# Real-Time Dense Multi-Scale Workspace Modeling on a Humanoid Robot

René Wagner Udo Frese Berthold Bäuml

***Abstract*— Without a precise and up-to-date model of its environment a humanoid robot cannot move safely or act usefully. Ideally, the robot should create a dense 3D environment model in real-time, all the time, and respect obstacle information from it in every move it makes as well as obtain the information it needs for fine manipulation with its fingers from the same map.**

**We propose to use a multi-scale truncated signed distance function (TSDF) map consisting of concentric, nested cubes with exponentially decreasing resolution for this purpose. We show how to extend the KinectFusion real-time SLAM algorithm to the multi-scale case as well as how to compute a multi-scale Euclidean distance transform (EDT) thereby establishing the link to optimization-based planning.**

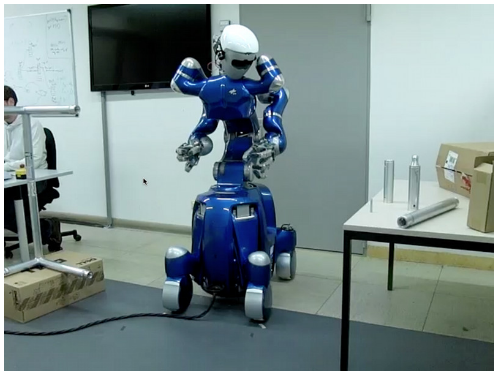
**We overcome the inability of KinectFusion's localization to handle scenes without enough constraining geometry by switching to mapping-with-known-poses based on forward kinematics. The latter is always available and we know when it is precise. The resulting map has the desired properties: It is computed in real-time ($7.5\,\mathrm{ms}$ per depth frame for a $(8\,\mathrm{m})^3$ multi-scale TSDF volume), covers the entire laboratory, does not depend on scene properties (geometry, texture, etc.) and is precise enough to facilitate grasp planning for fine manipulation tasks – all in a single map.**

## I. Introduction

The motivation for this work is grounded in the day-to-day operation of DLR's humanoid robot Agile Justin [2] in a research laboratory setting: Without a precise and up-to-date model of its environment the robot cannot move safely unless extra measures are taken, e.g., by coding environment information into experiment-specific code.

What we want is a way for the robot to create a dense 3D environment model in real-time, all the time, and without human interaction and respect obstacle information from it in every move it makes. Ideally, we want a single world model that is 1) generated at high frame rates in real-time 2) safe in the sense of accumulating only raw, dense depth/range sensor data without any post-processing steps (for hole-filling, etc.) and 3) detailed and precise enough to facilitate fine manipulation with the robot's fingers, i.e., to enable appropriate object detection and grasp planning. In previous work [11], we have demonstrated a prototype system that uses a custom KinectFusion [8] re-implementation to build a high-resolution surface model of the environment, computes the Euclidean distance transform (EDT) from this and uses the EDT as a structure function in an optimization-based motion planner (OMP). All processing steps in this work are

R. Wagner and B. Bäuml are with DLR Institute of Robotics and Mechatronics, 82234 Wessling, Germany. U. Frese is with Faculty 3 – Mathematics and Computer Science, University of Bremen, 28359 Bremen, Germany. R. Wagner is also with University of Bremen. Contact {rene.wagner,berthold.baeuml}@dlr.de, ufrese@informatik.uni-bremen.de.



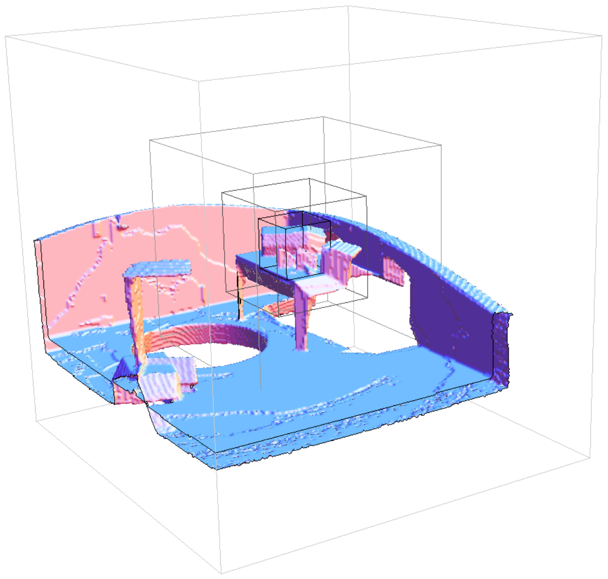


Fig. 1. DLR's humanoid Agile Justin [2] in our mobile manipulation scenario (top) and the multi-scale environment model (bottom) generated from Kinect depth measurements in real-time using a mapping-with-known-poses approach based on the forward-kinematics model.The environment model consists of four layers of concentric, nested voxel grid cubes. Each cube contains $256^3$ voxels. The voxel size doubles with each layer. The innermost, highest-resolution ($2\,\mathrm{mm}$) cube is positioned on the tabletop and encloses several objects to be manipulated (Fig. 8 depicts this volume in detail). The companion video shows the map building process in progress and how the KinectFusion SLAM algorithm fails in this scenario due to lack of constraining environment geometry.

real-time capable except for the algorithm [6] used for the EDT computation which could be replaced by the real-time alternative [5].

