime. In order to decouple further processing steps from the actual dataset in terms of runtime and amount of data, we first compute a *potentially interesting* particle subset. This subset contains all particles that are visible from all faces of a cube map spanning over a bounding box containing all target particles.

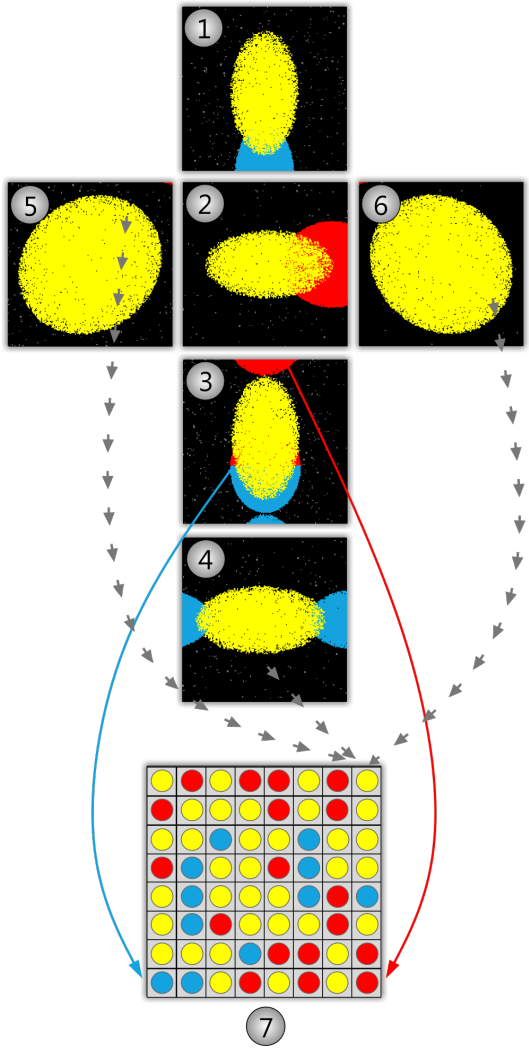

Figure 3: *Computation of all visible particles without duplicates from the cube map faces 1–6 (top). The resulting set of particles (bottom, 7) is smaller or equal compared to the cube map regarding the number of particles.*

Figure 3 visualizes the actually performed operations. First, we render a cube map from all sides of the input bounding box. The cube map contains 32bit (*I32*) values representing the actual particle *Id*s at these positions in 2D space. This allows us to retrieve further information about all particles later on. Furthermore, each particle will be mapped to a *single pixel* in this step only. This allows us to store more particles in each cube map face, which increases the effective resolution of the cube map being used (e.g. in contrast to rendered quads). Second, we determine a set of all particles (without duplicates) that have been visible from all sides of the cube map.

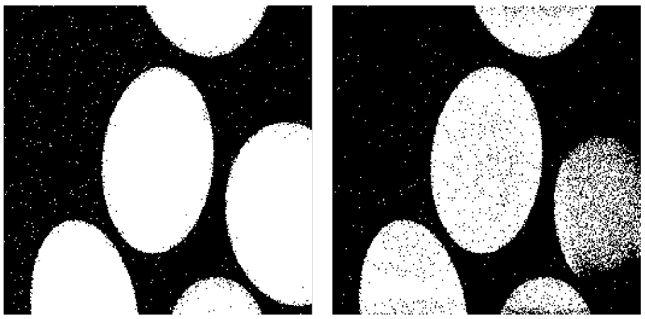

Figure 4: *An intermediate camera perspective rendered with the original dataset (left) and the reduced dataset we use for processing (right, step 3 in Figure 2). Since the reduced set contains particles that were visible from the different cube-map perspectives only (see Figure 3), there are slight deviations compared to the visualization of the original dataset.*

In all upcoming steps, we render the particles from this reduced subset only instead of the whole dataset. This idea is actually based on the concept of screen-space re-projection [Ngu07], which simulates slightly different camera perspectives while using an already rendered image with additional depth information. In our processing pipeline, we do not actually re-project particles from a screen-space buffer as we still have access to the actual particle positions from the dataset via the stored *Id* information from the cube map, and thus, the reduced dataset.

