GPU-Accelerated Next-Best-View Coverage of Articulated Scenes

Stefan Oßwald Maren Bennewitz

***Abstract*— Next-best-view algorithms are commonly used for covering known scenes, for example in search, maintenance, and mapping tasks. In this paper, we consider the problem of planning a strategy for covering articulated environments where the robot also has to manipulate objects to inspect obstructed areas. This problem is particularly challenging due to the many degrees of freedom resulting from the articulation. We propose to exploit graphics processing units present in many embedded devices to parallelize the computations of a greedy next-best-view approach. We implemented algorithms for costmap computation, path planning, as well as simulation and evaluation of viewpoint candidates in OpenGL for Embedded Systems and benchmarked the implementations on multiple device classes ranging from smartphones to multi-GPU servers. We introduce a heuristic for estimating a utility map from images rendered with strategically placed spherical cameras and show in simulation experiments that robots can successfully explore complex articulated scenes with our system.**

I. Introduction

Many tasks of service robots involve covering a known environment with the robot's sensor, for example when searching for an object, patrolling a building, or executing maintenance and inspection tasks. In home and office environments, such tasks often require that the robot actively manipulates objects to inspect their contents, for example open cupboards, drawers, and dishwashers, or move furniture aside to observe occluded areas. In this paper, we propose an information gain-based strategy for covering an environment with articulated objects that the robot can manipulate. We assume that the robot already has a model of the environment, for example a CAD model provided by the user or a 3D map recorded with a SLAM system, as well as user-defined regions of interest that the robot shall cover and knowledge about how to manipulate the articulated objects in the scene.

This coverage problem is hard even for static scenes, as well-known NP-hard problems such as the art gallery problem and the set coverage problem can be reduced to coverage of known scenes. Considering articulated objects adds even more degrees of freedom, making it infeasible to solve the planning problem by exhaustive search. While several approaches for covering known scenes exist in the literature, including sampling-based and probabilistic techniques, these methods mostly consider only static scenes due to the high complexity of the planning problem.

In this paper, we propose a greedy next-best-view approach that selects the next viewpoint by maximizing the expected utility, which trades off the costs for the robot navigating to the viewpoint and the expected information gain. Estimating this utility function contains several subproblems. First, we define the information gain as the size of the newly observed portion of the region of interest in the camera image, hence our algorithm needs to render a virtual camera view from each potential view pose. Second, in the navigation cost function, we need to consider that the robot has to keep a safety clearance to obstacles, so we follow the popular approach of computing an inflation cost map. Third, determining the costs for navigating to viewpoint candidates includes solving a single-source shortest path problem. All these problems are highly parallelizable. Hence, we propose to exploit the computer's graphics processing unit (GPU) to parallelize the workload for solving the subproblems, which allows us to compute utility maps for a large number of articulation configurations in a reasonable amount of time, making the planning problem more tractable.

Fig. 1. Nao inspecting a kitchen environment. The robot's task is to search for an object in the user-defined regions of interest marked in red. Kitchen model based on [1].

Fast GPUs are widespread and due to the popularity of deep learning algorithms their power and availability will likely increase even more in the future. In the literature, algorithms for some of the mentioned subproblems have already been successfully demonstrated on general-purpose GPUs (GPGPUs) that support high-level frameworks such as CUDA, OpenCL, or ROCm. Variations of Dijkstra's shortest path algorithm, for example, have been implemented on CUDA [2]. GPGPU approaches, however, need high-end graphics cards. Embedded systems such as robots, smartphones, and single-board computers often do have GPUs, but only provide a subset of functions needed for their original purpose of graphics processing. To leverage the computing power of embedded systems GPUs, we formulate all algorithms as rendering problems and implement them in OpenGL for Embedded Systems (OpenGL ES), which is an open standard and widely supported across platforms.

All authors are with the Humanoid Robots Lab, University of Bonn, Germany.

There are three main contributions of this paper: First, we adapt the jump flood algorithm for computing cost inflation maps, the Bellman-Ford algorithm for shortest path planning, and a view simulation algorithm for estimating the information gain for being solved with an OpenGL ES graphics pipeline. We show optimizations for increasing the throughput of the algorithms and make use of modern OpenGL features such as transform buffer feedback and random image access. We benchmark the algorithms on multiple device classes ranging from smartphones to multi-GPU servers.

Second, we introduce a heuristic for estimating a utility map based on rendering the scene with virtual spherical panorama cameras placed strategically at the edge of articulated objects and compare the result with a ground-truth utility map obtained by exhaustive sampling.

Third, we integrate the algorithms and the heuristic into a system for covering an articulated scene and show in simulation experiments how a simulated robot successfully explores home environments such as the kitchen environment in Fig. 1.

The source code of our implementation is available at `https://www.hrl.uni-bonn.de/gpu-coverage`.

