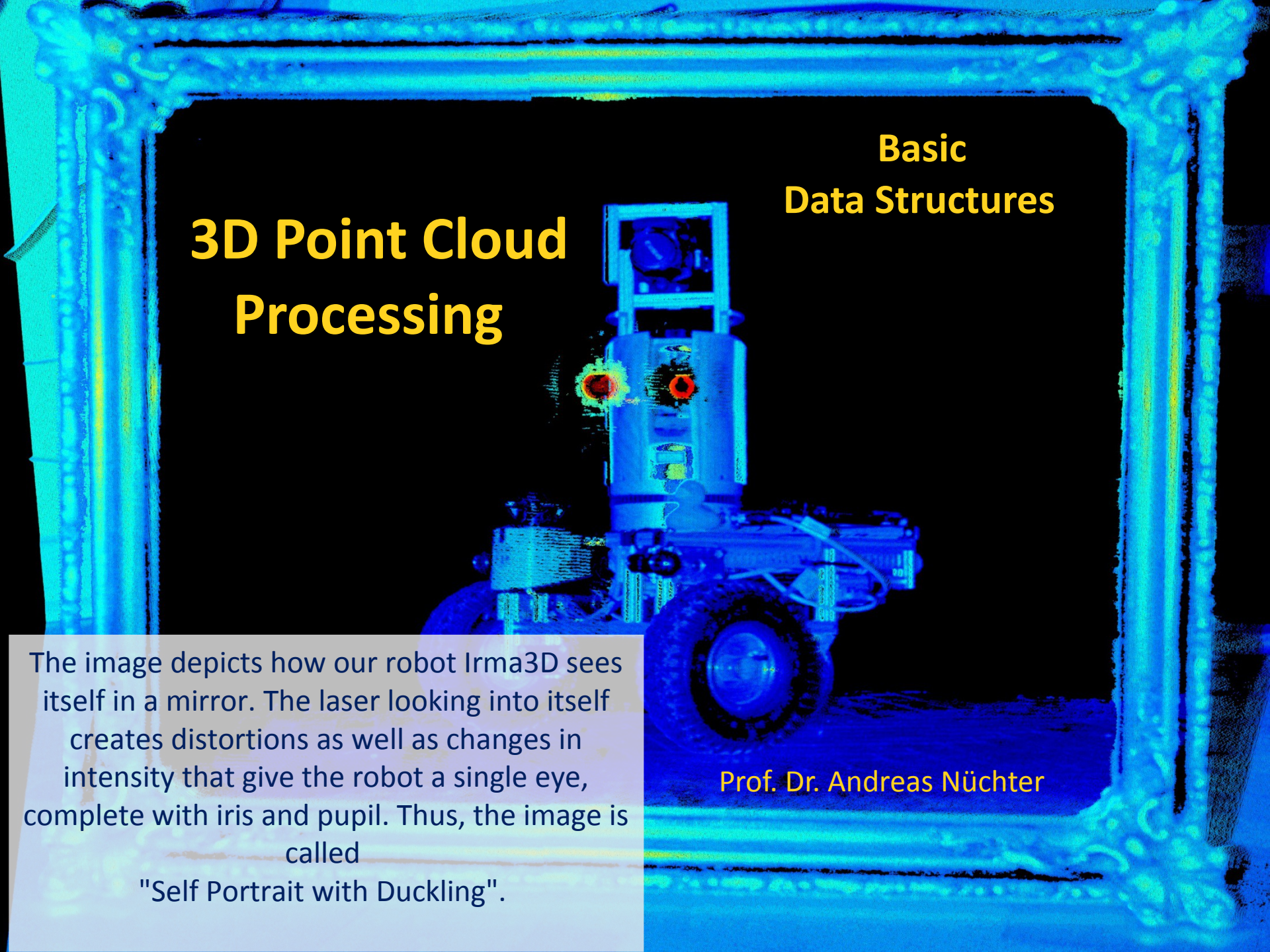
3D Point Cloud Processing

Basic Data Structures



The image depicts how our robot Irma3D sees itself in a mirror. The laser looking into itself creates distortions as well as changes in intensity that give the robot a single eye, complete with iris and pupil. Thus, the image is called
"Self Portrait with Duckling".

Prof. Dr. Andreas Nüchter

3D Point Cloud as ...

… vector of *(x,y,z)*-values

- In 3DTK we have …
  - While reading a 3D Point Cloud

```
virtual void readScan(const char* dir_path,
                      const char* identifier,
                      PointFilter& filter,
                      std::vector<double>* xyz,
                      std::vector<unsigned char>* rgb,
                      std::vector<float>* reflectance,
                      std::vector<float>* amplitude,
                      std::vector<int>* type,
                      std::vector<float>* deviation);
```

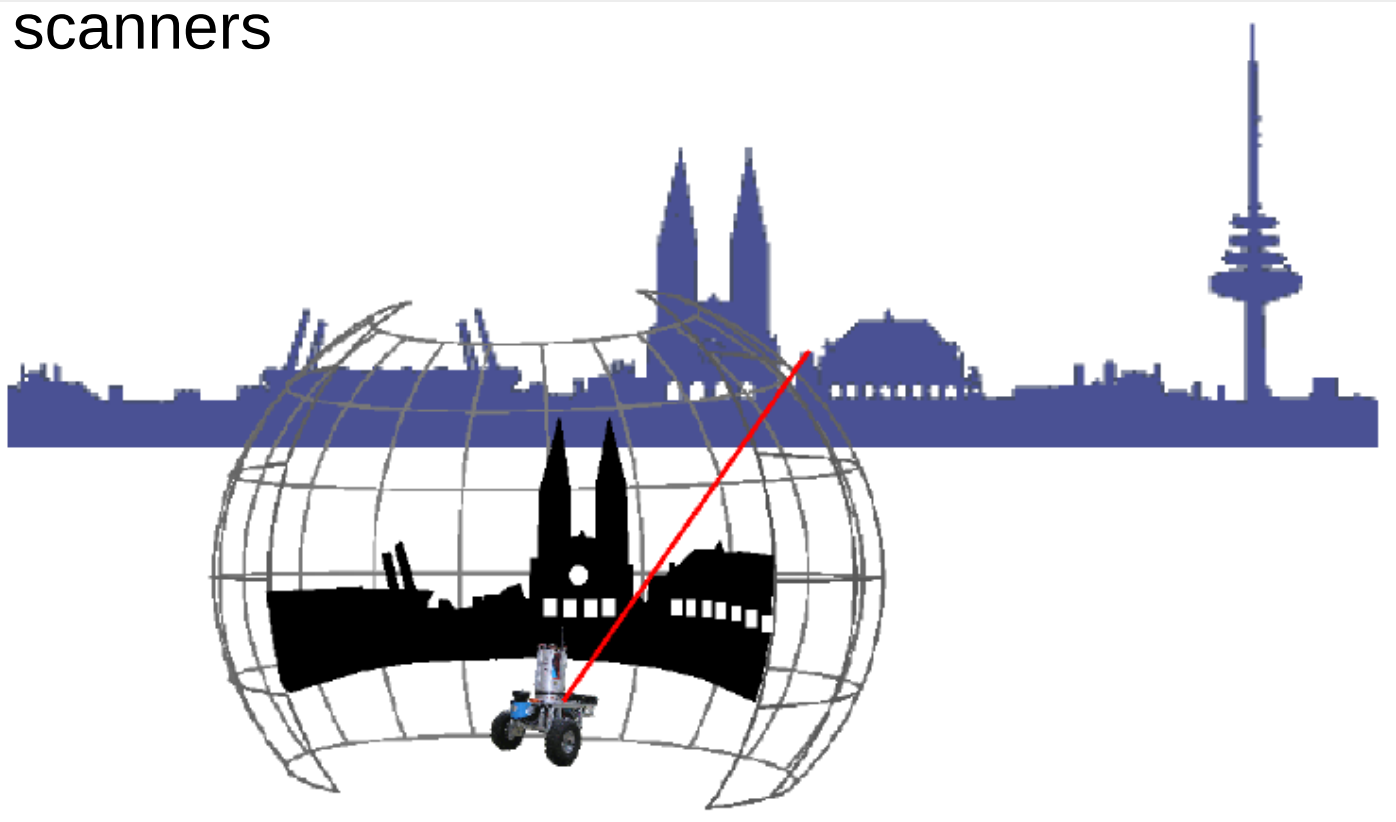
  - Called e.g., in the function BasicScan::get()
  - Finally the data ist stored in a STL-map

```
std::map<std::string, std::pair<unsigned char*, unsigned int>>
                    m_data;
```