Although this approach has major benefits in terms of runtime performance, it also reduces the quality of all camera perspective to be evaluated in the next step (see Section 3.2). Figure 4 visualizes possible deviations when using the reduced dataset in favor of all particles. Similar to related approaches using screen-space re-projection, such artifacts cannot be avoided in general. However, we argue that this is not a significant limitation of our internal functionality, since all camera perspectives will ultimately have some artifacts. Finally, the actually determined camera perspective mainly relies on the used evaluation function that can also further compensate these artificially introduced precision loss.

3.2. Evaluation of Potential Camera Perspectives

The heart of our approach is the efficient evaluation of potential camera perspectives (see also Section 4). As mentioned in the introduction, this step is heavily inspired by the general concept of particle-swarm optimization [EEB07]. The most important difference is that we do not require an iterative processing of candidate perspectives. Instead, our idea is to evaluate all camera-setup candidates in parallel in the scope of a single evaluation kernel on-the-fly. In order to evaluate each camera perspective, we must concep-

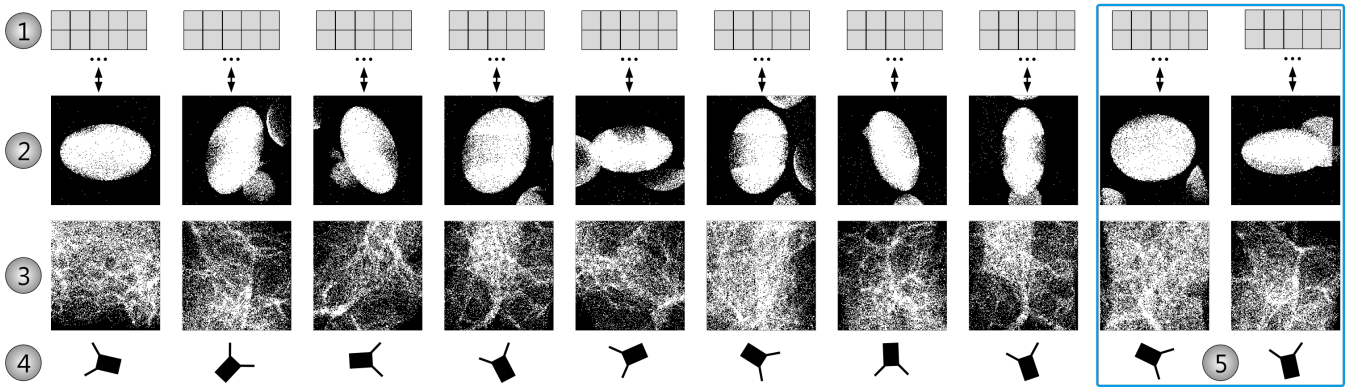

Figure 5: *Visualization of the parallel camera-perspective evaluation phase using chunks of shared memory (top, 1). Rows 2 and 3 show sample intermediate images that will be constructed during this phase based on two different evaluation scenarios (middle). Conceptually different camera perspectives used to render the images into shared memory are shown at the bottom (4). From an implementation point of view, we usually assign the rendering of two camera perspectives to the same multiprocessor to improve efficiency (blue border, 5).*

tually render all particles from each perspective into a back buffer in the first place. This is required to compute the closest "hit" (particle) for each pixel from a given camera orientation. We refer to a single image rendered for evaluation purposes as an *intermediate image*.

A straight-forward realization would be to allocate as many back buffers as possible in GPU memory and invoke a rendering pipeline interface to call the available hardware rasterization units. Although this seems to be good choice in the beginning, using a larger number of potential perspective candidates causes the amount of required memory to explode. Alternatively, we have to perform many render passes and fetch the results back into CPU memory after each iteration. This causes significant runtime and data-transfer overhead.

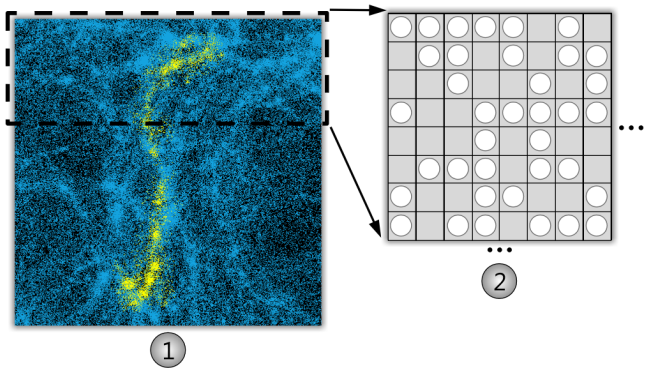

Figure 6: *Visualization of intermediate image slice (1) stored in shared memory (2). Each slice will be rendered one after another.*

