Real-time 3D Reconstruction at Scale using Voxel Hashing (Nießner et al.)

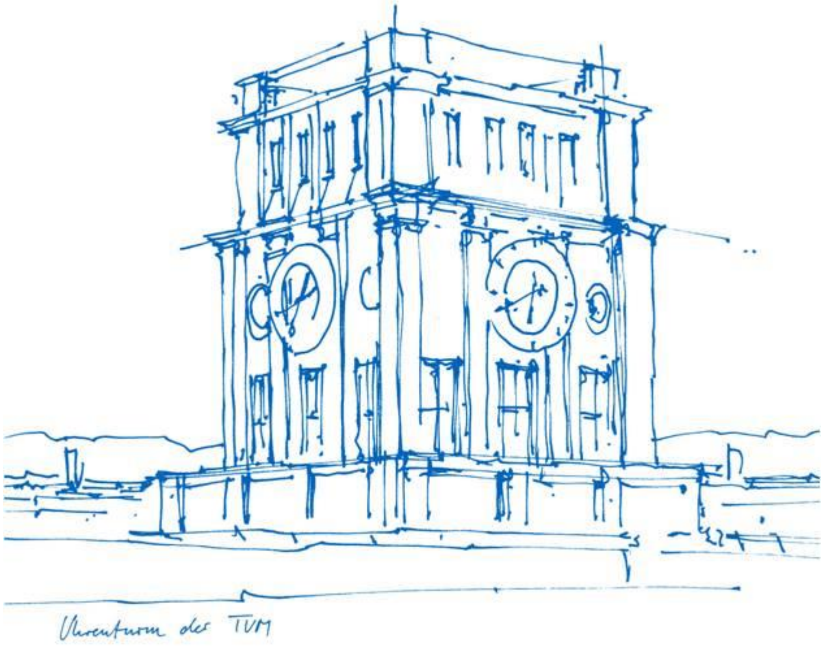
Lukas Huber

Supervisor: Christiane Sommer

The Evolution of Motion Estimation and Real-time 3D Reconstruction (Seminar)

Technische Universität München

Garching, 08 December 2020



Introduction

Agenda

- Introduction
- Overview
- Signed Distance Functions
- Foundation: Curless and Levoy
- Method description
- Results
- Personal comments

Overview

There exist two main types of reconstruction algorithms:

Offline reconstruction algorithms

- Higher quality scans
- More time consuming

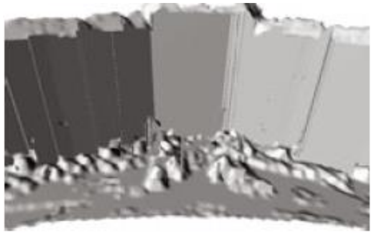
Online reconstruction algorithms

- Often lower quality
- Reconstruction is iteratively refined
- Results can directly be observed

Overview: Online approaches

- Point-based methods
  - Unstructured representations
  - Fail to reconstruct connected surfaces

- Height map methods
  - Efficient compression of connected surfaces
  - Trouble with complex surfaces



- Volumetric methods
  - Represent surface with implicit function
  - Inefficient voxel grids
  - Quality or scale

Volumetric reconstruction at scale

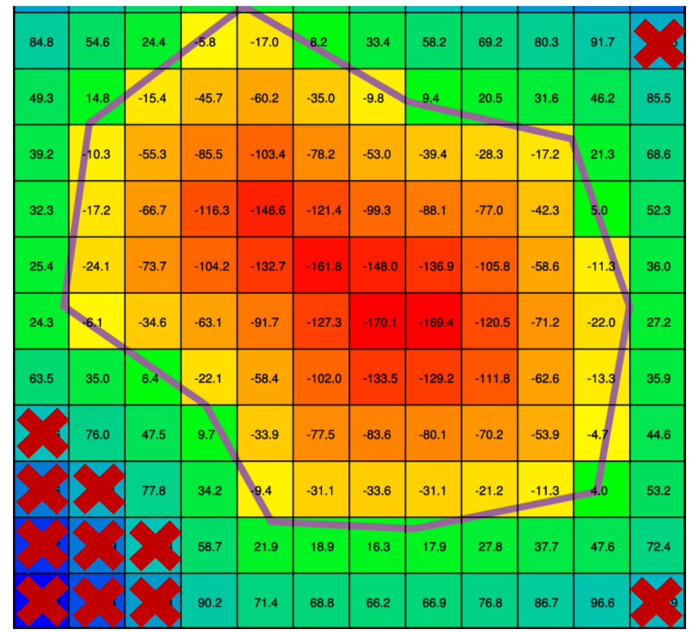
Combining fine quality and scale restricted by GPU memory

- Moving volume approaches
  - Streams voxel from GPU to host
  - Only one-way since surface is compressed to mesh on host
  - Active volume still needs to fit GPU memory.