3D Point Cloud as ...

… as range / intensity image

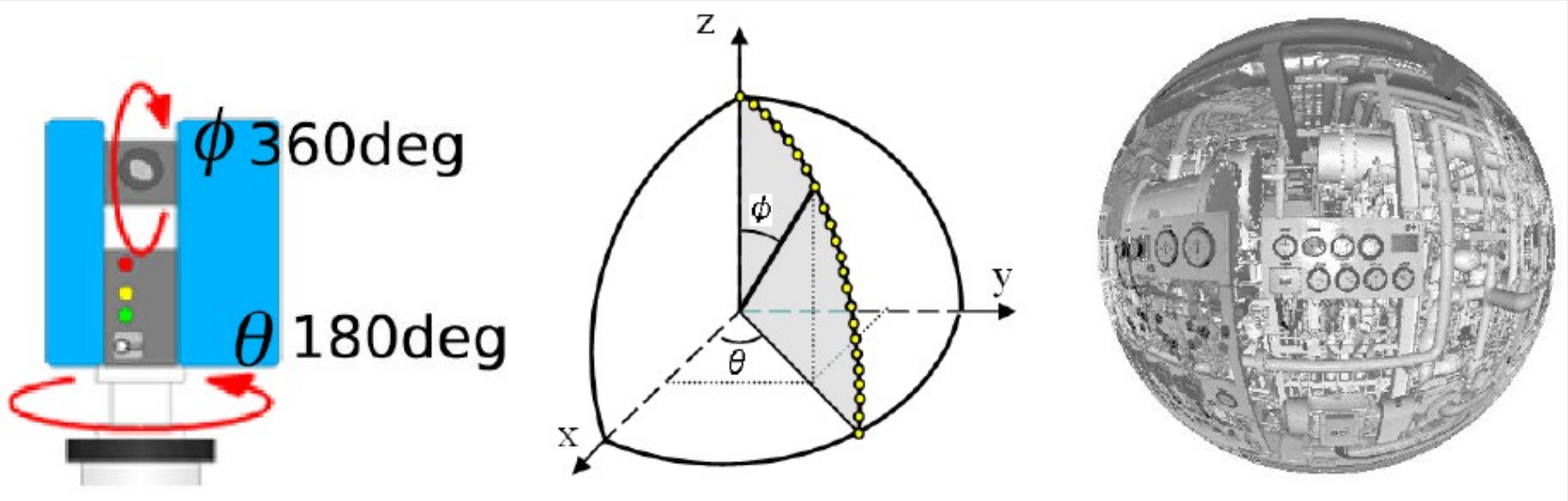
- 2D array for kinect-like sensors
- Laser scanners



3D Point Cloud as ...

… as range / intensity image

- 2D array for kinect-like sensors
- Laser scanners



3D Point Cloud as ...

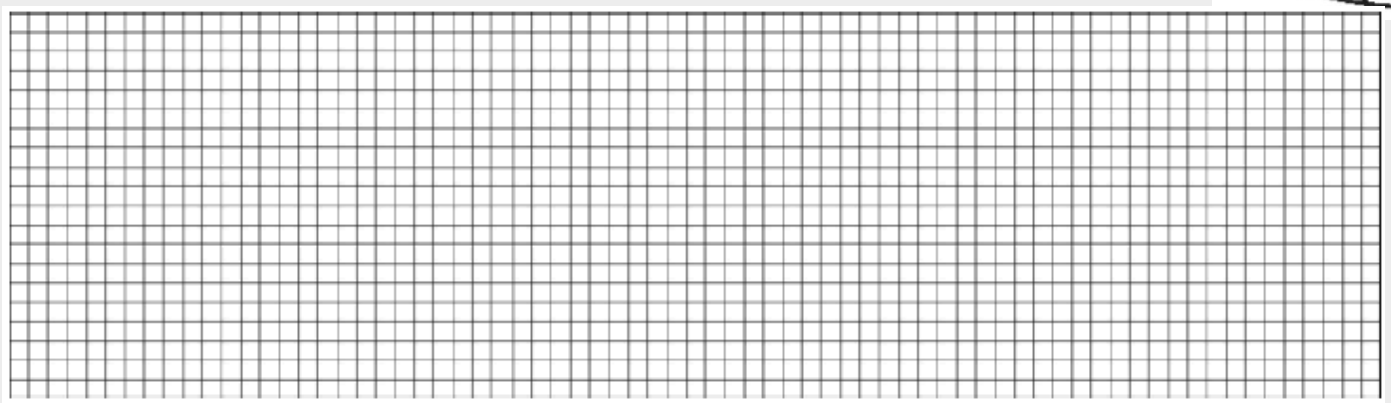
… as range / intensity image

- 2D array for kinect-like sensors
- Laser scanners

3D Point Clouds as 2D arrays (1)