One might think that this would already satisfy the requirements above. Unfortunately, there were two significant drawbacks:

1) As GPU memory is presently still rather limited so is the map volume that can be covered. E.g., at a grid voxel size of $2\,\mathrm{mm}$ (which is necessary for fine

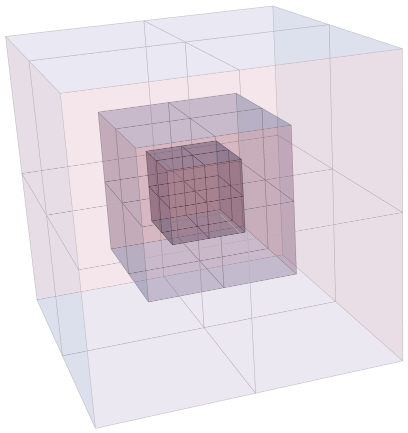


Fig. 2. Multi-scale grid consisting of three concentric, nested cube layers. Each layer has the same number of voxels ($2^3$ in this example). The voxel size, however, increases and thus the resolution decreases in the outer layers.

manipulation tasks) we can cover at most $(1.024\,\mathrm{m})^3$ with our current consumer-grade GPU.[1]

2) For the ICP-based [3] sensor tracking in KinectFusion to work the environment geometry must always constrain all six degrees of freedom (DoF) of the sensor pose. Whenever a DoF is unconstrained ICP will fail. KinectFusion usually detects this, but cannot recover from this situation. In some cases unconstraining geometry will go unnoticed leading to a gradual destruction of the map.

In this paper, we propose to solve the first issue by using a multi-scale model approach. Essentially, we employ a hierarchy of multiple grid layers with decreasing resolution (Fig. 2). By reducing the size of each grid to, e.g., $256^3$ we can store multiple grids in the same amount of memory that was previously occupied by a single, e.g., $512^3$ grid. The inner-most layer is then centered around the area we care the most about, e.g., an object the robot needs to grasp, and has a voxel side length of 2 mm, the next layer is twice as large and thus encloses the previous but at a resolution of only 4mm, the third at a resolution of 8mm and so forth. This allows us to easily model the entire laboratory on a single, consumer-grade GPU with a modified KinectFusion implementation in real-time as detailed below.

The second issue is a more fundamental one. We choose a very pragmatic approach to tackle it: The simple insight is that, on a (humanoid) robot, the Kinect sensor is not a freely moving camera. It is in fact firmly attached to the head of the robot. We know the robot's kinematics model and can use this in a mapping-with-known-poses approach. The tremendous advantage is that the forward kinematics model is always available, does not depend on any environment properties such as scene geometry (or texture), and we know exactly when the forward kinematics model is precise. In some situations it is very precise, most notably when only the head joints (pan/tilt) move. It turns out that very accurate models can be generated when this knowledge is taken into account. Note that this is not limited to just Agile Justin – it is very common for (humanoid) robots to have a very precise pan-tilt unit with high-resolution rotary encoders. A prerequisite to this sensor fusion is of course a precise calibration of the Kinect's external parameters (via an extension of our auto-calibration procedure [4]) as well as detailed knowledge of the timing of the sensors provided by our real-time-capable software middleware aRD [1].

The remainder of the paper is structured as follows. After a discussion of related work (II), we introduce our proposed multi-scale map data structure (III), show how KinectFusion needs to be modified to operate on it (IV), how to establish the link to optimization-based motion planning [9] by computing a multi-scale EDT and its gradient (V), discuss our experimental results (VI), and close with conclusions and future work (VII).

## II. Related Work

When it comes to dense, real-time mapping in indoor settings, the KinectFusion 3D SLAM system [8] is currently the state of the art. It tracks the sensor pose using a projective variant of the iterative closest point (ICP) algorithm [3] and represents the map as an implicit surface model in the form of a truncated signed distance function (TSDF) in a regular voxel grid. By performing virtually all computations on the GPU it manages to process the 30Hz dense depth data stream from the Microsoft Kinect depth camera in real time. Unlike other SLAM approaches, KinectFusion does not have any dedicated loop closure mechanism. However, this is often not needed when modeling a tabletop or other confined indoor scene with reasonably rich geometry. In the original algorithm, KinectFusion maps must fit into GPU memory.

Variants exist that store parts of the map in host (CPU) memory either in the same voxel format [10] or as a mesh representing the surface only [12]. The earlier approach is reported to have performance issues when map content is transfered (GPU-to-CPU memory bandwidth) and the volume stored on the GPU (where we also compute our obstacle avoidance path planning objectives) remains the same. The mesh representation of the latter approach loses free vs. unseen space information. In terms of applications, apart from our own previous work [11], KinectFusion has received surprisingly little attention as an environment modeling approach in the (humanoid) robotics community.

In contrast to this, the OctoMap library [7] has proven very popular. It does not provide any sensor tracking capabilities itself – given range scans or depth images and corresponding sensor poses it simply stores these measurements in a map data structure that is essentially a probabilistic 3D occupancy grid except that it uses an octree instead of a regular grid to store voxels (in leaf nodes). Each voxel holds the log-likelihood of it being occupied to accumulate occupancy and free space information and, optionally, extra data such as RGB values. The model granularity ends at leaf node

[1]We currently use a 2GB NVIDIA GTX 680. A 4GB model we have evaluated showed stability issues.

voxels, i.e. there is no sub-voxel modeling involved as in KinectFusion.

As a particular advantage of their approach over KinectFusion style maps, the OctoMap authors claim [7] that their map data structure can distinguish free from unseen space while KinectFusion's implicit surface data structure cannot. As we have previously shown [11, §V], this is in fact very well possible with KinectFusion as TSDF voxels representing unseen space have zero weights while free space voxels have non-zero weights and a TSDF value greater than 0 and can thus be easily distinguished.