- Sparse voxel octrees
  - Sparse representation of voxels in tree structure
  - Resolution increases with depth
  - Complex structure with overhead and only limited gains in scale

Recap: Signed Distance Functions

- Voxel store signed distance from its center to the observed surface
- Positive distances indicate voxels in the front, negative behind and zero on the surface
- Truncated Signed Distance Functions only store distances near the surface

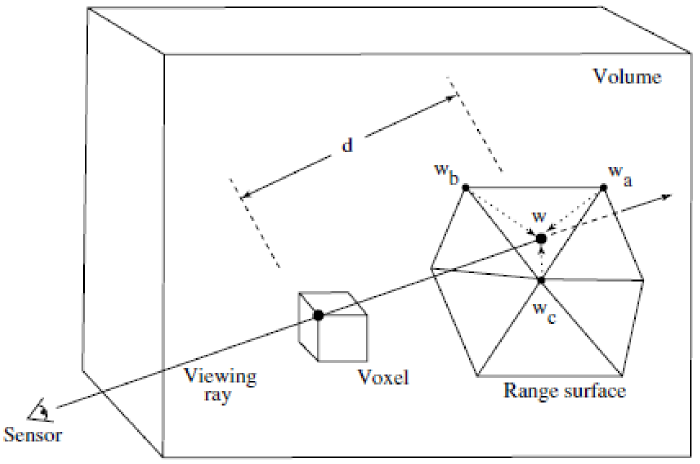


Truncated SDF with cutoff 100

Method foundation

Based on Curless and Levoy (1996)

- Sensor input
  - Input is a sequence of n aligned depth images
  - Weights $w_i(\boldsymbol{x})$ indicate sensor uncertainty



- Voxel grid generation
  - Convert the depth map to a triangle mesh
  - Raycast from sensor
  - Linearly interpolate the weights w_a, w_b, w_c to yield $w_i(\boldsymbol{x})$
  - Update SDFs based on ray

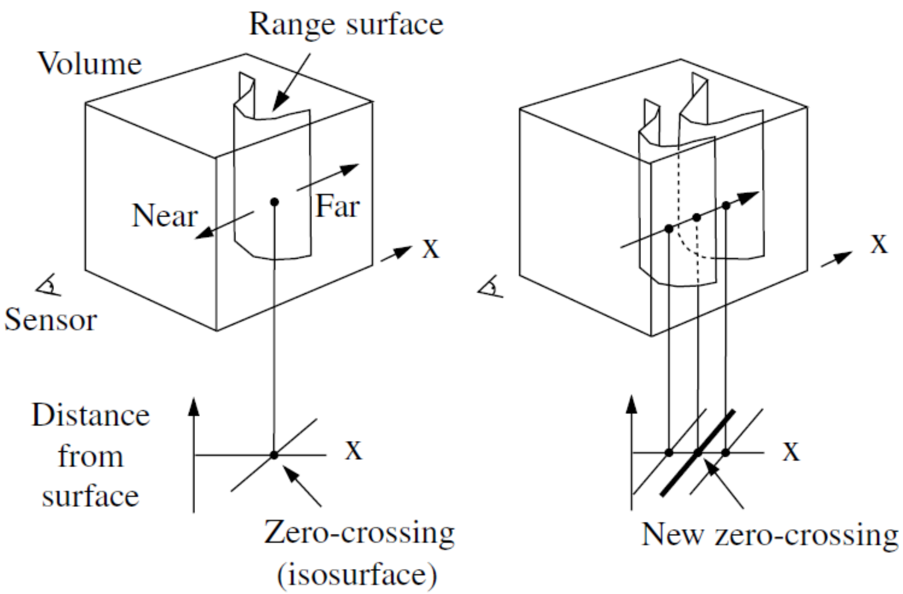
Method foundation

Cumulative SDF

$$D(\boldsymbol{x}) = \frac{\sum w_i(\boldsymbol{x}) d_i(\boldsymbol{x})}{\sum w_i(\boldsymbol{x})}$$