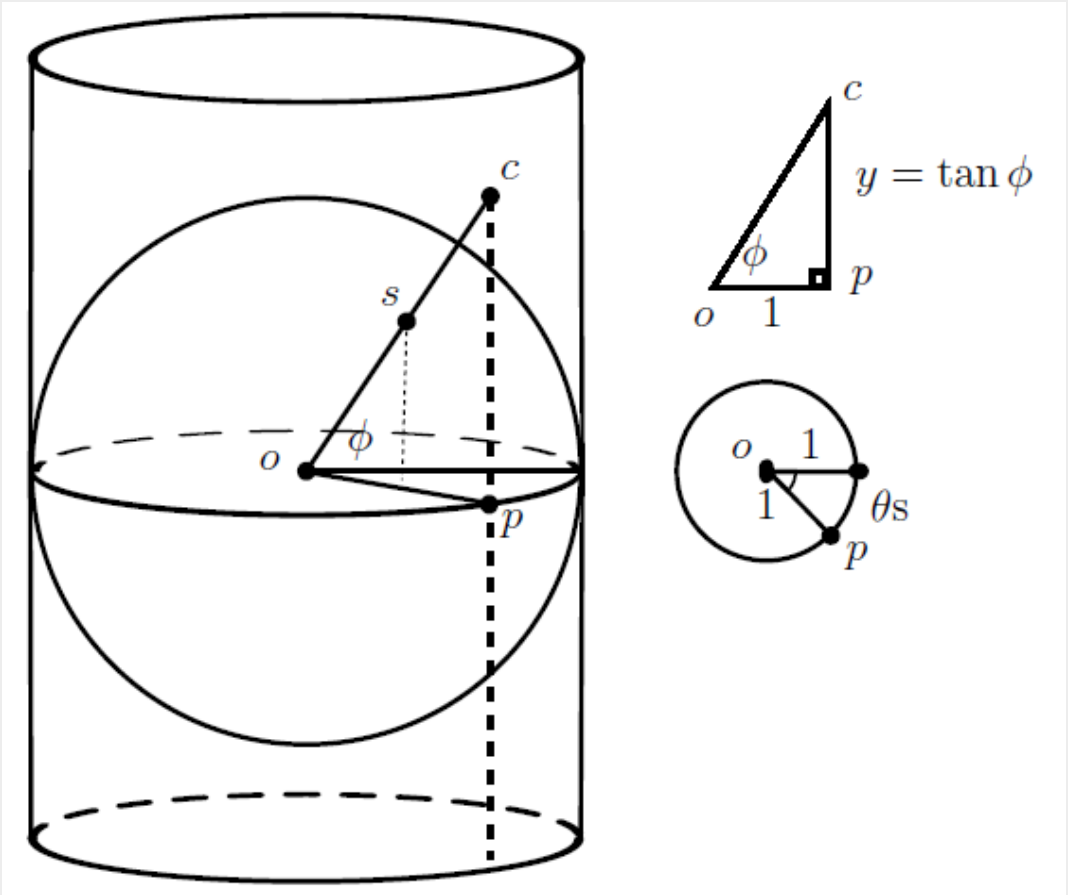
- Laser scanners
  - Equirectangular projection

$$x = \theta$$
$$y = \varphi$$



3D Point Clouds as 2D arrays (2)

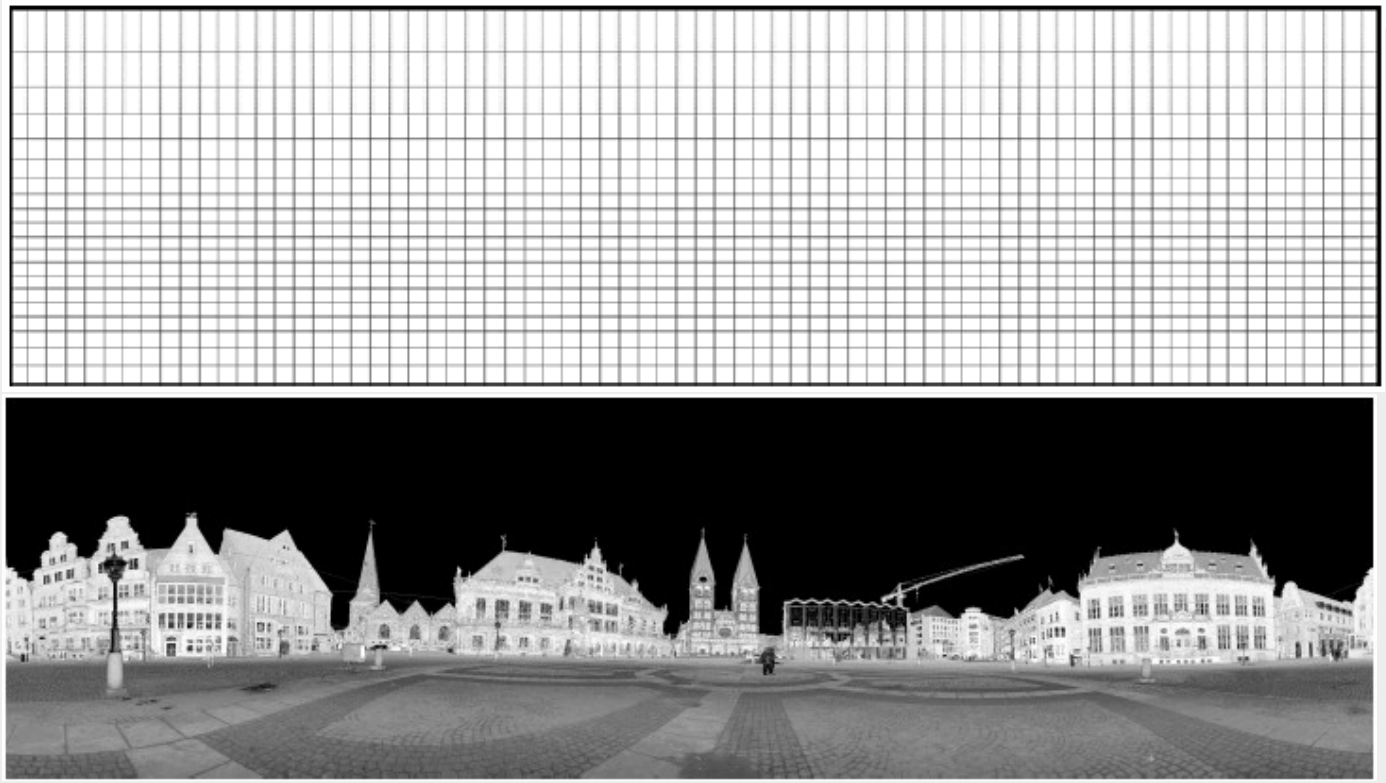
- Laser scanners
  - Cylindrical projection



3D Point Clouds as 2D arrays (3)