A much cleverer option is the use of a single massively-parallel GPU kernel to evaluate all perspectives in parallel (see Figure 5). Furthermore, we can leverage shared memory to store chunks of the intermediate images directly on the chip. As presented in Figure 6, it usually happens (see Section 5 and Section 6) that the amount of shared memory on a GPU is too small to store the back buffer completely. Therefore, we virtually split the intermediate images into chunks, where each chunk is as large as possible while still fitting into shared memory. For each slice, we iterate over all particles in the subset from step 3 (see Section 3.1) and try to "render" them into our current slice. Rendering here refers to a software-based screen-space projection method that performs the world-view-projection-matrix multiplication within the kernel rather than using a hardware circuit. Since we map each particle to a single pixel in an intermediate image only, the runtime benefit of using specific rasterization support will be negligible.

3.3. Decision Making

The final steps involves fetching of the evaluation results of all intermediate images. After loading all results into CPU (accessible) memory, they are sorted in order to determine the winning camera perspective. Based on our experience, there is no need to perform this operation on the GPU as we have never exceeded more than 10240 candidates in practice. The determined perspective is then converted into a 3D world position of the camera and its look-at target position, which points to the center of the input bounding box. Note that the distance to the selected target particles is fixed, as we only consider potential camera perspectives on a single sphere in this paper.

4. Algorithm

The main camera-perspective evaluation algorithm described in Section 3.2 is shown in Algorithm 1. It is designed as a GPU kernel where one thread group processes one camera perspective (see Figure 5). Note that it is particularly optimized to minimize thread divergences, and maximize L1 and L2 cache reuse [KLH*14; KGK20; LSG19].

The required inputs are a buffer containing all particle positions and kinds (whether a particle is a target particle or not), a buffer containing all world-view-projection matrices for each camera perspective and a result buffer to store the evaluation results. We rely on dynamically specified shared memory (line 1) to adjust the amount of shared memory depending of the back-buffer format (*TData*) and the size of a single slice. After getting a pointer to shared memory, we initialize the heuristic evaluator given by

the developer (*TEvaluator*, line 2). We then iterate over all slices (lines 3–26) and initialize the virtual back buffer in shared memory in each iteration (lines 4–7). Next, we perform the actual rendering step by projecting each particle position into the 2D screen space plus additional depth information (lines 8–19). If the projected position falls into the range of the current slice, we pack the screen-space information with the current particle index into the back buffer (see also Section 5, lines 13–16). The insertion into the back buffer data structure in shared memory is realized with the help of an *atomicMin* operation to prefer particles that are closer to the camera (depth-min test in software, lines 14–16). After projecting all particles into the current slice in shared memory, we call the evaluator on each "pixel" that contains a valid entry (lines 20–27). This includes unpacking of the stored depth and particle id and invoking the user-defined evaluator on the current pixel (x, y), depth and particle id tuple.

When all slices are evaluated we aggregate the information gathered by all threads of the current group into the first thread (lines 29–33). This can be achieved by warp-wide combined with group-wide value reductions to improve efficiency [NVI14; NVI21]. If the current thread is the first thread, we store the aggregated evaluation result in the result buffer and the processing of the current camera perspective is finished (line 32).

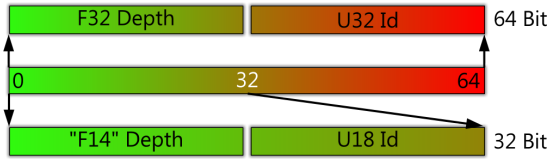

Figure 7: *Visualization of two possible back-buffer formats used within our optimization system. The depth values of each particle can either be stored as* FP32 *or* FP14 *whereas the particle* Id *can either be* 32 *or* 18*bits.*

5. Implementation Details

We implemented our approach in C# and used the ILGPU[†] JIT compiler for all GPU-based computations. As already described in Section 3, we used a pure software-based screen-space projection method to virtually render all particles as points into the back buffers. This gives us more flexibility with respect to the whole optimization pipeline since we do not need to have access to hardware rasterization units. Moreover, using hardware rasterization would require a different implementation which makes it difficult to leverage purely shared-memory based back buffers.