II. Related Work

The optimization problem of finding the smallest set of viewing points for observing a known environment has been formulated as the *art gallery problem*, which is known to be NP-hard and APX-hard even in 2D environments [3]. After determining the optimal view points, computing the shortest tour visiting those view points is an instance of the traveling salesman problem, which is also NP-hard. Due to the high complexity of the problem, exact solutions are out of reach. Hence, most approaches in the literature resort to iterative, greedy next-best-view algorithms. Bissmarck *et al.* [4] provide an overview and run-time comparison for existing next-best-view solutions. These algorithms, including newer probabilistic formulations of the information gain estimation problem [5], are designed for CPUs. In this paper, we propose to leverage the GPU to accelerate the search for the next-best-view candidates.

Dornhege *et al.* [6] use sampling and raycasting in an octree environment representation to generate a larger number of viewpoint candidates with high utility and apply set coverage optimization to minimize the number of viewpoints. A traveling salesman planner then calculates the shortest tour. As sampling and raycasting are particularly expensive operations, we propose to calculate utility maps by rendering on GPUs instead to improve performance. While we use a greedy, iterative scheme to compute the tour from viewpoints, integrating high-level symbolic planning could be integrated in future work to optimize the tours.

In our previous work [7], we focused on planning viewpoints for covering a known environment with a humanoid robot. Querying inverse reachability maps while searching for next-best-views allowed us to exploit the mobility of humanoid robots by bending over to peek into boxes or squatting down to look under tables. In the current paper, we present an approach for manipulating articulated objects for inspecting the contents of containers. Both approaches are complementary and will be combined in the future to leverage the full potential of humanoids.

To the best of our knowledge, there are no existing algorithms for calculating exploration or coverage tours in environments with articulated objects that the robot has to actively manipulate.

There are several approaches for solving navigation problems on GPGPUs, for example for path planning with Dijkstra's algorithm [2], the Bellman-Ford algorithm with bucketing [8], parallel breadth-first search [9], or potential fields [10]. These algorithms, however, need GPGPU frameworks such as CUDA. In our work, by contrast, we limit GPU usage to the embedded systems subset of OpenGL that is more widespread and cross-platform. Camporesi and Kallmann [11] also use OpenGL GPU shaders to compute shortest paths based on shortest path trees. In contrast to their approach, our approach is not restricted to 2D polygonal input data, but can use any renderable scene.

III. Problem formulation

In this paper, we consider the problem of covering user-defined regions of interest with a robot's sensor. We assume that the robot already has a renderable 3D model of the environment, either provided by the user or previously acquired using a SLAM approach and that the model contains a mechanism for moving the articulated objects. In our implementation, we represent the scene as a Collada model and the articulation mechanisms as bone animations. Each possible pose of an articulated object is mapped to a position variable $p \in [0, 1]$. For a sliding door, for example, $p = 0$ means that the door is closed and $p = 1$ means that the door is fully open. For movable furniture, p indicates the position of the piece of furniture along a defined path.

Let A be the set of articulation objects contained in the scene. Each articulation object $a \in A$ is represented as $a = (p_0, c)$, where $p_0 \in [0, 1]$ is the initial state of the articulated object and $c\,(p_{i-1}, p_i)$ is a cost function returning the costs for moving the articulated object from position p_{i-1} to p_i.

We assume that the user defines the robot's task by specifying one or more regions of interest R that the robot should cover. In our implementation, we represent the regions of interest as textured objects with a designated color, marked in red in the figures throughout this paper. The algorithm then keeps track of observed portions of R by marking them in a different designated color (green in this paper).

We also assume that the scene model contains one or more transparent surfaces C representing all locations where the robot's camera can be placed. For mobile robots with cameras mounted on the base these surfaces will typically be planes parallel to the ground, whereas for arm-like robots sphere surfaces can be used.

The goal of our approach is to find a sequence $s_1, s_2, \ldots$ of elements $s_i = \left(v_i, p_i^1, \ldots, p_i^{|A|}\right)$ consisting of viewpoints v_i

and articulation positions p_i^j with low costs from where the robot can see as much as possible of the defined regions of interest. We define the costs as

$$cost(s_i) = infl(v_i) + dist(v_{i-1}, v_i) + \sum_{a=1}^{|A|} c(p_{i-1}^a, p_i^a), \quad (1)$$

where $infl(v_i)$ are costs exponentially decaying with the distance to the nearest obstacle following the commonly used inflation costmap approach, $dist(v_{i-1}, v_i)$ is the shortest path distance from the robot's current position v_{i-1} to the new viewpoint v_i, with v_0 designating the robot's initial position, and c are the costs for manipulating the articulated objects as defined above. In a single timestep, multiple articulation objects may be moved, but typically only one articulation object has to be changed to uncover a region of interest.

Finally, we define the information gain $IG(v)$ of visiting a camera viewpoint v as the number of texels of the region of interest R that are visible from v and have not been observed before. The goal of our approach is then to maximize the utility function

$$U(s_i) = \alpha \cdot IG(v_i) - \beta \cdot cost(s_i) \quad (2)$$

to iteratively determine the next best view. The coefficients α, β trade off between the conflicting optimization goals of the required runtime and the completeness of the coverage and have to be adapted to the robot's task priorities. If the highest utility of all viewpoint candidates drops below zero, the algorithm ends, potentially leaving regions with low information gain and high costs unexplored.