Cumulative weight

$$W(\boldsymbol{x}) = \sum w_i(\boldsymbol{x})$$



Method: Overview

Problem: Most of the voxel grid is either free or unobserved space

→ Create data structure that makes use of sparse Truncated SDF

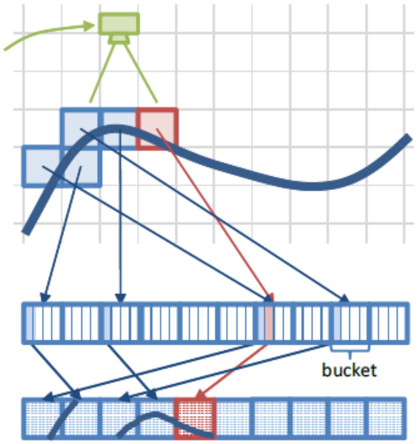
- Use a spatial hashing scheme as a fast and lightweight data structure
  - Compress TSDF and maintain resolution
  - Efficient insertion, update, deletion and resolution of collisions
  - Bi-directional streaming between host and GPU
  - Easy extraction of isosurfaces using raycasting

Method: Data structure

- The grid is subdivided into small regular voxel grids, called voxel blocks
  - Only allocated around surface

```
struct HashEntry {
  short position[3];
  short offset;
  int   pointer;
};
```

```
struct Voxel {
  float sdf;
  uchar colorRGB[3];
  uchar weight;
};
```

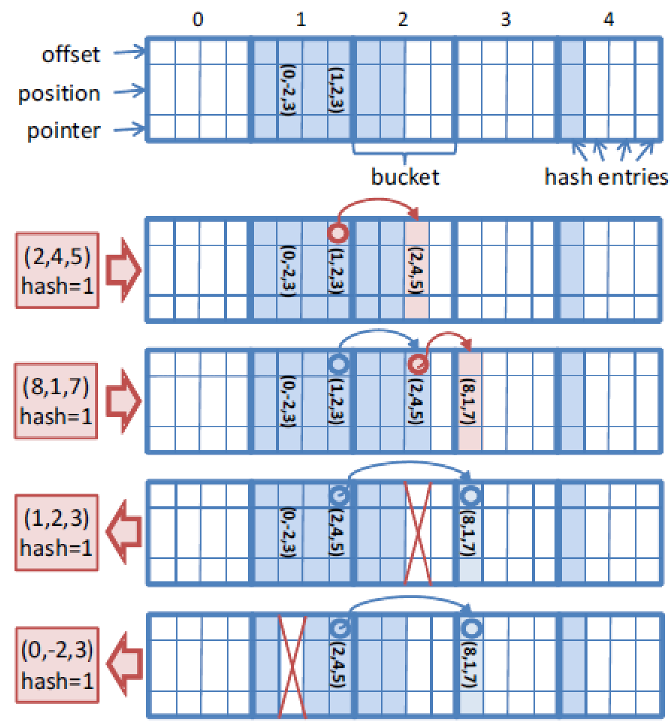


- Hash entries
  - Contain the world coordinates and pointer to voxel block
  - Retrieve block using world coordinates (x, y, z) and hash function

$$H(x,y,z) = (x \cdot p_1 \oplus y \cdot p_2 \oplus z \cdot p_3) \bmod n, \qquad p_1, p_2, p_3 \ \text{PRIME}$$

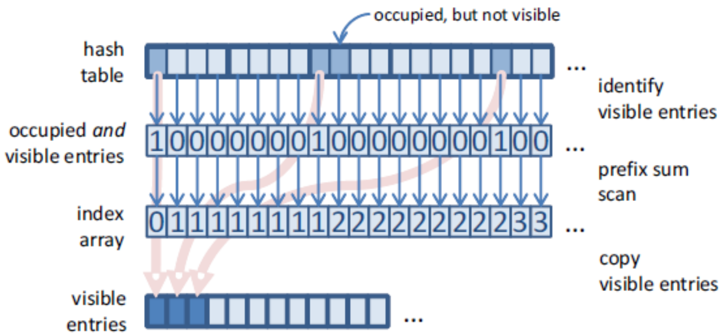
Method: Data structure

- Insertion
  - Evaluate hash value and search for free space in bucket and linked list
  - Collision handling
  - Return reference if already existent
- Deletion
  - Very much like insertion
  - If last element of bucket, copy offset entry to its place
- Retrieval
  - Search <u>complete</u> bucket and linked list