- Laser scanners
  - Cylindrical projection

$$x = \theta$$
$$y = \tan\varphi$$

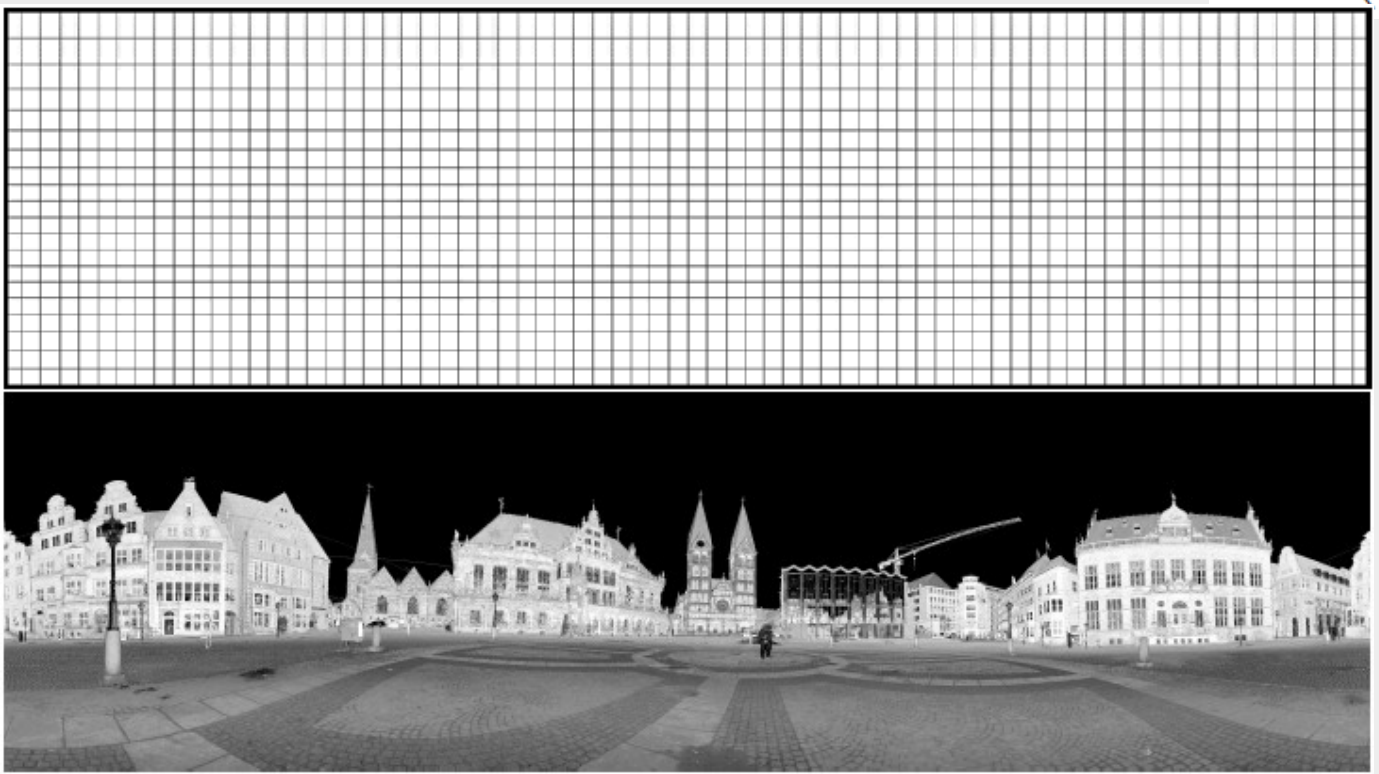


3D Point Clouds as 2D arrays (4)

- Laser scanners
  - Mercator projection
    - Cannot be “constructed”, only computational principle
    - The Mercator projection is an isogonic projection, i.e., angles are preserved

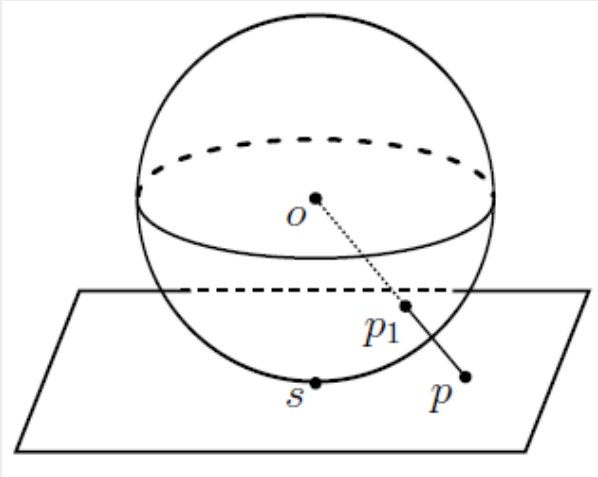
$$x = \theta$$

$$y = \ln\left(\tan\varphi + \frac{1}{\cos\varphi}\right)$$



3D Point Clouds as 2D arrays (5)

- Laser scanners
  - Rectilinear
  - also “gnomonic" or “tangentplane" projection.



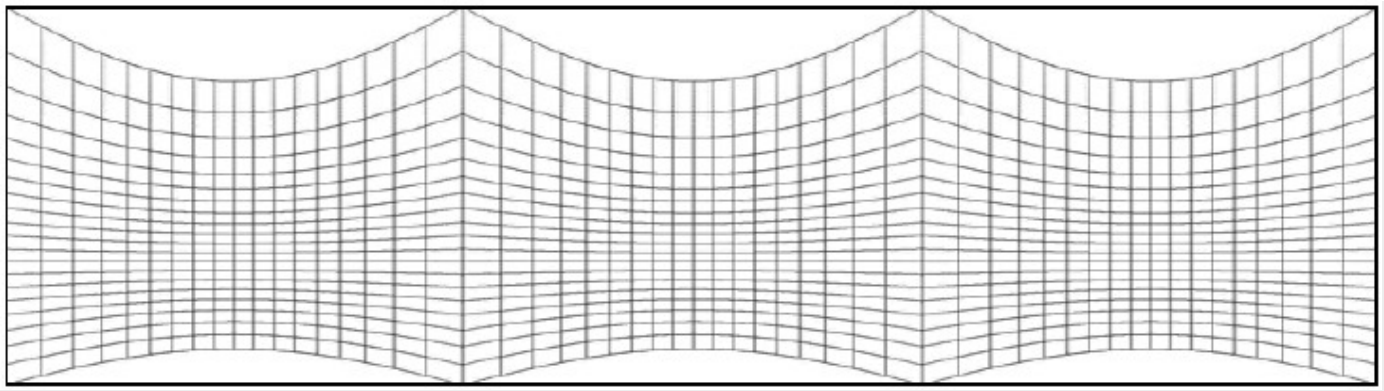
The primary advantage of the rectilinear projection is that it maps straight lines in 3D space to straight lines in the 2D image.

$$x = \frac{\cos\varphi \sin(\theta - \theta_0)}{\sin\varphi_1 \sin\varphi + \cos\varphi_1 \cos\varphi \cos(\theta - \theta_0)}$$

$$y = \frac{\cos\varphi_1 \sin\varphi - \sin\varphi_1 \cos\varphi \cos(\theta - \theta_0)}{\sin\varphi_1 \sin\varphi + \cos\varphi_1 \cos\varphi \cos(\theta - \theta_0)}.$$