In terms of back-buffer size in shared memory, we are always limited by the current state-of-the-art GPU capabilities. For instance, the NVIDIA RTX 3090 (used in our evaluation, see Section 6) provides $\approx$ 100KB shared memory [NVI21], which should be separated into two parts in order to achieve full thread occupancy. This yields a total amount of $\approx$ 50KB for each thread group, and thus, for each camera-perspective evaluation step. Using a back buffer consisting of 32bit float (*FP32*) and 32bit particle *Id* values (see Figure 7) results in $\approx$ 6400 pixels in shared memory. Reducing the depth-buffer precision to 14bit float values (*FP14*) and the particle *Id*s to 18bits, allows us two double the number of pixels in the back buffer. This significantly influences the overall runtime (see Section 6), since the number of rendering steps is halved in this scope.

In practice, we take the upper 14bits from the *FP32* mantissa while discarding sign and exponent bits, since all depth-buffer val-

Algorithm 1: Parallel evaluation algorithm of RECCS

```
   Input: particleBuffer, wvpMatrixBuffer, evalResultBuffer
   /* Initialize shared memory dynamically
      based on the group sizes and the size
      of each element TData                          */
1  sharedMemory := shared memory TData[...];
2  evaluator := new TEvaluator();
3  for s := 0; s < #slices; s++ do
       /* Clear back buffer                          */
4      for i := group index; i < len(sharedMemory); i++ do
5          sharedMemory[i] := maxValue(TData);
6      end
       /* Wait for all back-buffer elements
          to be cleared                              */
7      group barrier;
8      wvp := wvpMatrixBuffer[grid index];
       /* Iterate over all particles and
          render them into our buffer                */
9      for i := group index; i < len(particleBuffer); i++ do
10         position := particleBuffer[i];
11         pOnScreen = project(position, wvp);
12         if pOnScreen isInRangeOf(s, sharedMemory) then
13             backBufferVal := pack(pOnScreen, i);
14             atomicMin(
15                 sharedMemory[pOnScreen],
16                 backBufferVal]);
17         end
18     end
       /* Wait for all particles to be
          rendered                                   */
19     group barrier;
       /* Evaluate the current slice                 */
20     for i := group index; i < len(sharedMemory); i++ do
21         (depth, id) := unpack(sharedMemory[i]);
22         if id isValid then
23             (x, y) = reconstructFrom(i);
24             evaluator.Apply(x, y, depth, id);
25         end
26     end
       /* Wait for the evaluation results            */
27     group barrier;
28 end
   /* Aggregate results and propagate them
      to the first thread                            */
29 evalResult := evaluator.AggregateIntoFirstThread(
30     group index);
31 if IsFirstThreadOfGroup then
32     evalResultBuffer[grid index] := evalResult;
33 end
```

[†] www.ilgpu.net

ues are normalized to $[0,\dots,1]$. Also note that an *FP14*-based buffer only allows particle *Id*s up to $2^{18}-1 = 262143$ particles. It also imposes a limitation of the general intermediate image sizes which should not be larger than 209×209 pixels. This is due to the fact that an initial cube map of $209 \times 209 \times 6$ pixels can have up to 262086 unique particles in the worst case. However, in practice we limit ourselves to the resolution of 192×192 when using *F14* buffers in order to distribute them nicely across our launched thread grids.

6. Evaluation

The evaluation focuses on two aspects: runtime performance with respect to different back buffers and scalability with respect to different intermediate image resolutions. Figure 8 presents the different scenarios that were taken from already published work by Köster et al. [KK16], which are in turn also based on work by Yu et al. [YEII16]. All performance tests are executed on an NVIDIA GeForce RTX 3090. A performance measurement is the median execution time of 100 camera-perspective computations. Furthermore, all measurements are conducted using several particle-selection datasets with pre-selected particle subsets [KK16]. Moreover, we adjust the launch dimensions of the thread grid in a way to use at least two thread groups per multiprocessor in order to leverage shared memory and the available processing resources in a most efficient way. This also implies that a small number of camera perspectives does not yield highest occupancy, as not all multiprocessors will receive a reasonable amount of work. Therefore, we decided to use 10240 possible camera perspectives to achieve a reasonable occupancy and workload on our GPU.