For determining a sequence of viewpoints, several online and offline approaches can be used and the optimal choice depends on the robot's task. If the task requires full coverage as in mapping and inspection tasks, a cost minimizing planner such as a traveling salesman problem solver should be used. In this paper, however, we consider an object search task where it makes sense to start with panoramic view points where large parts of the environment are visible, as it is likely that the object can be seen from these view points and the search can be completed early. Hence, we choose a utility-based greedy approach for our scenario. For a detailed comparison and benchmark of different planning algorithms in exploration and coverage contexts see the survey by Dornhege *et al.* [6].

IV. GPU Algorithms

We tailor our algorithms to GPUs of embedded systems, hence we implement our approach in OpenGL for Embedded Systems version 3.2, which is an open standard and widely supported on many platforms.

A. Properties of the Graphics Pipeline

OpenGL defines a pipeline of *shaders*. Shaders are simple programs that the GPU executes either for each vertex or for each pixel. The pipeline stages relevant for this work are:

1) The *vertex shader* takes one vertex of the scene mesh at a time in world coordinates and transforms the coordinates to screen coordinates by multiplying with camera and projection matrices.
2) The *geometry shader* takes one primitive of the scene mesh at a time (i.e., a point, line, or triangle) and outputs an arbitrary number of primitives of the same type. This mechanism can be used to manipulate the geometry of scene parts or to send multiple copies of scene parts to subsequent stages.
3) The *rasterizer* converts geometric primitives to a set of pixels and interpolates vertex attributes such as color and texture coordinates. The combined data for generating an output pixel is called *fragment*.
4) The *fragment shader* takes the fragment data for one pixel at a time and computes the final output color of the pixel. The fragment shader can neither modify the pixel position, nor consider neighboring pixels.

After the fragment shader, the GPU optionally performs a *depth test* that ensures that a fragment only overwrites a previously written pixel if the new fragment's depth at the pixel location is closer to the camera than the previously written fragment's depth, thus solving the z ordering problem. If the fragment shader promises not to modify the depth value, the depth test can be performed already after the rasterizer in a technique called *early depth testing*.

All shaders can perform additional work, e.g., reading and writing to textures or passing variables to other pipeline stages. The performance gain that GPUs offer is mainly achieved by running the shader programs in parallel for multiple vertices or pixels using Single Instruction Multiple Data (SIMD) architectures. Considering the special architecture of GPUs, we followed these design guidelines:

- SIMD instructions by definition require that the same instruction is executed for all data. Branching into **if/else** constructs reduces the possibilities for parallelization, hence branching should be avoided where possible. While lazy evaluation paradigms speed up sequential computations, the overhead of computing unneeded data with SIMD might be smaller than branching based on whether or not the data is needed.
- The GPU processes the pipeline asynchronously. The graphics driver keeps the pipeline filled with about three next instructions for the GPU. Reading back data from the GPU to the CPU may introduce synchronization points that flush the pipeline. Following best practices [12], [13], [14], our GPU algorithms write to ring buffers that the CPU reads back with a delay of three frames, allowing the GPU to continue writing the next frame to a different buffer while data is transferred from an older buffer.
- In our implementation, we reduce data transfer between CPU and GPU to the minimum. All intermediate results are held on the GPU and only the final results are transferred to the CPU through a ring buffer.

B. Costmap Computation

For safe navigation, robots need to keep a safety clearance from obstacles. A commonly used approach (e.g., implemented in the ROS navigation stack [15]) is to create an occupancy grid map indicating the obstacles and free space

Algorithm 1: Costmap computation of a $n \times n$ gridmap

```
// Initialization
render obstacles
for all obstacle fragments p do in parallel:
    color_p ← encode(coordinates of p)
// Jump Flood Algorithm
for all i ∈ {1, ..., log(n)} do:
    s ← 2^(log(n)−i)
    for all map pixels p do in parallel:
        for q ∈ {p + (x, y) | x, y ∈ {−s, 0, s}} do:
            if |decode(q) − p| < |decode(p) − p| then
                color_p ← color_q
// Compute cost map
for all map pixels do:
    compute distance to color-coded cell coordinate
    compute cost value and write to output
```

and to "inflate" the size of the obstacles by the safety margin. In a costmap, the costs for all cells inside the inflation radius around obstacles are set to infinity or a high constant. To get smoother paths around obstacles, the cost function is often modeled as exponentially decaying from inflation edges towards free space so that the robot can get close to objects if necessary, but gets an incentive to stay further away.

For computing both the inflation cells and the decay function, the distance of each cell to the nearest obstacle is required. We use the jump flood algorithm (JFA) [16] to compute a Voronoi diagram where each cell contains the coordinates of the nearest occupied cell, which can directly be converted to a distance transform map. For a map of size $n \times n$, the algorithm needs one shader pass for initialization, $\log n$ shader passes for propagating the coordinates of the nearest obstacles, and one shader pass to compute the Euclidean distance and the cost function. The resulting distance transform is not exact, as the error discussion in [16] shows, but we found that the errors are negligible in our application, especially as the costmap is only an approximation of the actual navigation cost.