A key difference between OctoMap and KinectFusion style maps is that OctoMap explicitely mantains occupancy likelihood values for each voxel (leaf-node). In practice, this means that noisy binary information (occupied vs. free space) in a certain fixed voxel location is averaged over time. KinectFusion's TSDF, on the other hand, models the exact same quantity that depth sensors measure – the distance to surfaces. On each update it performs weighted averaging of distance values to the implicit surface. Each new measurement refines the location of the implicit surface perpendicularly to it rather than just accumulating evidence in the vicinity of the surface as OctoMap does. Since noise in depth/range sensors primarily affects the depth/range this leads to a very quick smoothing effect in KinectFusion (that actually achieves sub-voxel resolution perpendicularly to surfaces).

This smoothing effect is the reason why the TSDF map yields significantly better models when used with the Kinect sensor than OctoMap. To understand this it is important to note that the Kinect sensor is essentially a stereo camera – it measures pixel disparities not distance. Thus, while the image resolution is very high, the disparity discretization is $^1/_8$ pixel and leads to significant discretization errors that increase quadratically with distance. In an OctoMap model this will typically lead to thick borders around objects while in KinectFusion's TSDF map the discretization errors will be "averaged away" leaving just the surface itself (as soon as the sensor is moved). As for lateral measurement errors, in the particular case of the Kinect sensor, measurement errors parallel to surfaces are predominantly "holes" in the depth image which are rare and typically only happen when viewed from very specific angles. Thus, holes in the model are typically filled very quickly as the sensor is moved.

In terms of computing time, OctoMap is reported [7] to take 51ms to traverse 4 million leaf nodes. Updates are even slower. In our KinectFusion implementation it takes about 1.5ms to traverse and update 16 million ($256^3$) voxels.

The two approaches are, however, not mutually exclusive. In fact, there is a KinectFusion variant that uses an octree data structure to represent its map [14]. In the usual case it manages to compensate for the octree data structure overhead by skipping empty space more efficiently. We decided against this approach as it adds another step that is not guaranteed to be constant time on top of the already non-constant time (scene dependent) ray-casting step in KinectFusion which can cause the time budget to be exceeded in scenes involving a lot of clutter, i.e. a lot of space near surfaces which requires a small ray traversal step size.

Concerning robustness, it was very recently [13] proposed to fuse the geometry-based ICP result by KinectFusion with visual odometry. While this approach alleviates problems caused by KinectFusion's reliance on environment geometry constraining the sensor pose it does not fully solve the problem in environments where there is both little clutter and little texture on surfaces. In our laboratory setting this is a fairly frequently occuring issue: Large areas are dedicated to people walking or robots driving around so that a down-facing Kinect sees nothing but a (nearly) perfectly flat homogeneously colored floor.

It should be noted that both OctoMap and KinectFusion's TSDF map as such assume an inherently static world – changes will only gradually propagate through the evidence accumulation of OctoMap and the weighted averaging of KinectFusion's TSDF. Dynamic environments thus require extra measures to be taken. The most common source of changes in the field of view of the robot is the robot itself. Given a volumetric robot model we simply mask out the robot in the input depth image. If the robot moves an object this is also fairly easy to handle: The location of the object needs to be known for grasping anyway so that one could simply segment it in the depth image and in the map. Any changes that are not under the control of the robot are more difficult to handle but not impossible, e.g., the original KinectFusion authors report [8] that an extension of KinectFusion segments moving objects from a static background and tracks their motion over time.

## III. Multi-Scale TSDF

KinectFusion uses a truncated signed distance function (TSDF) to represent the 3D geometry of the environment as an implicit surface. The TSDF is stored in a regular, dense 3D voxel grid where each voxel holds the signed distance to the nearest surface truncated to $[-\mu, \mu]$ and further normalized to $[-1, 1]$. The surface lies at the zero-crossing of the distance values.

Storing this dense voxel grid requires a lot of memory. At four bytes per voxel, a single $512^3$ voxel grid consumes 512MB. Since we also store EDT grids of similar sizes on the GPU this constitutes the limit of what the current consumer-grade GPU generation can hold in GPU memory. Thus, the tradeoff is to decide for either high resolution or a large volume.

Ideally, however, we would like to have both and the idea behind the multi-scale TSDF grid data structure is just that: We move to $256^3$ voxels per grid so that the same memory can hold 8 of these smaller grids. We then use different voxel sizes in each grid to form different layers essentially viewing the world at different resolutions.

We have chosen to arrange these layers as concentric, nested cubes. All cubes have the same number of voxels in all dimensions but the side length of the voxels is doubled with each additional grid layer as illustrated in Fig. 2. The regular placement and cubic shape is not strictly required but greatly simplifies the implementation. It is, however, required that in all but the inner-most layer voxels enclose an integral number of voxels of the next-finer grid layer.

Such a multi-scale approach is also very appropriate in a mobile manipulation scenario as it leads to the following hierarchy of layers 0 to 3:

- Layer 0 (2 mm resolution, $(0.512\,\mathrm{m})^3$ volume) models the immediate surroundings of the object to be manipulated/grasped.
- Layer 1 (4 mm resolution, $(1.024\,\mathrm{m})^3$ volume) models the immediate workspace center, e.g. the tabletop.
- Layer 2 (8 mm resolution, $(2.048\,\mathrm{m})^3$ volume) models the immediate surroundings of the workspace center should motion of the robot base be necessary.
- Layer 3 (16 mm resolution, $(4.096\,\mathrm{m})^3$ volume) models the "room".

Thus, we can provide a useful whole-room model in just 4 layers (maybe 5 for very large rooms/laboratories).

Note that one could alternatively center grids around the sensor as the precision of the Kinect decreases with distance and possibly combine this with a 3D ring buffer data structure like in [12]. However, this is beyond the scope of our application.