Method: Voxel block integration

- Selection
  - Store binary flag in array if voxel block is visible and occupied.
  - Scan array using parallel prefix sum
  - Create final array of visible entries

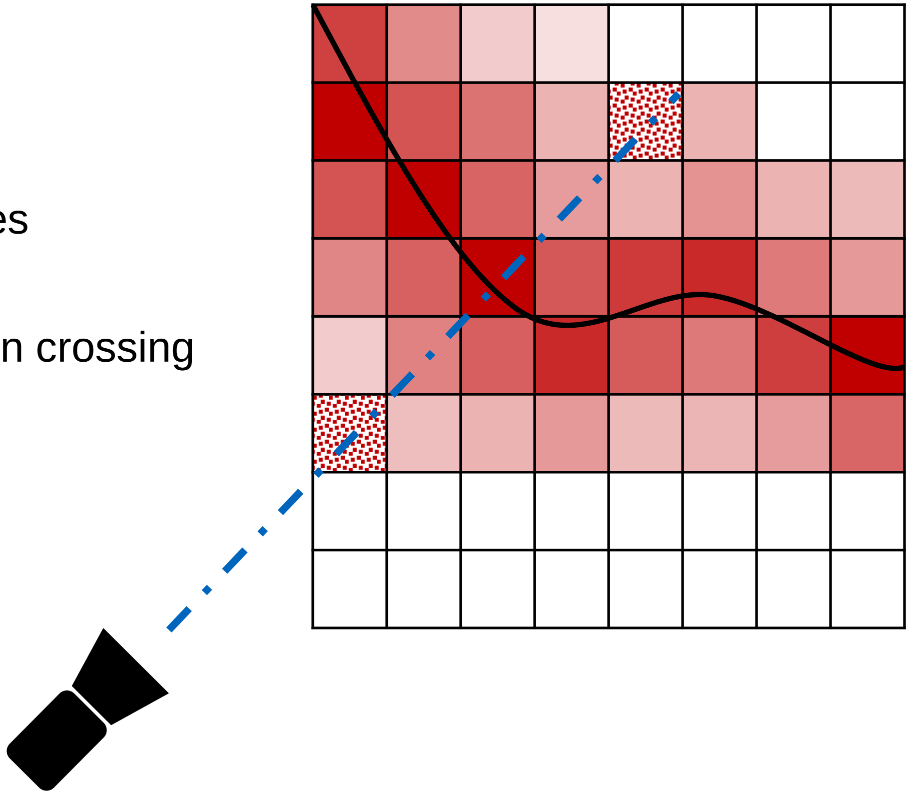


- Update
  - Compute new TSDFs like Curless and Levoy
  - Weights according to depth values
  - Update color as running average

Method: Surface extraction

Use raycasting to extract the implicitly stored isosurfaces

- Raycasting
  - March from min to max voxel
  - Use tri-linear interpolation for TSDF values
  - Determine zero crossings for surface
  - Skip defined intervals + use line search on crossing



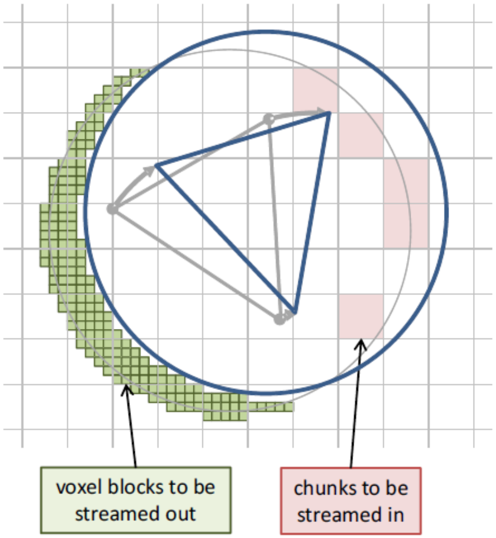
Method: Streaming

- Limited by GPU memory
  - Use bi-directional streaming

- GPU to host
  - Host uses uniform grid of $1m^3$ chunks (no voxel blocks)