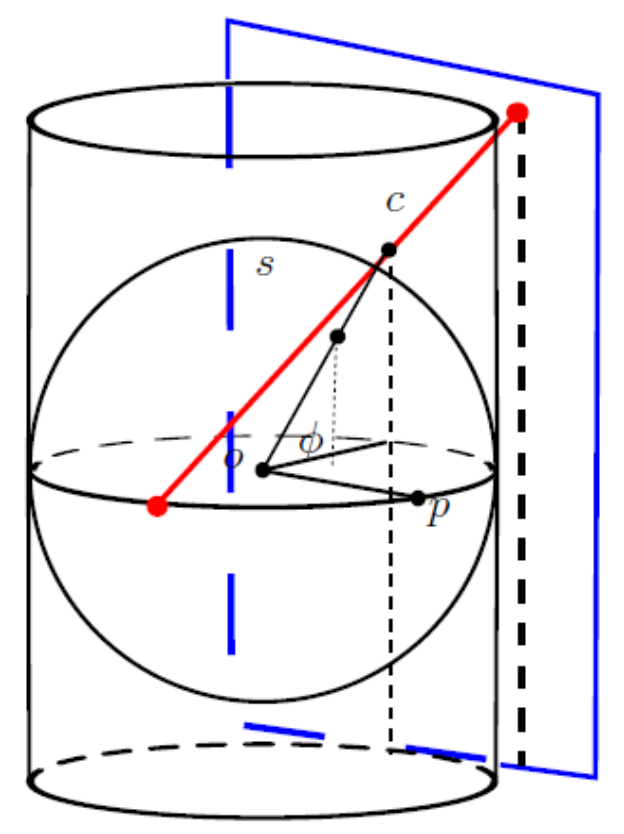
3D Point Clouds as 2D arrays (6)

- Laser scanners
  - Rectilinear
  - also “gnomonic" or “tangentplane" projection.



3D Point Clouds as 2D arrays (7)

- Laser scanners
  - Pannini, also called Panini or "Recti-Perspective" or "Vedutismo"
    - This projection can be imagined as the rectilin projection of a 3D cylindrical image.



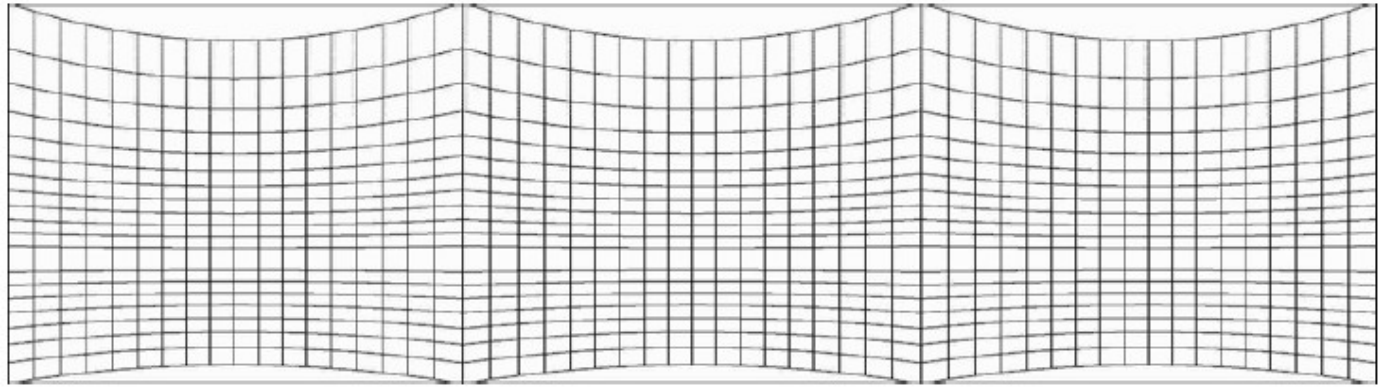
- This image is itself a projection of the sphere onto a tangent cylinder.
- The center of the rectilinear projection can be different and is on the view axis at a distance of d from the cylinder axis
- The recommended field of view for the Pannini projection is less than 150° in both vertical and horizontal directions.

3D Point Clouds as 2D arrays (8)

- Laser scanners
  - Pannini projection

$$x = \frac{(d+1)\sin(\theta-\theta_0)}{d+\sin\varphi_1\tan\varphi+\cos\varphi_1\cos(\theta-\theta_0)}$$

$$y = \frac{(d+1)\tan\varphi\left(\cos\varphi_1-\sin\varphi_1\left(\frac{1}{\tan\varphi}\right)\cos(\theta-\theta_0)\right)}{d+\sin\varphi_1\tan\varphi+\cos\varphi_1\cos(\theta-\theta_0)}$$



3D Point Clouds as 2D arrays (9)

- Laser scanners
  - Stereographic projection
    - It can be imagined by placing a paper tangent to a sphere and by illuminating it from the opposite pole.

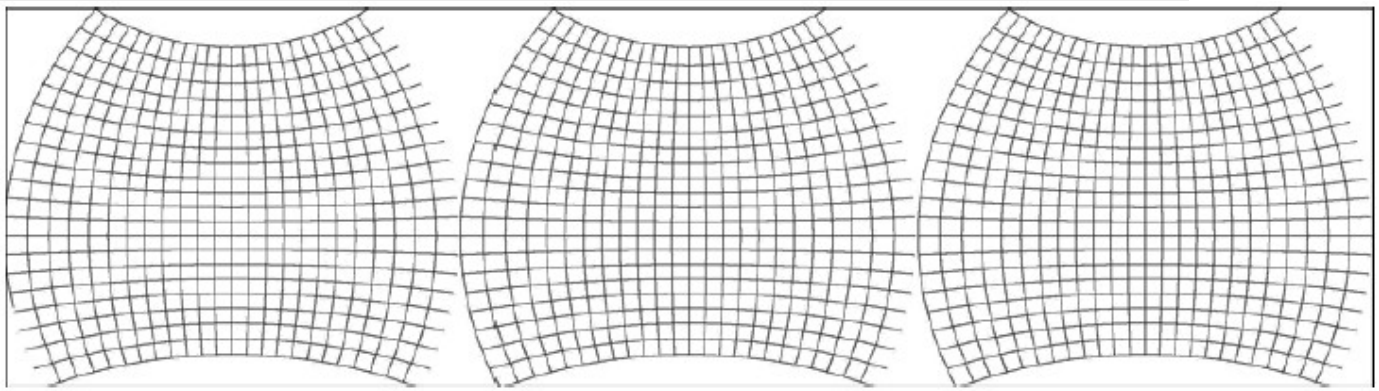
$$x = \frac{2R\cos\varphi\sin(\theta-\theta_0)}{1+\sin\varphi_1\sin\varphi+\cos\varphi_1\cos\varphi\cos(\theta-\theta_0)}$$

$$y = \frac{2R\left(\cos\varphi_1\sin\varphi-\sin\varphi_1\cos\varphi\cos(\theta-\theta_0)\right)}{1+\sin\varphi_1\sin\varphi+\cos\varphi_1\cos\varphi\cos(\theta-\theta_0)}$$

    - R = 1 generates exactly the same equations as the Pannini projection and high values for R introduce more distortion.