Alg. 1 gives a high-level overview of the algorithm. The basic idea of the JFA is to encode the coordinate of the nearest obstacle found so far as a color value in each cell and to propagate the information on the nearest obstacle in exponentially decreasing "jumps" across the map. We use a method encode $(\cdot)$ to represent cell coordinates as color values and its inverse method decode $(\cdot)$. Hence, $|\text{decode}\,(p) - p|$ is the Euclidean distance between the nearest obstacle encoded as a color value in cell p and the coordinates of cell p itself. For details on the JFA, we refer to [16].

C. Single-Source Shortest Path

Planning the shortest path from the robot's current location to one or more destinations is a basic component for many navigation tasks. While there exist efficient algorithms for solving the single-source shortest path (SSSP) problem, these algorithms rely on special data structures such as

Algorithm 2: Bellman-Ford algorithm variant with transform feedback buffer (XFB)

```
// Initialization
for all map pixels p do in parallel:
    dist_p ← { 0   if pixel is robot's current location
             { ∞   otherwise
    changed_p ← { true   if pixel is robot's location
                { false  otherwise
// Wavefront propagation
emit robot's neighbor cell coordinates to XFB
while XFB is not empty do:
    for all cells p in XFB do in parallel:
        changed_p ← false
        for all neighbor pixels q of p do:
            if changed_q then
                d ← dist_q + ‖p − q‖ + cost_p
                if d < dist_p then
                    dist_p ← d
                    changed_p ← true
        if changed_p then
            emit neighbor cells of p to XFB
```

Fibonacci heaps (e.g., Dijkstra's algorithm [17]) or bucketing structures (e.g., Thorup's algorithm [18]) that are not suitable for being implemented in a graphics pipeline. The core problem with these algorithms is that they rely on processing nodes sequentially in a particular order, whereas GPUs are most efficient at processing nodes in parallel.

The Bellman-Ford algorithm [19], [20], by contrast, can be parallelized. In a grid map with the same size as the occupancy grid map of the environment, we store the shortest-path distance from the robot's location. We initialize the robot's current grid cell with distance 0 and all other cells with ∞. For each grid cell, we then check which of the eight neighbor cells have changed and decrease the cell's distance in case a shorter path via a changed neighbor has been found. In this way, a wavefront of cells with decreasing distance emerges, and we repeat the process as long as cells change. The speedup of using the GPU stems from processing all wavefront cells in parallel.

We present two implementations. In our first implementation in Alg. 2, we use *transform feedback buffers* (XFB) that modern OpenGL ES implementations support. Transform feedback buffers capture the output of the geometry shader in a buffer that can be used as input for the vertex shader in subsequent passes. We enqueue the neighbor cells of a changed cell by emitting the neighbor cell's coordinate from the geometry shader. The next pass processes in parallel only the queued cells, reducing overhead. However, this technique requires both XFB and texture access in the geometry shader, which is not supported on some of the tested devices.

For devices that do not support XFB, we propose the second implementation in Alg. 3. We use a "changed" flag to mark changed cells similar to the *EBellflaging* algorithm in [21]. We

Algorithm 3: Bellman-Ford algorithm for single-source shortest path computation

// *Initialization*
for all map pixels p **do in parallel:**
    $dist_p \leftarrow \begin{cases} 0 & \text{if pixel is robot's current location} \\ \infty & \text{otherwise} \end{cases}$
    $changed_p \leftarrow \begin{cases} \mathbf{true} & \text{if pixel is robot's location} \\ \mathbf{false} & \text{otherwise} \end{cases}$
// *Wavefront propagation*
$\#changed \leftarrow 1$
while $\#changed > 0$ **do:**
    $\#changed \leftarrow 0$
    for all map pixels p **do in parallel:**
        $changed_p \leftarrow \mathbf{false}$
        for all neighbor pixels q of p **do:**
            if $changed_q$ **then**
                $d \leftarrow dist_q + \|p - q\| + cost_p$
                if $d < dist_p$ **then**
                    $dist_p \leftarrow d$
                    $changed_p \leftarrow \mathbf{true}$
        if $changed_p$ **then**
            atomically increment $\#changed$

encode the "changed" flag as the sign and the current distance as the absolute value of a single-channel integer texture. As we are only interested in the length of the shortest path, we do not store a pointer to the predecessor of each cell. OpenGL enforces protection against memory access collisions when processing pixels in parallel by denying fragment shaders to write to other pixels and by requiring separate buffers for reading and writing. Hence, we read from and write to two separate buffers and swap the buffers after each pass. We implemented the counter for counting changed pixels as an atomic counter and read back the counter value with a delay of three frames to prevent CPU-GPU synchronization points with pipeline flushes as explained in Sec. IV-A. The overhead of processing two extra shader passes after the wavefront has stopped is negligible in comparison to the loss of computation power that would occur with synchronous readbacks requiring to flush the graphics pipeline.

The disadvantage of the second implementation is that in each shader pass, the fragment shader has to check for each neighbor whether it has changed in the previous iteration. Reading neighbor cells require costly texture fetches.

After computing the shortest distance map with either of the Bellman-Ford implementations, we combine the distance map including the inflation costs from Alg. 1 with the articulation costs, which are constant for all cells, to a combined cost map according to Eq. (1).