In order to evaluate the camera-perspective computation properly, we designed an initial heuristic to determine the "best camera" perspective. The heuristic uses a cost function weighting visible particles according to their distance to the center of the image and their distance in screen space. To do so, it stores a 2D bounding box and a weighted sum of all depth values:

$$\mathrm{dist}(x,y) = \mathrm{dot}\left((x,y)^T, \left(\frac{w}{2},\frac{h}{2}\right)^T\right), \tag{1}$$

$$\mathrm{depth}(i,d) = \begin{cases} d & \text{if kind}(i) = \text{target} \\ 1-d & \text{else,} \end{cases} \tag{2}$$

$$\mathrm{weight}(i,x,y,d) = \mathrm{dist}(x,y) \cdot \mathrm{depth}(i,d), \tag{3}$$

where x, y and d refer to the current pixel (x,y) and its normalized depth value $d \in [0,\dots,1]$ in the back buffer. Thereby, w and h refer to the width and height of an intermediate image and i refers to the particle *Id* of the i-th particle that is stored at this screen-space position. Note that target particles are preferred if they are close to the camera, whereas non-target particles are preferred if they are in the background. The total weight W_t is then given by the sum over all weights in the back buffer that are accumulated one after another while processing all slices:

$$W_t = \sum_{(i,x,y,d)} \mathrm{weight}(i,x,y,d). \tag{4}$$

At the same time, we maintain a bounding box spanning the maximum dimensions in 2D of all target particles. This ensures that a reasonable amount of target particles are visible on the screen (while taking the weight into account). This sums up to a total number of 20 bytes per evaluation result.

Table 1 shows the performance measurements in milliseconds on all evaluation scenarios using an intermediate image size of 192×192 in combination with *FP14* and *FP32* back buffers. The cube maps column denotes the time required to render the initial cube maps and to compute the reduced processing datasets (steps 2 and 3 from Figure 2). The measurements vary between $\approx$ 3ms and $\approx$ 6ms, due to the fact that we need to render all particles (unordered in the dataset and in global memory) into the cube maps. Moreover, the more particles are visible from the perspective of the different cube-map faces, the longer it takes to compute the reduced processing dataset. The actual camera-perspective evaluation steps (including the resolution of the best result on the CPU) are presented in the *F14* and *F32* columns.

As expected, the *F14* evaluations are approximately twice as fast in most cases, which is caused by reducing the number of iterations over all slices by a factor of two. However, it also turns out that the processing time does not primarily depend on the input complexity of the dataset (e.g. scenario 2 vs. 3). It mainly depends on the number of visible particles from the reduced processing dataset that depends on individual characteristics of the input data rather than the number of input particles. Note that the runtime of the cube-map column and either the *F14* or the *F32* column must be added to get the total runtime, as both processing steps happen sequentially. This results in a total runtime of $\approx$ 47ms in the worst case on our evaluation scenarios while analyzing roughly 442000 particles in scenario 7. Possible result deviations between the *F14* and *F32* back buffer implementations are not considered here, since they do not reflect the actual imprecision on arbitrary heuristics in general. Moreover, these deviations highly depends on the distribution of the particles and the actual heuristic being used.

Scenario	#Particles	Cube-maps	σ	FP14	σ	FP32	σ
1	371792	4.4	0.2	8.1	0.4	14.3	0.5
2	286136	5.3	0.2	14.0	0.5	26.3	1.0
3	449734	4.9	0.3	10.2	0.5	18.7	0.8
4	153917	4.7	0.7	11.4	0.5	20.8	0.8
5	153917	3.1	0.4	6.0	0.4	10.0	0.4
6	251650	5.6	1.3	14.8	0.7	26.9	1.2
7	442079	6.2	0.5	21.8	0.7	41.1	1.2

Table 1: *Performance measurements in ms of the evaluation scenarios for intermediate images of size* 192×192.

Changing the intermediate image resolution has a significant impact on the overall runtime of our algorithm (see Table 2). Note that larger resolutions can only be used in combination with the *FP32* back buffer (see Section 5). As shown in the table, doubling the resolution causes a runtime increase of about 4.6x to 7.2x. Considering that the actual problem grows quadratically as opposed to runtime, our method has good scaling behavior [Amd67; Kri01]. However, the use of larger intermediate images depends on the actual problem domain and the heuristic being used. Based on our experience, we recommend using smaller intermediate images while increasing the number of camera perspectives.