3D Point Clouds as 2D arrays (10)

- Laser scanners
  - Stereographic projection



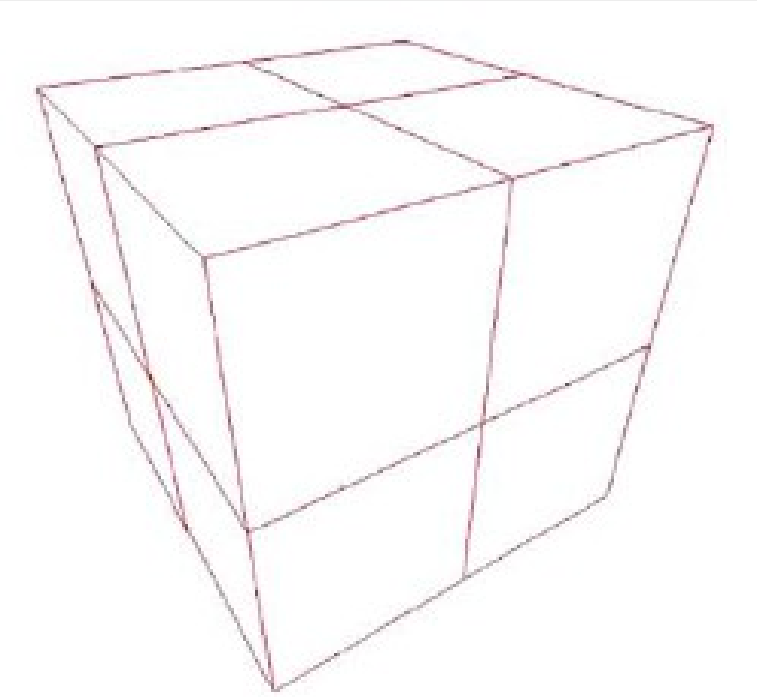
More Information per Pixel

- For representing a 3D point cloud as array it is advantageous to store more information per (x,y)-pixel in a panorama image (cf. panorama.h and panorama.cc)

```
iReflectance.at<uchar>(y,x) = (*it)[3]*255;   //reflectance
iRange.at<float>(y,x) = range;                //range
if(mapMethod == FARTHEST){
  //adding the point with max distance
  if( iRange.at<float>(y,x) < range ){
     iMap.at<cv::Vec3f>(y,x)[0] = (*it)[0];  //x
     iMap.at<cv::Vec3f>(y,x)[1] = (*it)[1];  //y
     iMap.at<cv::Vec3f>(y,x)[2] = (*it)[2];  //z
  }
} else if(mapMethod == EXTENDED){             //adding all the points
  cv::Vec3f point;
  point[0] = (*it)[0];                        //x
  point[1] = (*it)[1];                        //y
  point[2] = (*it)[2];                        //z
  extendedIMap[y][x].push_back(point);
```