D. Information Gain Computation

The information gain of a viewpoint is based on counting texels of the region of interest that are observable from the viewpoint and have not been seen before. When rendering

Algorithm 4: Computing the information gain of a viewpoint candidate

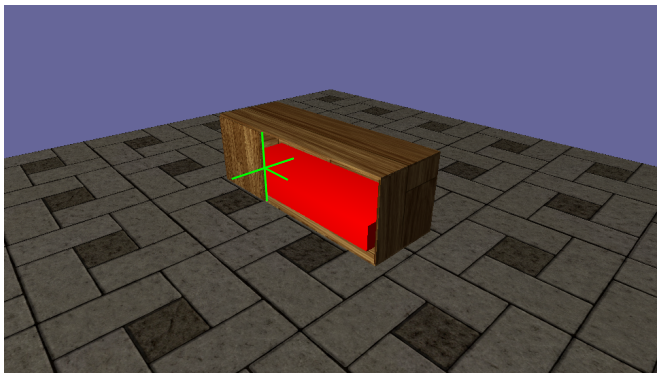
enable depth test
render obstacles to depth buffer
render region of interest surfaces
for all front facing region of interest fragments p **do in parallel:**
    compute texture coordinate t corresponding to p
    store "observed" flag to region of interest texel t
$\#covered \leftarrow 0$
for all region of interest texels t **do in parallel:**
    if t has "observed" flag **then**
        atomically increment $\#covered$

a region of interest object, the fragment shader accesses the texel at the texture coordinates given in the fragment data to determine the fragment's color. Using the `imageStore` command present since OpenGL ES 3.1, the fragment shader can modify the texture at the same time. We use this mechanism to mark the texel as observed by writing a flag to the texture. By rendering the region of interest objects last with early depth tests and backface culling enabled, we make sure that the fragment shader only marks texels visible from the camera as observed. Counting the newly marked texels then yields the information gain value as Alg. 4 shows.

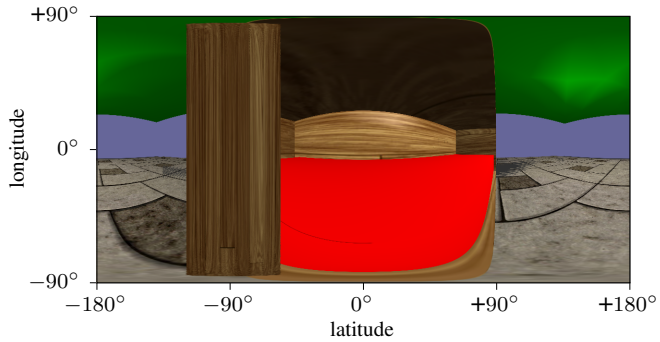
E. Estimating the Utility Map

Alg. 4 can be used to estimate the information gain for a single camera viewpoint. For choosing the next best view optimizing the utility function in Eq. (2), we would need to compute the information gain for every possible camera location. Alternatively, the algorithm could render an image with virtual cameras placed at every possible region of interest surface location to determine the freespace volume from where the given region of interest texel can be observed. Both variants would require rendering a large number of images from different camera locations, causing high computational loads. Hence, we propose a novel heuristic for estimating an information gain map by rendering a spherical panorama image with a virtual camera placed at the edge of articulated objects or other obstacles. The core idea is that a spherical camera image placed in the opening of a drawer, cupboard, or other container shows the inside of the container on one side and the freespace volume from where the contents can be observed on the opposite side of the panorama image. Fig. 2b shows an example panorama rendered by a virtual camera placed at the crosshair in Fig. 2a.

Alg. 5 describes the standard method of rendering a spherical panorama by first rendering all objects to the six sides of a cubemap and then projecting the cubemap texture to equirectangular coordinates. We first render all obstacles to a depth cubemap only, as we do not need the color information of the obstacles. With depth testing enabled, we then render the region of interest surfaces with the texture from Alg. 4 that indicates for each texel whether it has been observed



(a) Scene with a cupboard. The region of interest is marked in red.



(b) Same scene rendered with a spherical panorama camera located at the crosshair in (a). The green surface indicates the reachable camera poses.

Fig. 2. Example scene illustrating the process of estimating an information gain map. The scene is rendered as a spherical panorama image from virtual cameras placed at the edges of articulated objects. If a region of interest (red) is opposite the surface of reachable camera poses (green), then there is a direct line of sight. We use the panorama image to determine which region of interest parts are getting into the field of view when the camera moves on the reachable surface.

before or not. As described in Sec. III, we assume that the environment model contains one or more surfaces where the robot's camera can be placed. We render these surface with a special texture that encodes the texture coordinate of each texel as its color value. This technique allows us to convert back from spherical coordinates to coordinates of the reachable surface later for projecting information gain values onto the costmap. Fig. 2b shows the result for an example scene. If a region of interest texel t and a camera viewpoint v on the surface of reachable poses are on opposite sides of the sphere, that means that t just becomes visible when the camera moves to v.