As for the actual in-memory data structure of this multi-scale grid, it essentially needs to support two basic operations: Given a world point find the voxel that represents it. And, conversely, iterate over all voxels and determine their world location. We can enable both by simply storing some extra metadata for each grid layer: the metric voxel size $l_k$, the integral grid dimensions $L$ in voxels and the Cartesian coordinates of its origin $o_k$, such that

$$l_k = 2^k \cdot l_0, \tag{1}$$

$$s_k = L \cdot l_k, \tag{2}$$

$$o_k = {}^1\!/_2 \cdot (s_{kmax} - s_k), \tag{3}$$

where $l_0$ is the voxel size of the highest resolution layer and $kmax$ the index of the lowest-resolution layer. This is sufficient to determine which grid layer is "responsible" for a given world point by simply iterating over each grid layer $k$ starting at the outer-most one and checking whether the point $(x, y, z)^T$ is outside the next finer layer $k-1$, i.e. whether with a margin $m := l_k$

$$\begin{aligned} & x \le o_{k-1} + m \vee x > o_{k-1} + s_{k-1} - m \\ \vee\ & y \le o_{k-1} + m \vee y > o_{k-1} + s_{k-1} - m \\ \vee\ & z \le o_{k-1} + m \vee z > o_{k-1} + s_{k-1} - m \end{aligned} \tag{4}$$

and then performing the actual lookup within the layer as usual, e.g., for the x-ordinate this yields the index

$$xidx = \lfloor (x - o_k)/l_k \rfloor. \tag{5}$$

Iteration over all voxels can trivially happen by iterating over all layers in an outer loop and then over all voxels within each layer as usual. The world-location of a voxel is simply, e.g., for the x-ordinate:

$$x = o_k + xidx \cdot l_k. \tag{6}$$

As we will show below, this modified data structure is applicable to KinectFusion and mapping with known poses.

## IV. Multi-Scale KinectFusion

Apart from minor pre- and post-processing steps, KinectFusion [8] consists of three main components which are called in an infinite loop:

1) The ray-casting step computes a synthetic point cloud (with corresponding surfaces models) that would have been measured by a depth sensor given the current TSDF model and estimated sensor pose.
2) The iterative closest points (ICP) step then takes a new depth measurement from the Kinect sensor and adjusts the sensor pose to minimize the squared point-to-surface errors between the actual and the expected (ray-cast) depth measurements.
3) The map update step then takes this estimated sensor pose and updates the TSDF according to the Kinect depth measurements.

Obviously, only the first and the last step operate on the map data structure and thus only these need to be modified to make KinectFusion operate on multi-scale TSDF grids as detailed below.

### *A. Ray-Casting*

The general idea behind the ray-casting in KinectFusion is the following: Starting at the optical center step through the map along a line until an intersection with a surface is found. The distance to this intersection point is the expected depth measurement. The surface normal is computed from the TSDF gradient. In priciple, this is easy to implement for the multi-scale TSDF: At every point considered along the ray take its world location and perform a multi-scale grid lookup to find the "responsible" voxel with the highest resolution (smallest voxel size).

However, two modifications are needed. The ray-traversal step size in the original algorithm is slightly less than $\mu$ (much larger than the voxel size) while traversing free space and less than the voxel size near surfaces (voxels with weight $> 0$ and TSDF $< 1$). $\mu$ is chosen to reflect the depth sensing precision of the sensor (with a Kinect and indoor scenes a good choice is $\mu = 3\,\mathrm{cm}$). Surfaces must be at least $2\mu$ apart to be considered separate. Now, the problem is that as grid layers become coarser their size increases and so does the length of the longest ray through them (i.e. the diagonal $\sqrt{3} \cdot s$ for an $s^3$ cube). With a $(4\,\mathrm{m})^3$ or $(8\,\mathrm{m})^3$ cube this would take prohibitively long.

Instead, we use a different $\mu$ for each grid layer and compute it from its voxel size. We use

$$\mu_k = 15 \cdot l_k \tag{7}$$

which was tuned to yield a sensible $\mu$ at the highest grid resolution 2 mm. This also means that, in coarser grid layers, surfaces need to be farther away from each other (measured along a ray from the sensor to the surface) to be treated as separate. However, this simply reflects the fact that the discretization is also more coarse and thus leads to overall consistent results.

The second modification is a consequence of the first. Once a zero crossing has been located by overstepping it, its exact location is refined by trilinearly interpolating the TSDF values at the previous and the current point on the ray and determining the zero crossing by solving for zero the 2D line equation through these two values. Unfortunately, the TSDF representation stores the normalized truncated distance, i.e., instead of the range $[-\mu, \mu]$ it stores values in the range $[-1, 1]$. Thus, as per modification to compute $\mu$ from the voxel size in (7), the same value has different meanings at different grid resolutions and, hence, zero crossings near resolution boundaries would not be computed correctly. This is easy to fix, though: We simply multiply the TSDF value with $\mu_k$ before trilinear interpolation.

With these changes applied, our KinectFusion implementation can build multi-scale models with 4 layers ($(4\,\mathrm{m})^3$ overall volume) at frame rate in typical scenes and slightly below frame rate ($\approx 40\,\mathrm{ms}$ per frame) in cluttered scenes due to the scene-dependent computing time of the ray-casting step.

### B. Map Update

The map update step iterates over all voxels and projects each voxel into the image based on the sensor pose and a pinhole camera model. It then updates the TSDF distance value by comparing the distance from the optical center to the voxel to the value at the same pixel coordinate in the current Kinect depth image.