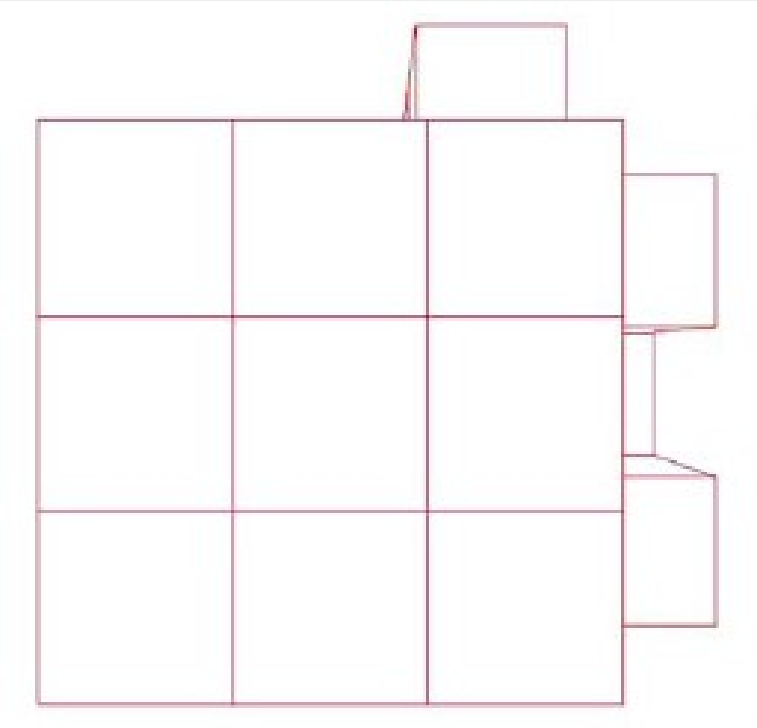
Other Representation of 3D Point Clouds

- Please consider the following



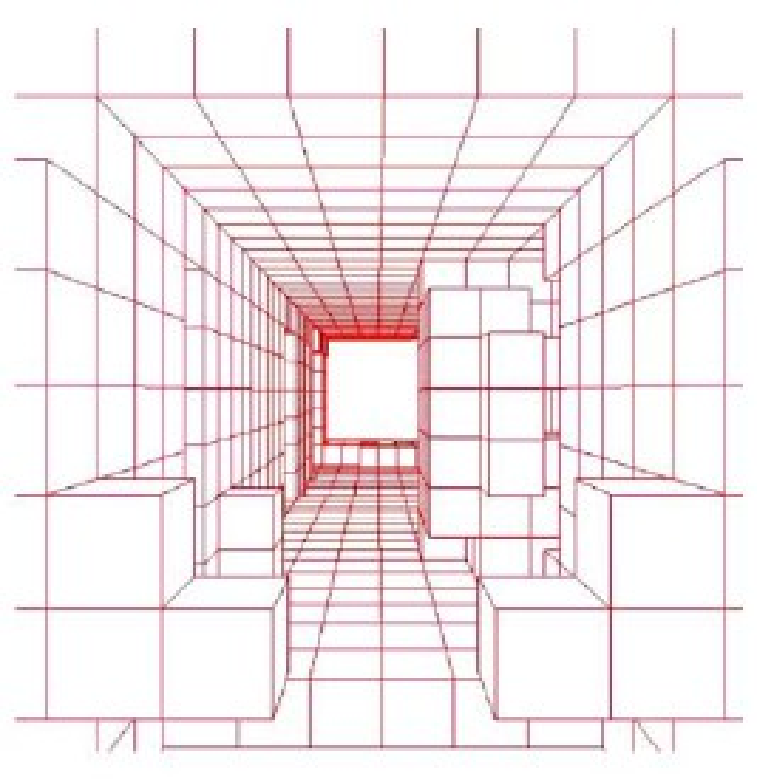
Other Representation of 3D Point Clouds

- Please consider the following



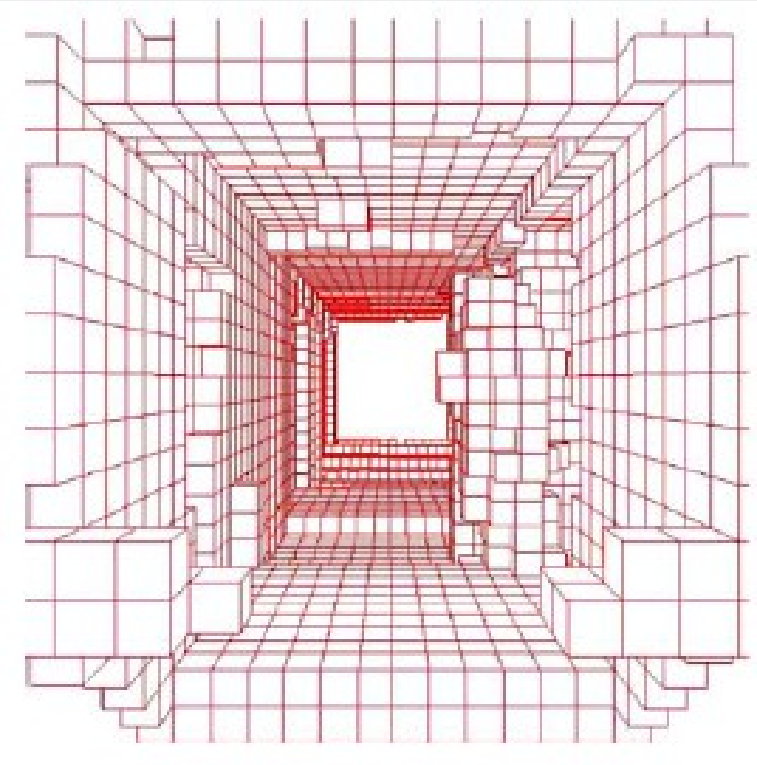
Other Representation of 3D Point Clouds

- Please consider the following



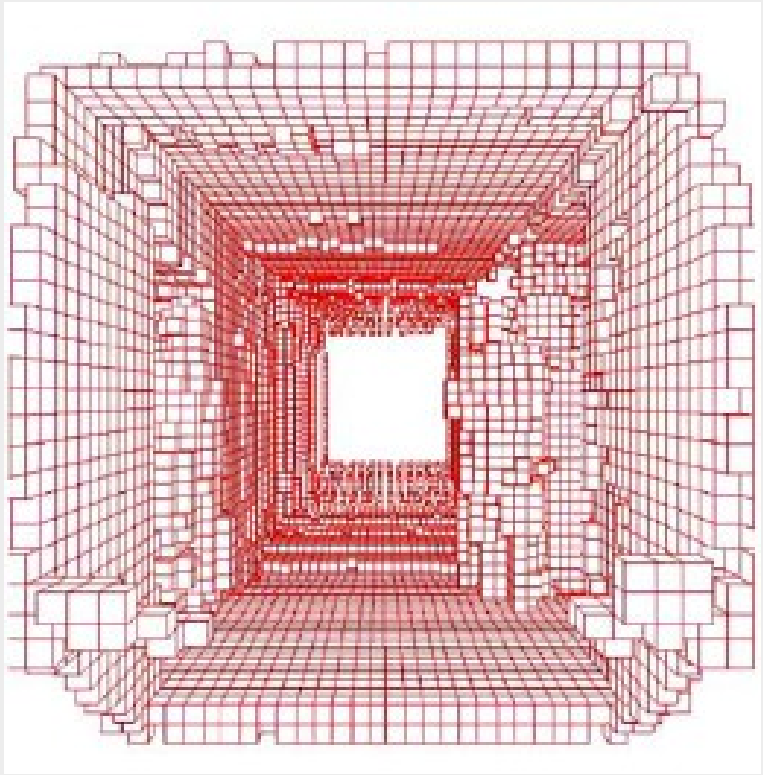
Other Representation of 3D Point Clouds

- Please consider the following



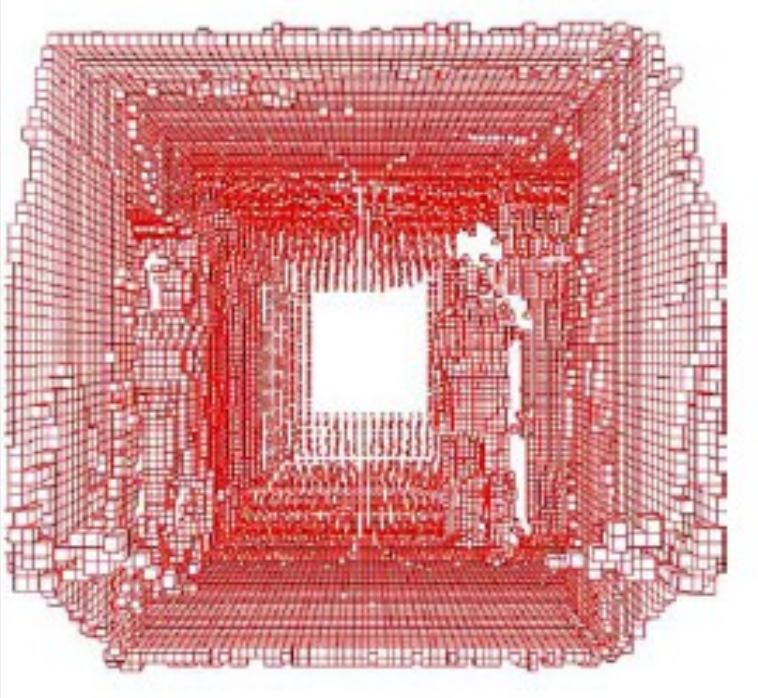
Other Representation of 3D Point Clouds

- Please consider the following



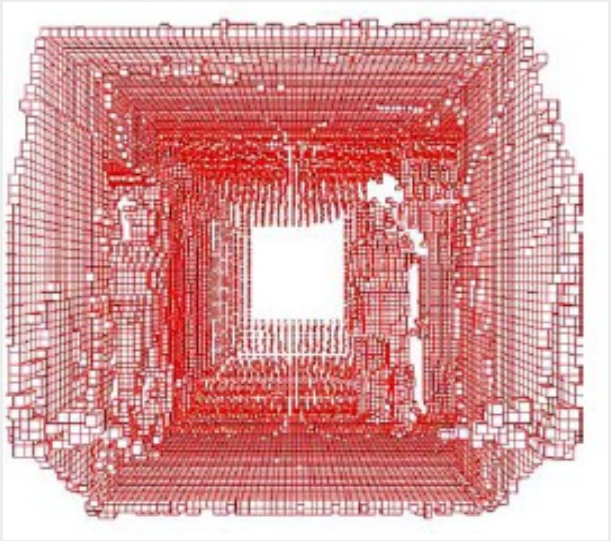
Other Representation of 3D Point Clouds

- Please consider the following



Other Representation of 3D Point Clouds

- Please consider the following



- Oc-trees represent a way to store 3D point cloud data

Further Readings

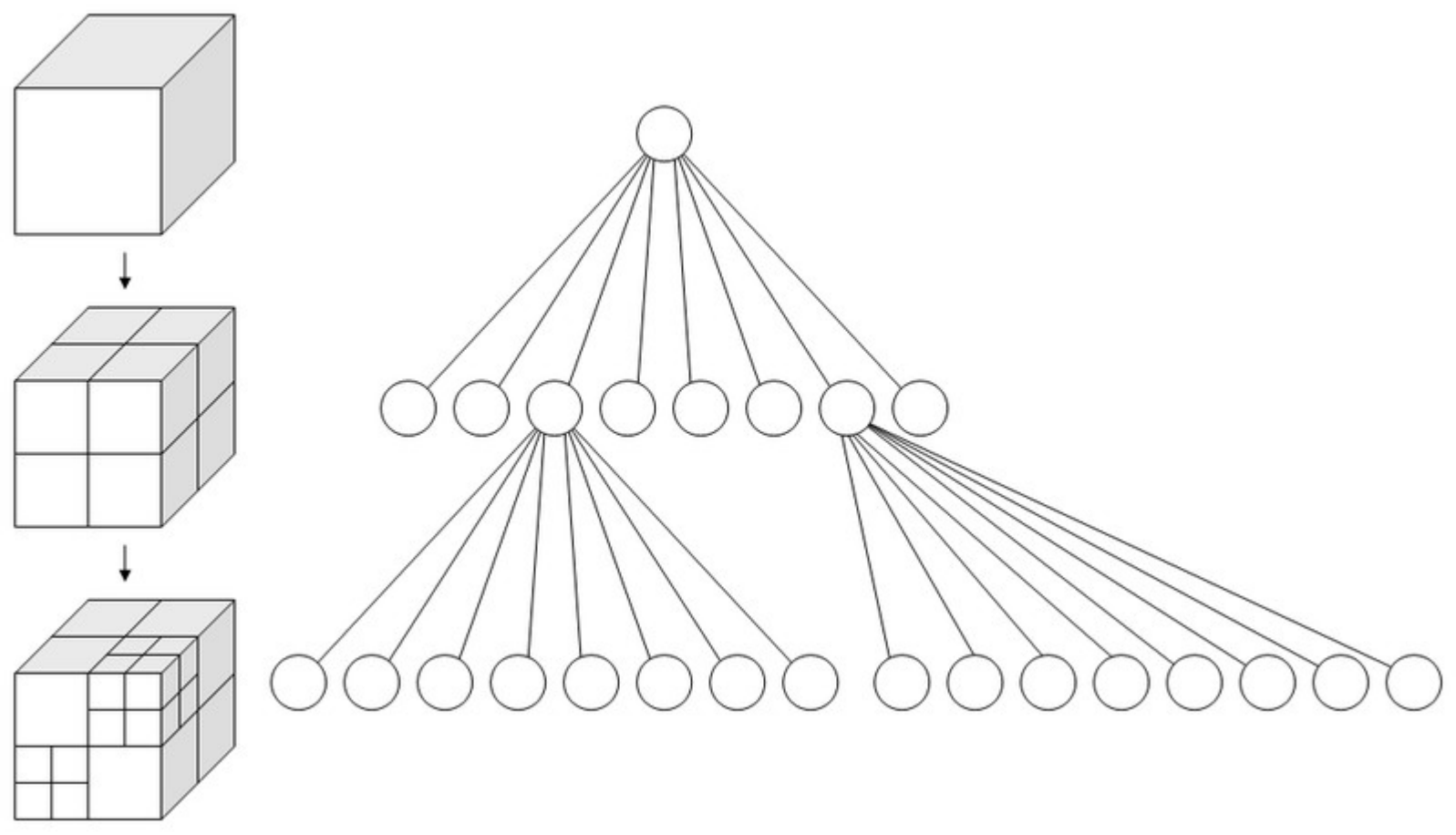
- Please consider the paper - “A Study of Projections for Key Point Based Registration of Panoramic Terrestrial 3D Laser Scans“
- Please read the paper - “Octrees for storing 3D point clouds” of the paper “One Billion Points in the Cloud – An Octree for Efficient Processing of 3D Laser Scans”
- Things to try
  - Viewing a high resolution outdoor 3D scan with colors

bin/show –s 0 –e 0 –f riegl_txt –-reflectance bremen_city --saveOct

bin/show –s 0 –e 0 –f riegl_txt –-reflectance bremen_city --loadOct

Oc-trees (1)