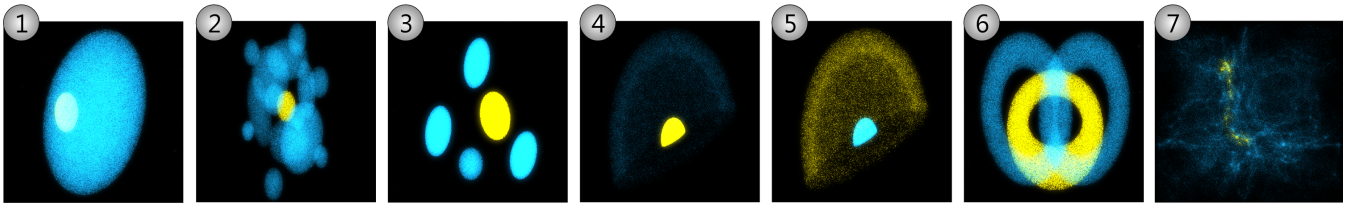

Figure 8: *A visualization of the evaluation scenarios based on the datasets published in [KK18]. Each scenario already contains a set of selected particles (yellow) and a set of (possibly) occluding particles in the surrounding. The scenarios were chosen in a way to capture common, well known simple and more complex scenarios in terms of camera control.*

192×192	384×384	768×768
41.1	295.5	1385.3

Table 2: *Performance measurements for different intermediate image resolutions using the Millenium evaluation scenario (7), an* FP32 *back buffer and* 10240 *camera perspectives.*

6.1. Memory Consumption

The memory consumption mainly depends on the width (w) and height (h) of the intermediate images. We require 6 cube map faces to be rendered in step 2 (see Figure 2) and the reduced dataset consisting of all visible particles from the cube map without duplicates (step 3, see Figure 2). As described in Section 5, the reduced dataset might become as large as the 6 cube map faces in the worst case, if there are no duplicate particles. Although this is theoretical edge case, the memory consumption should reflect the upper bound. Thereby each particle *Id* is represented as a 32bit integers in the scope of these steps. This yields

$$\text{buffers} = w \cdot h \cdot 6 \cdot 2 \cdot \text{sizeof(I32)}. \tag{5}$$

Moreover, we require additional memory allocations to store the evaluation results. The size of this buffer depends on the number of camera-perspectives (n) to be evaluated in parallel and is given by

$$\text{results} = n \cdot \text{sizeof(evalResult)}. \tag{6}$$

Depending on the actual implementation, an additional buffer storing all view matrices for the individual camera perspectives might be necessary. For evaluation purposes, we precomputed all view matrices using a fast RNG [Mar03] on the CPU and transferred all of them to a separate GPU buffer in global memory. In this case, users require an additional buffer of 4×4 float matrices, and thus

$$\text{views} = n \cdot 4 \times 4 \cdot \text{sizeof(F32)} \tag{7}$$

bytes to store all matrices.

The total memory consumption of our method using a resolution of 192×192 pixels for all intermediate images and 10240 camera perspectives to be evaluated is then around 2.5MB:

$$1728\text{KB}(\text{buffers}) + 200\text{KB}(\textit{results}) + 640\text{KB}(\textit{views}) \approx 2.5\text{MB}.$$

Assuming that we would store all intermediate images for all camera perspectives in global memory, rather than in shared memory, we would require additional

$$n \cdot w \cdot h \cdot \text{sizeof(I64)} = 2880\text{GB} \tag{8}$$

in global memory, which would exceed the memory capacity by orders of magnitude. An alternative would be use the same slicing concept we currently use in shared memory. This would reduce the number of required bytes considerably, however, it would still be significantly slower compared to the fast on-chip caches.

7. Conclusion

In this paper, we have presented a newly designed method to realize real-time camera control for particle systems. We solve this challenging task by providing a massively parallel implementation designed for GPUs that is capable of evaluating thousands of possible perspectives at the same time. Using our technique, we are even able to handle datasets consisting of hundreds of thousands of particles in real time while storing the intermediately generated images in shared memory. This avoids expensive allocations and tremendous memory buffers in global GPU memory. Therefore, our method is perfectly suitable for data- and scientific visualization applications that want to add automatic camera control.

In terms of evaluation heuristics, we used a simple and handcrafted heuristic to compute the actual camera perspectives on our evaluation scenarios. This is not a limitation of our method in general, since the overall concept can be combined with arbitrary user-defined and domain-specific evaluation functions. It supports machine-learning based heuristics that accept back-buffer inputs at the same time.