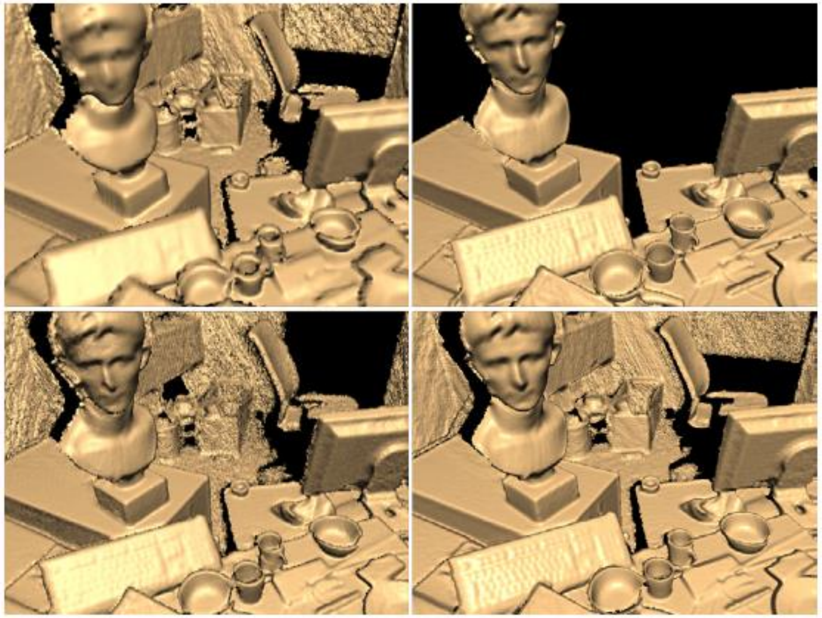
- Host to GPU
  - Identify chunks that fall completely in active region
  - Select one chunk per frame

- Synchronization
  - Possible memory leaks
  - Store chunk based, binary occupancy grid on GPU