The multi-scale map update is trivial to implement: We can treat each multi-scale grid layer as a separate single-scale map and apply the original update procedure. This means that some of the physical volume is represented multiple times but simplifies the code and allows us to perform per-layer operations independently when computing the EDT (V).

Note that the map update step is independent from the sensor tracking (ICP) parts of KinectFusion and can be used as a standalone module by passing in the sensor pose obtained by other means (mapping with known poses). We will show results using rotary encoders and a forward kinematics model for this purpose in VI.

## V. Multi-Scale EDT and EDT-Gradient Computation

The Euclidean distance transform (EDT) determines for each voxel the signed distance to the closest free vs. occupied/unknown space boundary. It can thus be used as a structure function to encode obstacle information and thereby establishes the link from environment modeling to optimization-based planning [9]. We have previously shown [11] that the TSDF already models free vs. occupied/unknown space so that the EDT can also be directly computed on the GPU. Thus, we just need to lift the EDT, EDT gradient and objective computation to the multi-scale case so the planner can work with multi-scale models, too.

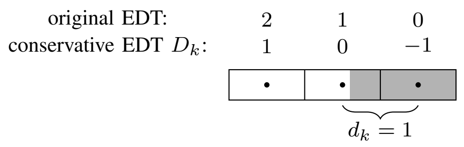


Fig. 3. The occupancy of a voxel is discretized based on its center potentially making obstacles (shown in gray) seem farther away than they are. We can fix this, i.e., make the original EDT conservative by subtracting the diagonal $d_k$. For simplicity, a 1D EDT is shown. In 3D, $d_k = \sqrt{3} \cdot l_k$.

### A. Euclidean Distance Transform (EDT)

Essentially, we follow the same approach as with the multi-scale TSDF except that the grid now stores floating point numbers instead of TSDF voxels. The EDT is first computed for each layer independently based on the discretized occupancy of the corresponding TSDF layer. Since a TSDF voxel stores the (truncated) distance of its center to the nearest surface we get the EDT with respect to the center of each voxel if we apply our modified version [11] of the EDT algorithm from [6]. From this, we subtract $d_k = \sqrt{3} \cdot l_k$ to yield the conservative EDT $D_k$ as illustrated in Fig. 3. Also, at this stage, space outside each EDT layer is treated as unknown space, i.e., like an obstacle. Thus, the inner layers now represent a safe, but very limited view of the world and we want to propagate the extra knowledge the outer layers have about the world to the inner layers.

To achieve this, we successively merge two consecutive layers $k$ and $k-1$ starting at the outer-most (lowest-resolution) layer until we reach the inner-most layer at $k-1 = 0$. The merge operation first needs to apply trilinear interpolation to the coarse layer $k$. Otherwise, we would not gain a smooth EDT gradient as is needed for optimization-based planning. As for the interpolation error, consider the worst case interpolation – at the center $x$ of the $l_k$-sized cube spanned by the interpolation points $p_i$. Here, with $d = \sqrt{3} \cdot l_k$ being the diagonal of the interpolation cube in the coarser layer and $\operatorname{dist}(x)$ denoting the signed distance before interpolation the following holds:

$$\operatorname{dist}(x) \geq \operatorname{dist}(p_i) - \tfrac{d}{2}. \tag{8}$$

And thus with interpolation weights $\lambda_i$ also:

$$\begin{aligned}\operatorname{dist}(x) &\geq \sum_i \lambda_i \operatorname{dist}(p_i) - \sum_i \lambda_i \tfrac{d}{2} \\ &\geq \sum_i \lambda_i \operatorname{dist}(p_i) - \tfrac{d}{2}.\end{aligned} \tag{9}$$

Thus, we can compensate for the interpolation error by subtracting $\frac{d}{2}$ to get a lower bound on the distance value. This lower bound can now be combined with the value from the finer layer $k-1$ using the $\max$ operation as both are valid lower bounds. Hence, with $D_{k-1}(u)$ being the EDT value at layer $k-1$ and voxel $u$, and $\operatorname{Interpolate}(D_k, u)$ the trilinearly interpolated EDT from the next-coarser layer $k$, the full iterative merge operation reads as

$$\begin{aligned}D_{k-1}(u) \leftarrow \max(\operatorname{Interpolate}(D_k, u)) - \tfrac{d}{2}, \\ D_{k-1}(u)).\end{aligned} \tag{10}$$

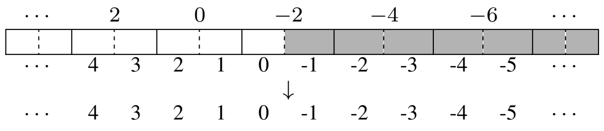


Fig. 4. Merging of the EDT layer $k$ (lower-resolution values, top) into layer $k-1$ (higher-resolution, original values in the middle, result at the bottom): In fully overlapping parts of the volume the $\max$ operation retains the finer structure from layer $k-1$. For simplicity a 1D EDT is shown. Obstacle voxels are marked in gray.

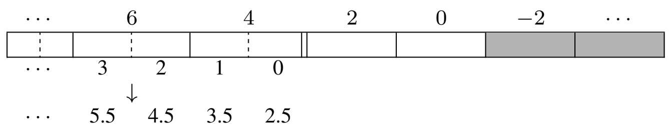


Fig. 5. Merging of the EDT layer $k$ (lower-resolution values, top) into layer $k-1$ (higher-resolution, original values in the middle, result at the bottom): At the border of the inner layer $k-1$ (marked by the $||$) the $\max$ operation propagates the extra context known by layer $k$ into layer $k-1$. Note how the inner layer originally assumes space outside its volume to be an obstacle in order to achieve conservativeness and that the values of layer $k$ are conservatively interpolated according to (10).