In the future, we would like to investigate more advanced heuristics and their influence on the determined camera perspectives. We would also like to cover the previously mentioned machine-learning-based heuristics to overcome potential inadequacies caused by manually tweaked heuristics. This might slightly affect the algorithm implementation, as we would like to leverage tensor processing cores [NVI21]. Furthermore, we would like to extend the method to explore more potential camera perspectives as we are currently limiting our solution space to an input bounding box. This involves leveraging more degrees of freedom to find potentially better camera setups.

Acknowledgement

This work has been developed in the project APPaM (01IW20006), which is partly funded by the German ministry of education and research (BMBF).

References

[Amd67] AMDAHL, GENE M. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities." *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. 1967 7.

[BSP05] BODOR, R., SCHRATER, PAUL, and PAPANIKOLOPOULOS, NIKOLAOS. "Multi-camera positioning to optimize task observability." IEEE, 2005 2.

[CO09] CHRISTIE, MARC and OLIVIER, PATRICK. "Camera control in computer graphics: Models, techniques and applications." 2009 2.

[CON08] CHRISTIE, MARC, OLIVIER, PATRICK, and NORMAND, JEAN-MARIE. *Occlusion-free Camera Control*. Research Report. INRIA, 2008, 27 2.

[EEB07] ELSHAMY, WESAM, EMARA, H.M., and BAHGAT, A. "Clubs-based Particle Swarm Optimization." *Proceedings of the Swarm Intelligence Symposium*. IEEE, 2007 2, 4.

[GKK19] GROSS, JULIAN, KÖSTER, MARCEL, and KRÜGER, ANTONIO. "Fast and Efficient Nearest Neighbor Search for Particle Simulations." *Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2019)*. The Eurographics Association, 2019 2.

[GKK20] GROSS, JULIAN, KÖSTER, MARCEL, and KRÜGER, ANTONIO. "CLAWS : Computational Load Balancing for Accelerated Neighbor Processing on GPUs using Warp Scheduling." *Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2020)*. The Eurographics Association, 2020 2.

[Hai09] HAIGH-HUTCHINSON, MARK. "Real Time Cameras: A Guide for Game Designers and Developers." CRC Press, 2009 2.

[HWVF12] HEGE, HANS-CHRISTIAN, WIEBEL, ALEXANDER, VOS, FRANZ. M., and FOERSTER, D. "WYSIWYP: What You See Is What You Pick." *IEEE Transactions on Visualization and Computer Graphics* (2012) 2.

[Kel06] KELAGER, MICKY. "Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics." *Camb. Monogr. Mech.* (2006) 2.

[KGK19a] KÖSTER, MARCEL, GROSS, JULIAN, and KRÜGER, ANTONIO. "FANG: Fast and Efficient Successor-State Generation for Heuristic Optimization on GPUs." *19th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP-2019)*. Springer, 2019 2.

[KGK19b] KÖSTER, MARCEL, GROSS, JULIAN, and KRÜGER, ANTONIO. "Parallel Tracking and Reconstruction of States in Heuristic Optimization Systems on GPUs." *Parallel and Distributed Computing, Applications and Technologies (PDCAT-2019)*. IEEE, 2019 2.

[KGK20] KÖSTER, MARCEL, GROSS, JULIAN, and KRÜGER, ANTONIO. "Massively Parallel Rule-Based Interpreter Execution on GPUs Using Thread Compaction." *International Journal of Parallel Programming* (2020) 5.

[KK16] KÖSTER, MARCEL and KRÜGER, ANTONIO. "Adaptive Position-Based Fluids: Improving Performance of Fluid Simulations for Real-Time Applications." *International Journal of Computer Graphics & Animation* (2016) 2, 7.

[KK18] KÖSTER, MARCEL and KRÜGER, ANTONIO. "Screen Space Particle Selection." *Proceedings of the Conference on Computer Graphics & Visual Computing (CGCV-2018)*. The Eurographics Association, 2018 2, 8.

[KLH*14] KÖSTER, MARCEL, LEISSA, ROLAND, HACK, SEBASTIAN, et al. "Code Refinement of Stencil Codes." *Parallel Processing Letters (PPL)* 24 (2014) 5.