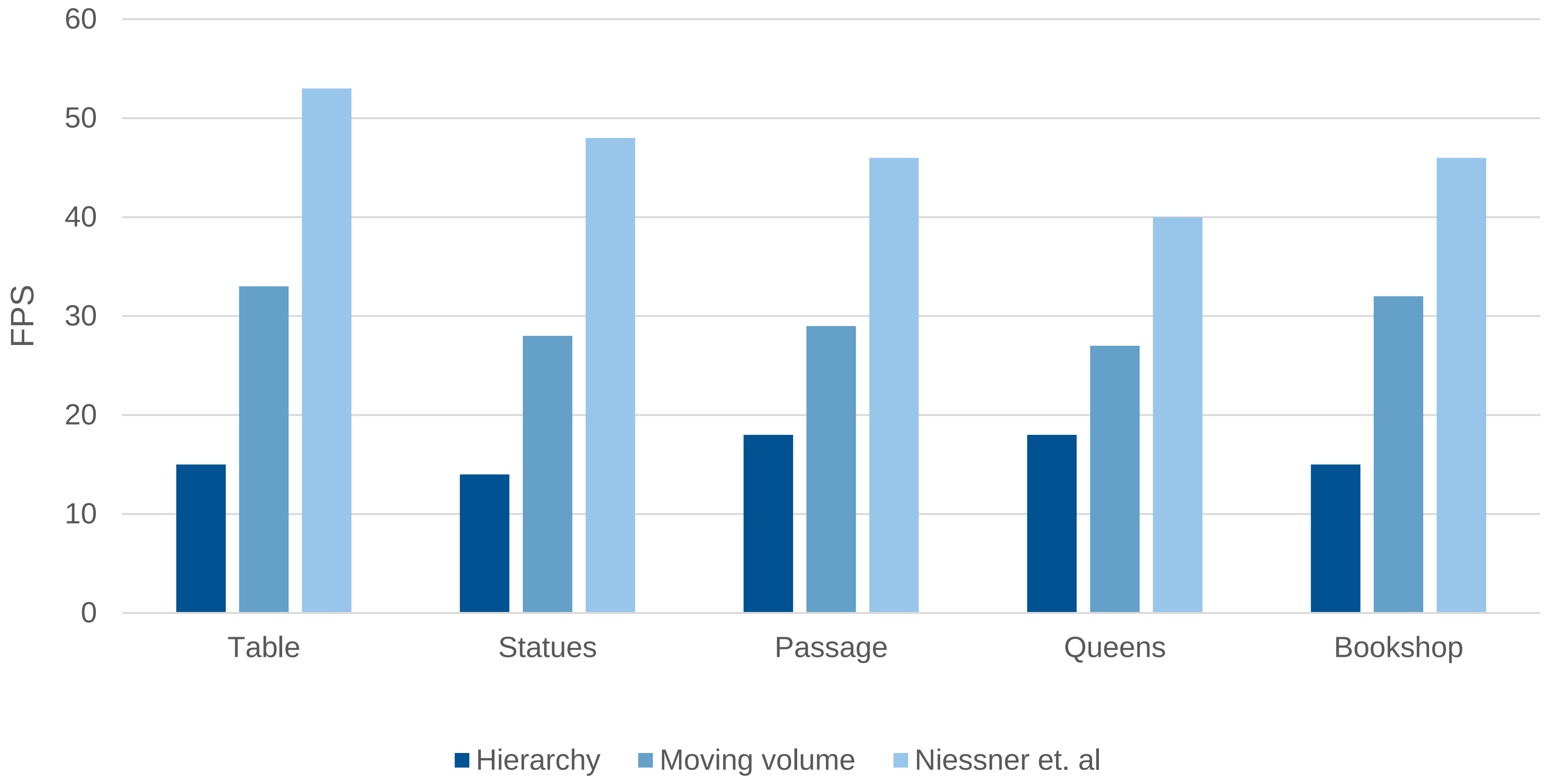
Qualitative results

Top: Niessner et al.
Bottom: Offline method by Zhou and Koltun



Top left: Moving volume
Top right: Moving volume (less extent)
Bottom left: Hierarchical
Bottom right: Niessner et. al

Quantitative results



Personal comments

- Easy to implement
  - Simple data structures and well-known algorithms

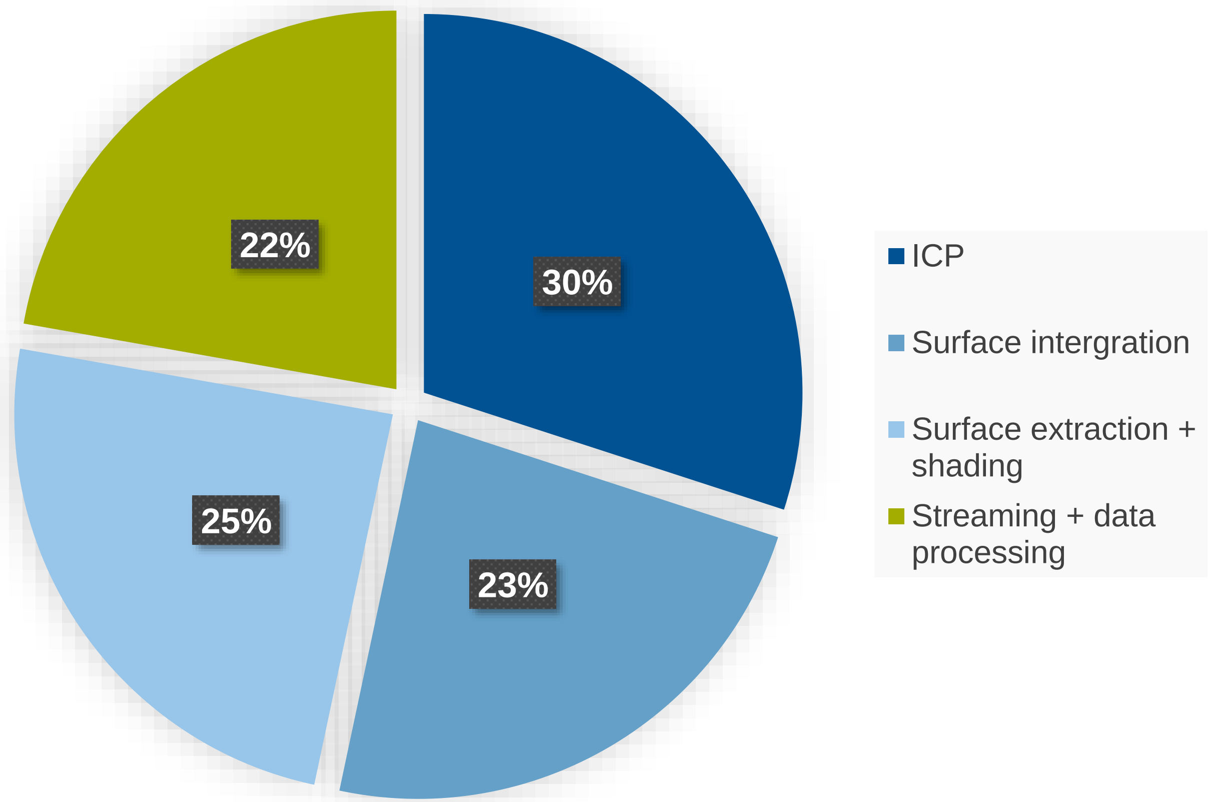
- Lightweight
  - Smart choice for data structure with low memory footprint and high speed