The resulting EDT is not only conservative but represents a bound that is tight in two important ways: Firstly, as illustrated in Fig. 4 the $\max$ operation ensures that deep within the inner layer its finer structure is retained. Secondly, at the border of the inner layer $k-1$, the EDT values assume that anything outside the volume of the inner layer is an obstacle so that the inner layer never overestimates the true distance to an obstacle. But the outer layer $k$ sees more context and as illustrated in Fig. 5 this is propagated as intended by the $\max$ operation.

Finally, it should be noted that in contrast to our previous work [11] the EDT grids are not downsampled but use the same grid size of $256^3$ voxels per layer as the TSDF grid.

## B. EDT Gradient

Once all EDT layers have been updated with the (interpolated) maximum over all overlapping parts, the EDT gradient computation is a pure per-layer operation and no modification to our original 3D Sobel-filter-based implementation [11] is needed.

## C. Use in Optimization-Based Planning

To use the multi-scale version of the EDT and its gradient in optimization based planning, we have simply modified the objective function and gradient computation from our previous work [11] to always perform the lookup of the EDT and EDT gradient value in the highest resolution grid wherever there is overlap between grid layers as per (4) and (5).

## VI. Experiments

We have integrated the multi-scale data structures and algorithms discussed above into our software stack [11]. The system runs online on the real robot.

For the experimental evaluation we have chosen a typical laboratory setting. Agile Justin is positioned in front of a tabletop workspace as illustrated in Fig. 1. The highest-resolution grid layer is positioned as if it was resting on the table aligned with the front-right corner of the table as indicated by the inner-most wire-frame cube at the bottom of Fig. 1. Several objects to be manipulated each of different size are placed on the table.

For the map generation, Justin executes a sweep with its head and torso pan joints. The sweep starts with the right hand side of the table in view of the Kinect sensor. Justin then sweeps to the left over the rest of the table and beyond until a laboratory corner, wall and door come into view. It then performs a full 360° sweep to the right until the door is in view again. Out of a variety of data sets we have deliberatively chosen this one as it is both representative of the environment Justin operates in and because different stages of the data set trigger very specific behavior in KinectFusion (rich geometry fully constraining the sensor pose, drift due to one unconstrained DoF, abort due to too many unconstrained DoF; see below for details).

The resulting data set is processed once by the full multi-scale KinectFusion algorithm and once by a mapping-with-known-poses approach. The latter uses the multi-scale mapping part but gets the sensor poses from the forward kinematics model and precise rotary encoder readings.

To demonstrate the quality of the generated maps we use ray-casts of the model from the same pose as the (estimated) sensor pose employing the same camera parameters as the Kinect except for a wider-angle focal length. This means that the part of the scene that is visible to the Kinect can be easily identified in the first frame and always remains within the same rectangle at the center of the rendered image. The renderer does not apply any extra smoothing (Phong shading, etc.) but uses the raw surface normals for lighting thus exposing even the smallest dents in the map.

Selected frames from the resulting sequence for the full KinectFusion algorithm are depicted in Fig. 6. The full sequence is shown in the companion video along with an external camera view. While the rich geometry on the tabletop is in view KinectFusion works well generating a nicely smooth, detailed map. As soon as only the wall is in view it gradually destroys the map. Note how the bottom door hinge appears multiple times in the map and how the left corner of the table is cut off. This happens because not all degrees of freedom (DoF) of the sensor pose are constrained by the sensor data in this case: With only the wall and parts of the ground in view the pose estimate can be shifted left or right without resulting in any (significant) point-to-surface errors. During the reverse sweep, KinectFusion manages to recover while the table is in view – the accumulated drift was small enough for ICP to be able to snap in again. To the right of the table only the floor is in view leaving too many DoF unconstrained so that the corresponding checks in the ICP code signal a failure and KinectFusion aborts.

The same frame sequence is shown for the multi-scale mapping-with-known-poses case in the two top rows of Fig. 7, the bottom row contains selected frames from the remainder of the sweep, i.e., after KinectFusion has already aborted. For a pure mapping-with-known-poses approach the results show remarkably good precision. Even the small bolt