To estimate the information gain of moving the camera to v, we calculate an integral image by accumulating the number of region of interest pixels while pivoting a ray around the spherical camera's center. Fig. 3 illustrates the concept of computing the integral images for two spherical panorama cameras c_1 and c_2. At the panorama image cell A, the container edge blocks the line of sight through the camera origin to the region of interest. Pivoting counterclockwise around the camera center, the region of interest first becomes visible at B. Moving towards C, we increment the information gain estimate as long as new region of interest texels come

Algorithm 5: Rendering a spherical panorama with semantic information

```
enable depth test
render to 6-side cubemap:
    obstacles (depth only)
    region of interest surfaces with texture from Alg. 4
    reachable surfaces with index texture
for all color cubemap pixels do
    project pixel to equirectangular coordinates
```

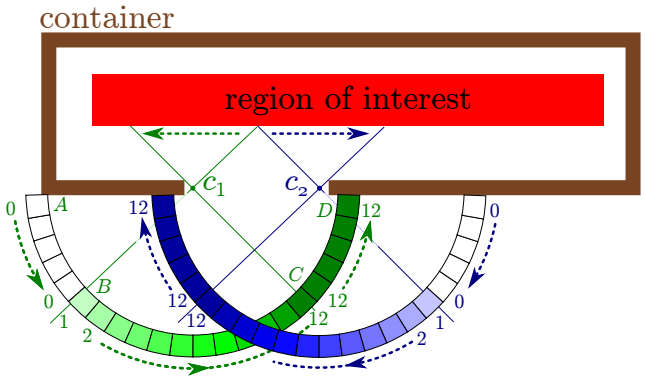


Fig. 3. Estimating the information gain map by calculating the integral images of two spherical cameras c_1 and c_2. The numbers indicate the estimated information gain accumulated by pivoting rays through the camera center in the direction indicated by arrows.

into view. Between C and D, the container edge again blocks the line of sight to new region of interest texels, hence the information gain stays constant. Adding the information gain estimates for the two cameras c_1 and c_2 gives a good approximation of the true information gain map without having to render panoramas at all camera positions between c_1 and c_2. Computing the integral image is done in the first part of Alg. 6.

The remainder of the algorithm estimates the utility map. As we render the surface of reachable camera poses with a special index texture that encodes the texture coordinates on the surface in Alg. 5, we can use these coordinates to compute the corresponding position on the cost map. If a pixel $p = (lat, lon)$ on the sphere image belongs to the reachable surface, the algorithm looks up the associated cost value from the cost map and subtracts it from the estimated information gain value encoded at the pixel $(\overline{lat}, \overline{lon})$ on the opposite side of the sphere, writing the output to a utility map with the same coordinate system as the cost map.

F. Covering Environments with a Robot's Camera

We use the utility map estimated with Alg. 6 to determine the next best view pose in a greedy iterative scheme. For each articulation object, the robot uses Alg. 6 to estimate how the utility map changes when manipulating the object. Our algorithm currently actuates objects separately to avoid collisions with other articulated objects, as for example doors and drawers of cupboards can block each other. A high-level planner could be used to resolve this issue and generate

Algorithm 6: Estimating utility map from spherical panorama image

```
// calculate integral image I
for all lon ∈ [−90°, 90°] do in parallel:
    c(lon) ← 0
for lat ∈ {−180°, ..., 180°} do in sequence:
    for all lon ∈ [−90°, 90°] do in parallel:
        if pixel at (lat, lon) is region of interest then
            c(lon) ← c(lon) + 1
        I(lat, lon) ← c(lon)
// calculate utility map U
for all pixel p at lon ∈ [−90°, 90°], lat ∈ [−180°, 180°] do in parallel:
    coord ← color-decode(p)
    if p is marked as reachable surface then
        lat ← lat + 180°
        lon ← −lon
        U(lat, lon) ← I(lat, lon) − cost(coord)
    else
        U(lat, lon) ← −cost(coord)
```

feasible configurations of multiple objects. The algorithm then determines the articulation object and the robot position on the utility map with the highest estimated utility. To determine the best camera orientation, we sample orientations on a unit sphere and use Alg. 4 to determine the orientation with the highest information gain.

V. Experimental Evaluation

We implemented our approach in OpenGL ES 3.2 and evaluated it with models of home and office environments created with CAD software. We first present the results of benchmarking the individual algorithms and then show experiments where a robot successfully covers an environment with our novel utility map heuristic that takes into account the articulation of objects.

A. Benchmarking GPU Algorithms

We designed our implementation to be compatible with a wide range of devices ranging from embedded devices to multi-GPU servers, hence we evaluated the performance on different device classes. Tab. I lists the technical specifications of the tested devices. We chose an Android smartphone as a representative of embedded devices and implemented the CPU-side code using the Android Native Development Kit (NDK) in C++. Apart from device-specific initialization, we used the same code base on all systems.

Tab. II shows the benchmarking results for each device. Alg. 2 could not be tested on the notebook device due to missing support for image access in geometry shaders. While the OpenGL API is standardized, the GPU architectures and driver implementations differ significantly between manufacturers and device generations, hence the benchmarking results also vary strongly between devices. The algorithms for costmap computation, information gain estimation, and panorama rendering run in less than 3 ms on all devices, showing the potential of our approach. The standard deviation of computation times is very small in many cases, which is due to the design principle of reducing branching, making the runtime independent of the input data.