- Best of both worlds
  - Combines the benefits of accurate offline methods with real-time performance of online methods

Open discussion

BACKUP

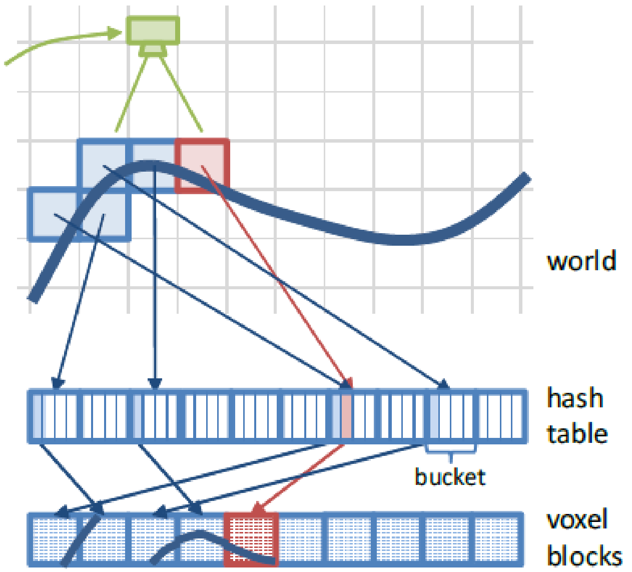
Phase timings



Method: Overview

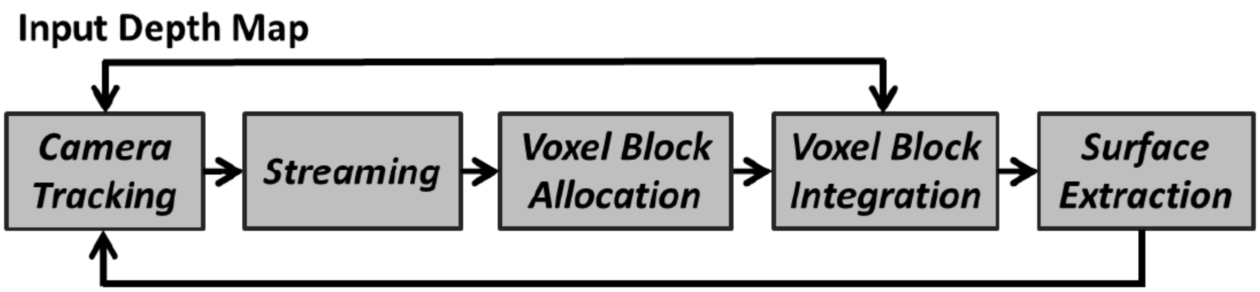
- Subdivide space into voxel blocks saved in a hash table
  - Each voxel consist of a TSDF, weight and color
  - Hash table is spatially unstructured

- Integration
  - Allocate voxel blocks
  - Update SDFs, color and weight for all voxels
  - Garbage collect voxels to far away from the isosurface



Method: Overview

- Surface extraction
  - Raycast implicit surface
  - Estimate new 6DoF pose using a frame-to-model version of point-plane ICP
  - Voxel blocks are streamed to host if their world positions exit the camera view



Method: Data structure

- Collisions
  - Appear when coordinates get mapped to same hash value
  - Organize table into buckets for unique hash values
  - Store new entry in next free spot in bucket
  - If already full, add to other free bucket and use offset for last element in bucket
  - Last element of bucket reserved