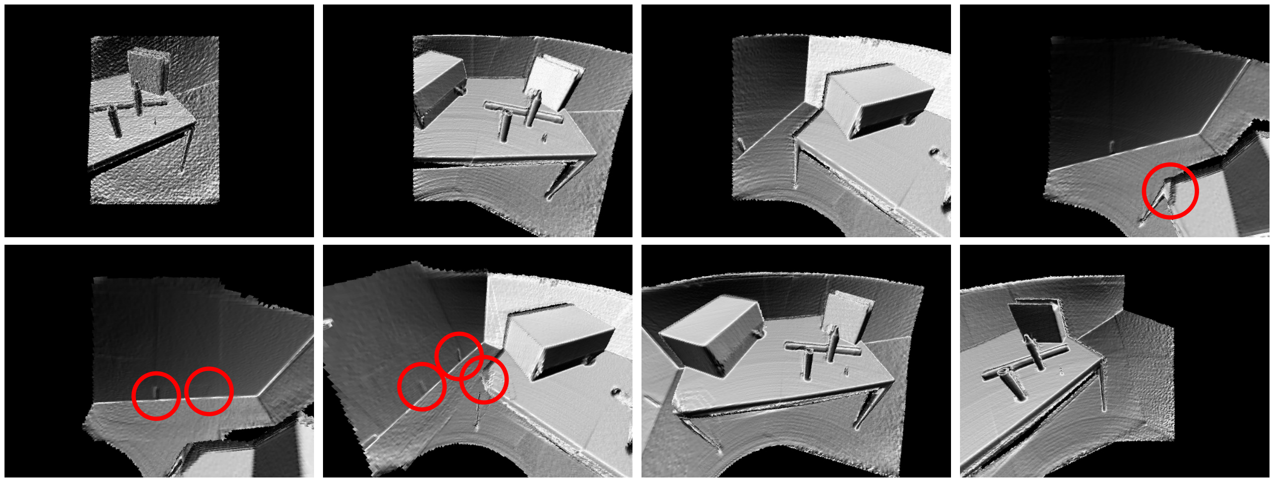


Fig. 6. Full sweep of the robot's workspace using head and torso pan-axes (top-left to bottom right): KinectFusion works fine while the table is in view (image 1,2,3), intermittently leads to an inconsistent model to the left of the table (image 4,5,6; note the duplicate door hinges and the chopped-off corner of the table), partially recovers on the return sweep across the table (image 6,7,8), but aborts when the scene geometry no longer constrains sensor pose at all to the right of the table (just after image 8). See text for details and also the companion video.

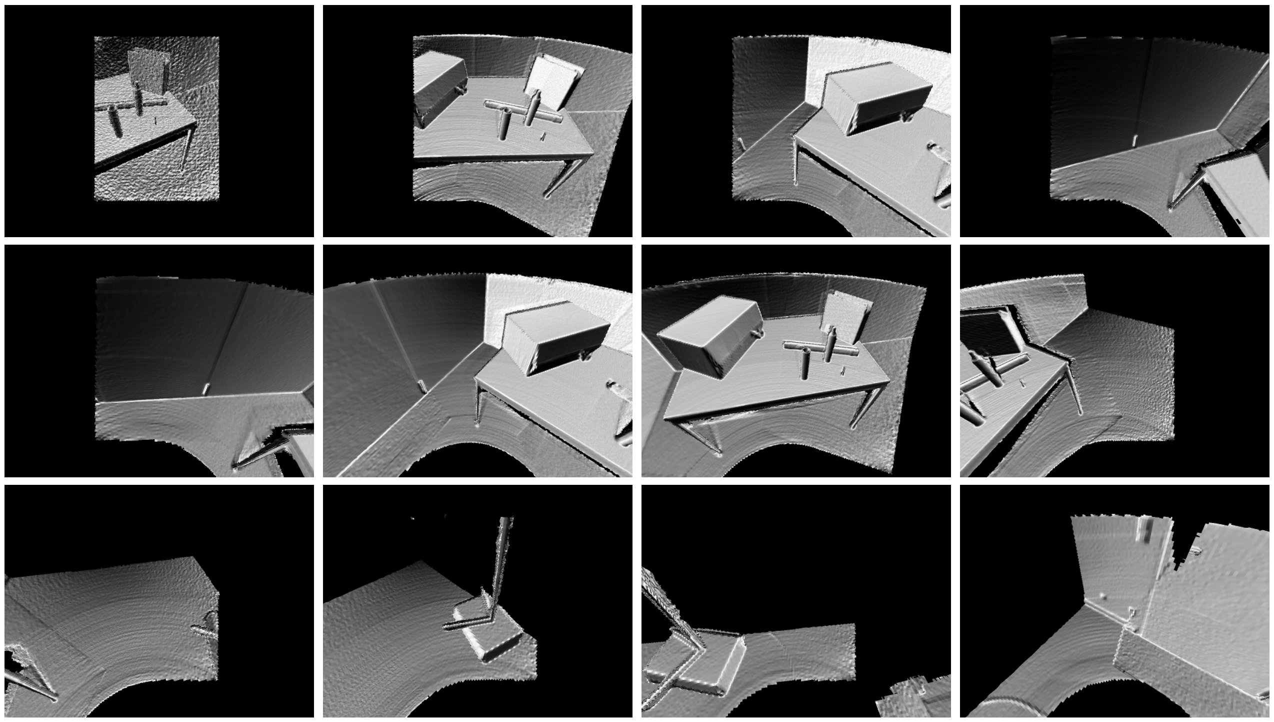


Fig. 7. Full sweep of the robot's workspace using head and torso pan-axes: Mapping-with-known poses based on forward kinematics leads to a consistent model throughout the entire sweep. See the companion video for a direct split-screen comparison with KinectFusion.

on the table with a diameter of just 0.5cm at the top and about 1cm at the bottom does not wash out in the model after the forward and reverse sweep. The table is nearly perfectly flat and the map is correctly aligned where the left-most and right-most parts of the sweep overlap (Fig. 7, bottom-right).

Contour plots of the EDT are depicted in Fig. 8. These show two things: a) The high-resolution in the inner-most grid is needed for grasp planning with small objects such as the bolt which washes out at the lower resolutions. b) Our multi-scale TSDF mapping-with-known-poses approach actually delivers the precision required to, e.g., localize objects or perform grasp planning.

A comparison of the per-frame computing times of the multi-scale mapping versus our previous single-scale implementation [11] is given in Table I.