TABLE I

Devices used for benchmarking

Class	Test device
Smartphone	Device: Google Pixel CPU: Qualcomm Snapdragon 821 (4 cores, 2.4 GHz) GPU: Qualcomm Adreno 530 OpenGL profile: 3.2 ES
Notebook	CPU: Intel i7-4710MQ (4 cores, 8 threads, 3.5 GHz) GPU: Intel Haswell Mobile HD 4600 OpenGL profile: 3.3 core
Desktop	CPU: Intel Core i7-3770 (4 cores, 8 threads, 3.4 GHz) GPU: NVIDIA GeForce GTX 660 Ti OpenGL profile: 4.4 core
Server	CPU: Intel Xeon E5-1630 (4 cores, 8 threads, 3.7 GHz) GPU: 4×NVIDIA GeForce GTX 1080 OpenGL profile: 4.6 core

TABLE II

Benchmark results: time per frame with standard deviation. See Tab. I for device specifications.

Algorithm	Smartphone	Notebook
Alg. 1: costmap	2.572 ± 2.492 ms	0.142 ± 0.010 ms
Alg. 2: Bellman-Ford XFB	251.4 ± 26.87 ms	N/A
Alg. 3: Bellman-Ford	272.9 ± 21.05 ms	52.29 ± 0.987 ms
Alg. 4: information gain	2.116 ± 7.228 ms	0.092 ± 0.006 ms
Alg. 5: panorama rendering	0.667 ± 0.541 ms	0.103 ± 0.069 ms
Alg. 6: utility map	575.2 ± 32.17 ms	245.4 ± 2.284 ms

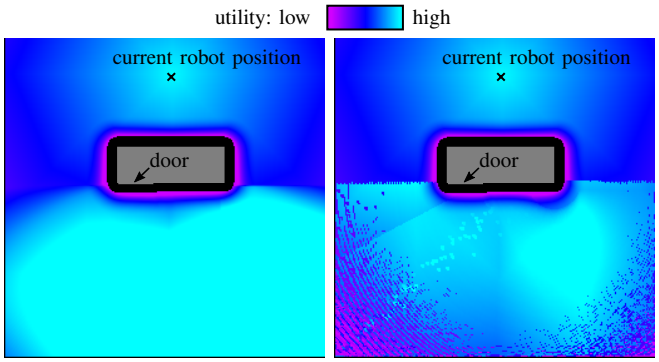
Algorithm	Desktop	Server
Alg. 1: costmap	0.097 ± 0.013 ms	0.071 ± 0.002 ms
Alg. 2: Bellman-Ford XFB	11.08 ± 0.274 ms	8.694 ± 0.040 ms
Alg. 3: Bellman-Ford	10.92 ± 0.103 ms	9.432 ± 2.191 ms
Alg. 4: information gain	0.099 ± 0.016 ms	0.084 ± 0.002 ms
Alg. 5: panorama rendering	0.083 ± 0.009 ms	0.060 ± 0.024 ms
Alg. 6: utility map	30.92 ± 0.745 ms	12.34 ± 0.723 ms

B. Estimating Information Gain

Fig. 4 shows a comparison of the estimated utility map and the real utility map generated by exhaustive sampling for the cupboard scene in Fig. 2a. As can be seen, the estimated utility map is similar to the real utility map. Our heuristic, however, focuses on covering hard-to-reach corners due to the placement of the virtual spherical cameras at the edges of articulated objects. The information gain of center portions of regions of interest tend to be underestimated, but these regions are usually covered along the way when inspecting the rest of the region. Coverage of the center regions can be improved by adding additional panorama cameras inside the openings of articulated objects.

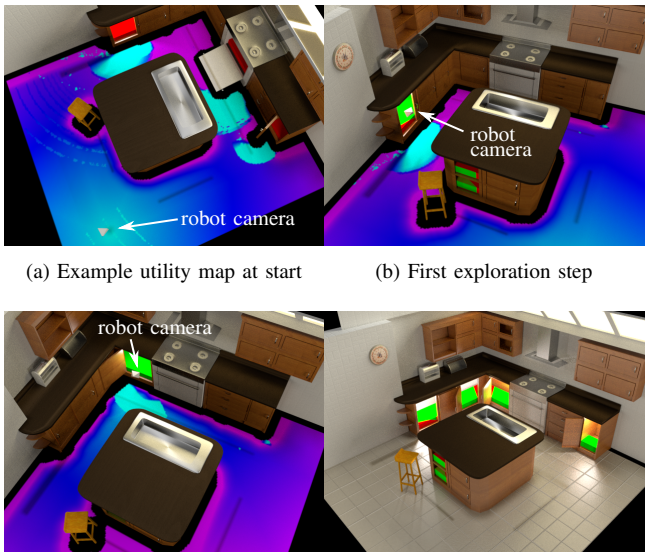
C. Covering Environments with a Robot's Camera

Fig. 5 shows the progress of a robot covering a kitchen environment. Starting from the initial pose in Fig. 5a, the robot estimates a utility map for each articulated object in turn. The floor texture visualizes the combined estimated utility map for an example configuration with three articulated objects partly opened. The robot then chooses the articulated object



(a) Real utility map (b) Estimated utility map

Fig. 4. Comparison of the real utility map obtained by exhaustive sampling and the estimated utility map generated by Alg. 6 for the scene with a cupboard shown in Fig. 2.