[KM05] KNEAFSEY, JAMES and MCCABE, HUGH. "CameraBots: Cinematography for Games with Non-Player Characters as Camera Operators." *DiGRA Conference*. 2005 2.

[Kri01] KRISHNAPRASAD, S. "Uses and Abuses of Amdahl's Law." *J. Comput. Sci. Coll.* (2001) 7.

[LSG19] LUSTIG, DANIEL, SAHASRABUDDHE, SAMEER, and GIROUX, OLIVIER. "A Formal Analysis of the NVIDIA PTX Memory Consistency Model." *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019 5.

[LST04] LIN, TING-CHIEH, SHIH, Z., and TSAI, YU-TING. "Cinematic Camera Control in 3D Computer Games." *WSCG*. 2004 2.

[Mar03] MARSAGLIA, GEORGE. "Xorshift RNGs." *Journal of Statistical Software, Articles* 8 (2003) 8.

[MCG03] MÜLLER, MATTHIAS, CHARYPAR, DAVID, and GROSS, MARKUS. "Particle-based Fluid Simulation for Interactive Applications." *Symposium on Computer Animation*. 2003 2.

[MK06] MCCABE, HUGH and KNEAFSEY, JAMES. "A Virtual Cinematography System for First Person Shooter Games." *Proceedings of International Digital Games Conference*. 2006 2.

[MLBT11] MELAB, NOUREDINE, LUONG, THÉ VAN, BOUFARAS, K, and TALBI, EL-GHAZALI. "ParadisEO-MO-GPU: a framework for parallel GPU-based local search metaheuristics." *11th International Work-Conference on Artificial Neural Networks*. 2011 2.

[MMCK14] MACKLIN, MILES, MÜLLER, MATTHIAS, CHENTANEZ, NUTTAPONG, and KIM, TAE-YONG. "Unified Particle Physics for Real-time Applications." *ACM Trans. Graph.* (2014) 2.

[Ngu07] NGUYEN, HUBERT. *GPU Gems 3*. Addison-Wesley Professional, 2007 4.

[NVI14] NVIDIA. *Faster Parallel Reductions on Kepler*. 2014 6.

[NVI21] NVIDIA. *CUDA C Programming Guide v11.3*. 2021 6, 8.

[ONI05] OWADA, SHIGERU, NIELSEN, FRANK, and IGARASHI, TAKEO. "Volume Catcher." *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. 2005 2.

[PF05] PHARR, MATT and FERNANDO, RANDIMA. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, 2005 2.

[Sho92] SHOEMAKE, KEN. "ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse." *Proceedings of the Conference on Graphics Interface '92*. Morgan Kaufmann Publishers Inc., 1992 2.

[Tuk77] TUKEY, JOHN W. *Exploratory Data Analysis*. 1977 2.

[VFSG06] VIOLA, IVAN, FEIXAS, MIQUEL, SBERT, MATEU, and GROLLER, MEISTER EDUARD. "Importance-Driven Focus of Attention." *IEEE Transactions on Visualization and Computer Graphics* (2006) 3.

[YBH11] YI-CHUN, XU, BANGJUN, LEI, and HENDRIKS, EMILE. "Camera Network Coverage Improving by Particle Swarm Optimization." *EURASIP Journal on Image and Video Processing* 2011 (2011) 3.

[YEII12] YU, LINGYUN, EFSTATHIOU, KONSTANTINOS, ISENBERG, PETRA, and ISENBERG, TOBIAS. "Efficient Structure-Aware Selection Techniques for 3D Point Cloud Visualizations with 2DOF Input." *IEEE Transactions on Visualization and Computer Graphics* (2012) 2.

[YEII16] YU, LINGYUN, EFSTATHIOU, KONSTANTINOS, ISENBERG, PETRA, and ISENBERG, TOBIAS. "CAST: Effective and Efficient User Interaction for Context-Aware Selection in 3D Particle Clouds." *IEEE Transactions on Visualization and Computer Graphics* (2016) 2, 7.

[YMJ10] YANNAKAKIS, GEORGIOS, MARTÍNEZ, HÉCTOR, and JHALA, ARNAV. "Towards affective camera control in games." *User Modeling and User-Adapted Interaction* (2010) 2.

[ZSS11] ZHAO, YAO, SHURALYOV, DMITRI, and STUERZLINGER, WOLFGANG. "Comparison of Multiple 3D Rotation Methods." IEEE, 2011 2.