## VII. Conclusions and Future Work

In this paper we have introduced the multi-scale TSDF map data structure and modified the KinectFusion algorithm to work with it retaining its real-time SLAM capabilities. We have also (re-)established the link to optimization-based

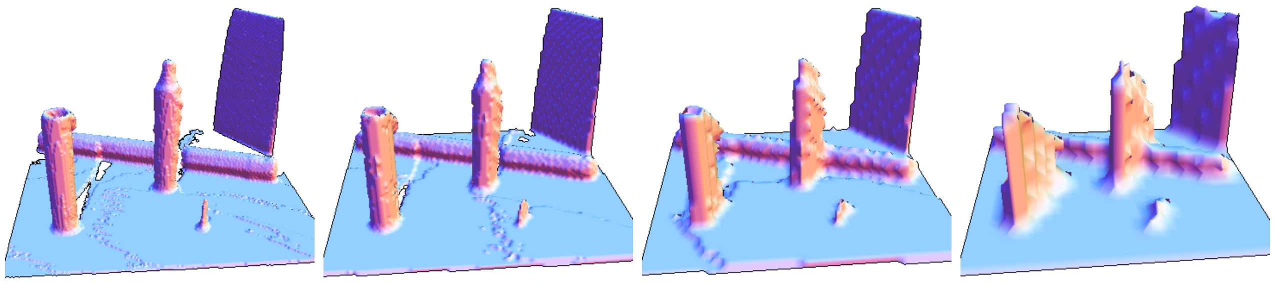


Fig. 8. The same multi-scale EDT model at different resolutions (decreasing exponentially from 2 mm on the left to 16 mm on the right) generated using mapping-with-known poses and the forward kinematics. The contour plots focus on the workspace center on the tabletop (lower-resolution grid layers are cut off where the highest resolution ends) with several objects to be manipulated. It is obvious that grasp planning would require the highest resolution for the bolt at the front.

TABLE I
PER-FRAME COMPUTING TIMES

| Step | Multi-Scale [ms] (four $256^3$ layers) | Single-Scale [ms] (one $512^3$ layer) |
|---|---|---|
| Ray-Casting & ICP | 16-33 | 19-23 |
| Map Update | 6 | 8 |

motion planning by computing the multi-scale Euclidean distance transform (EDT) as a structure function encoding obstacle information.

The multi-scale approach allows us to represent views of the world starting with a high resolution but a small volume all the way to very large volumes albeit at a lower resolution. Since these views are maintained simultaneously we get the benefits of both ends of this spectrum all within the still very limited GPU memory. Although KinectFusion benefits from the added volume of the coarse multi-scale layers and the extra resolution of the fine layers it is still very dependent on the scene geometry and fails if this does not constrain the sensor pose sufficiently.

It turns out that if we take just the mapping part of our multi-scale KinectFusion variant in a mapping-with-known-poses approach based on forward kinematics we get maps all the time independently from the scene. The result has the desired properties that it is computed in real-time (1.5 ms per TSDF grid layer and depth frame), covers the entire laboratory, does not depend on scene properties (geometry, texture, etc.), and is precise enough to facilitate grasp planning for fine manipulation tasks in the center of the workspace – all in a single map.

In future work, we intend to investigate fusion of KinectFusion (ICP) pose estimates and odometry information possibly following a similar approach as [13] to allow Agile Justin to drive around. Odometry alone is insufficient for mapping with known poses over longer distances.

## ACKNOWLEDGEMENTS

This work was partly supported under DFG grant SFB/TR 8 Spatial Cognition. We thank O. Birbach for performing the Kinect calibration and T. Hammer for preparing the graphics in Fig. 2.

## REFERENCES

[1] B. Bäuml and G. Hirzinger. When hard realtime matters: Software for complex mechatronic systems. *Robotics and Autonomous Systems*, 56(1):5–13, 2008.

[2] B. Bäuml, F. Schmidt, et al. Catching Flying Balls and Preparing Coffee: Humanoid Rollin'Justin Performs Dynamic and Sensitive tasks. In *IEEE Int. Conf. on Robotics and Automation*, 2011.

[3] P. J. Besl and N. McKay. A method for registration of 3-D shapes. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 14(2):239 – 256, 1992.

[4] O. Birbach, B. Bäuml, and U. Frese. Automatic and self-contained calibration of a multi-sensorial humanoid's upper body. In *IEEE Int. Conf. on Robotics and Automation*, 2012.

[5] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan. Parallel banding algorithm to compute exact distance transform with the GPU. In *ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, 2010.

[6] P. F. Felzenszwalb and D. P. Huttenlocher. Distance transforms of sampled functions. Technical Report TR2004-1963, Cornell University, 2004.

[7] A. Hornung, K. M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 34(3):189–206, 2013.

[8] R. A. Newcombe, S. Izadi, et al. KinectFusion: Real-time dense surface mapping and tracking. In *IEEE Int. Symposium on Mixed and Augmented Reality*, 2011.

[9] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *IEEE Int. Conf. on Robotics and Automation*, 2009.

[10] H. Roth and M. Vona. Moving volume KinectFusion. In *British Machine Vision Converence*, 2012.

[11] R. Wagner, U. Frese, and B. Bäuml. 3D modeling, distance and gradient computation for motion planning: A direct GPGPU approach. In *IEEE Int. Conf. on Robotics and Automation*, 2013.

[12] T. Whelan, J. B. McDonald, M. Kaess, M. F. Fallon, H. Johannsson, and J. J. Leonard. Kintinuous: Spatially extended KinectFusion. In *RSS Workshop on RGB-D: Advanced Reasoning with Depth Cameras*, 2012.

[13] T. Whelan, H. Johannsson, M. Kaess, J. Leonard, and J. McDonald. Robust real-time visual odometry for dense RGB-D mapping. In *IEEE Int. Conf. on Robotics and Automation*, 2013.

[14] M. Zeng, F. Zhao, J. Zheng, and X. Liu. A memory-efficient kinectfusion using octree. In S.-M. Hu and R. R. Martin, editors, *Computational Visual Media*, volume 7633 of *Lecture Notes in Computer Science*, pages 234–241. Springer, 2012.