(a) Example utility map at start (b) First exploration step

(c) Second exploration step (d) Result after 7 steps

Fig. 5. Progress of exploring a kitchen scene. The robot first estimates utility maps for multiple articulation configurations. The robot then chooses the articulation object and view point with the highest utility (b). Afterwards, the algorithm computes utility maps from the new location taking into account already observed regions. Regions of interest are marked in red, observed regions in green, and the robot's camera is symbolized as a white pyramid.

with the highest estimated utility, articulates the object, and investigates the scene from the location and camera orientation with the highest utility (Fig. 5b). The observed regions of interest are marked in green. The average time required to compute the next-best-view position is $4.85\,\mathrm{s} \pm 0.29\,\mathrm{s}$ for this scene on the desktop computer specified in Tab. I.

While we focused on covering regions of interest occluded by articulated objects, the algorithms proposed in this paper are also applicable to covering arbitrary scenes. To apply our heuristic for estimating utility maps as described in Sec. IV-E, suitable locations for the spherical cameras have to be determined beforehand depending on the scene.

VI. Conclusion

In this paper, we proposed a novel approach for covering known scenes containing articulated objects that the robot has to manipulate for inspecting user-defined regions of interest. We presented algorithms for costmap estimation, path planning, and information gain estimation that run on GPUs. In this way, our system can parallelize the necessary computations to determine the next best view. We introduced a heuristic for estimating information gain maps based on spherical panoramic images and showed in simulation experiments that our approach enables a robot to successfully inspect home environments. Our algorithms run on GPUs for embedded systems and we show benchmark results for multiple device classes.

References

[1] A. Gonzalez. (2013) "Kitchen Nr 2". Blender model, available under a Creative Commons Attribution (CC-BY 3.0) license. [Online]. Available: `https://www.blendswap.com/blends/view/70272`

[2] P. J. Martín, R. Torres, and A. Gavilanes, "CUDA solutions for the SSSP problem," in *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 904–913.

[3] J. O'Rourke, *Art Gallery Theorems and Algorithms*. New York, NY, USA: Oxford University Press, Inc., 1987.

[4] C. F. Bissmarck, M. Svensson, and G. Tolt, "Efficient algorithms for next best view evaluation," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2015.

[5] J. Daudelin and M. Campbell, "An adaptable, probabilistic, next best view algorithm for reconstruction of unknown 3D objects," in *Proc. of the IEEE Intl. Conf. on Robotics & Automation (ICRA)*, 2017.

[6] C. Dornhege, A. Kleiner, and A. Kolling, "Coverage search in 3D," in *IEEE Int. Symp. on Safety, Security, and Rescue Robotics (SSRR)*, 2013.

[7] S. Oßwald, P. Karkowski, and M. Bennewitz, "Efficient coverage of 3D environments with humanoid robots using inverse reachability maps," in *Proc. of the IEEE Intl. Conf. on Humanoid Robots*, 2017.

[8] A. Davidson, S. Baxter, M. Garland, and J. D. Owens, "Work-efficient parallel GPU methods for single-source shortest paths," in *Proc. of the IEEE Int. Parallel and Distributed Processing Symp. (IPDPS)*, 2014.

[9] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2012.

[10] L. G. Fischer, R. Silveira, and L. Nedel, "GPU accelerated path-planning for multi-agents in virtual environments," in *Brazilian Symp. on Games and Digital Entertainment*, 2009.

[11] C. Camporesi and M. Kallmann, "Computing shortest path maps with GPU shaders," in *Proc. of the Int. Conf. on Motion in Games (MIG)*, 2014.

[12] S. Venkataraman, "Programming multi-GPUs for scalable rendering," in *Proc. of the GPU Technology Conference (GTC)*, 2012.

[13] J. McDonald, "Avoiding catastrophic performance loss: Detecting CPU-GPU sync points," in *Proc. of the Game Developers Conference (GDC)*, 2014.

[14] L. Bishop, C. Kubisch, and M. Schott, "High-performance, low-overhead rendering with OpenGL and Vulkan," in *Proc. of the Game Developers Conference (GDC)*, 2016.

[15] D. V. Lu, D. Hershberger, and W. D. Smart, "Layered costmaps for context-sensitive navigation," in *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, 2014.

[16] R. Guodong, "Jump Flooding Algorithm on graphics hardware and its applications," Ph.D. dissertation, National University of Singapore, 2007.

[17] M. Fredman and R. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," in *Annual Symp. on Foundations of Computer Science*, 1984.

[18] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *Journal of the ACM*, vol. 46, no. 3, pp. 362–394, 1999.

[19] L. R. Ford, "Network flow theory," RAND Corporation," Paper, 1956.

[20] R. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, pp. 87–90, 1958.

[21] N. Kaushik and A. Kaushik, "Extended Bellman Ford algorithm with optimized time of computation," in *Advances in Intelligent Systems and Computing*. Springer Singapore, 2016, pp. 241–247.