- Every node has 8 children



Oc-trees (2)

- Empty nodes / voxels can be pruned
- Every node has 8 children

```
struct OcTree {
  float center[3];
  float size[3];
  OcTree *child[8];
  int nr_points;
  float **points;
};
```

- Definition of an oc-tree with redundant information and eight pointers to child nodes. The size of this node is 100 Bytes.

Oc-trees (3)

- Statistics of the Bremen City data set

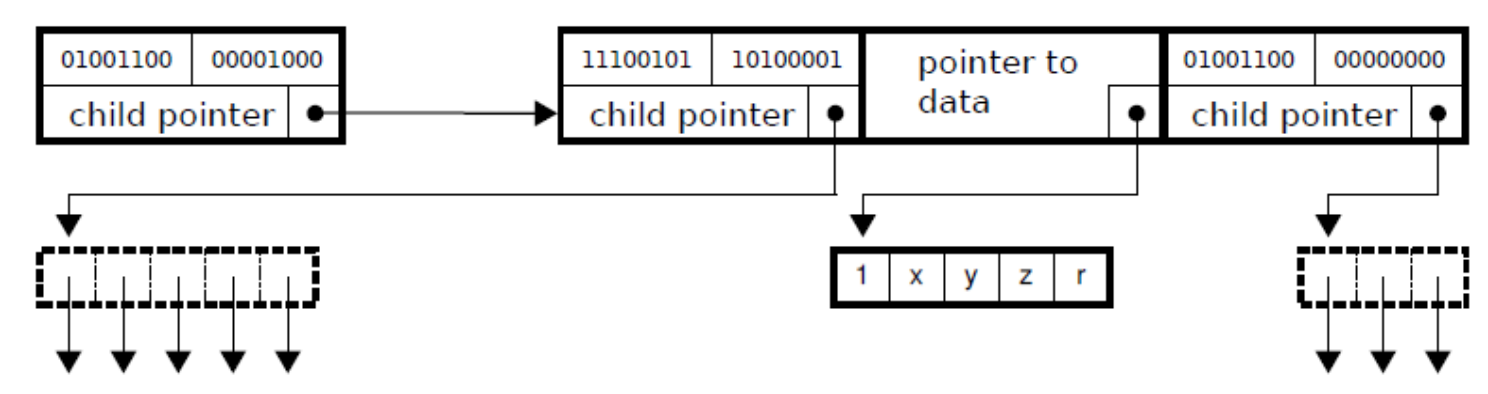
	100 byte	
Leaf size (cm)	Mem. Size	Constr. time (ms)
8560	1.9 kB	557.1
4280	4.8 kB	694.1
2140	13.9 kB	939.2
1070	45.3 kB	1165.7
535	130.1 kB	1279.3
267	405.2 kB	1529.4
133	1.25 MB	1656.5
66.8	3.85 MB	1895.5
33.4	11.91 MB	2002.3
16.7	36.65 MB	2146.5
8.35	109.53 MB	2290.3
4.17	301.28 MB	2471.9
2.08	742.98 MB	2576.0
1.04	1.568 GB	2899.6
0.52	2.643 GB	3199.9

- Exponential growth

Efficient Oc-Trees (1)

- Two proposed encodings of an octree node optimized for memory efficiency.
- The child pointer as the relative pointer is the largest part of an octree node, but varies in size to accommodate different systems. In our implementation for 64 bit systems, it is 48 bit. valid and leaf are 8 bit large.
- Left: The proposed encoding with separate bit fields for valid and leaf. An entire node is thus contained in only 8 bytes of memory.
- Right: Alternative solution resulting in a constant depth octree.

Efficient Oc-Trees (2)



- An example of a simple oc-tree as it is stored in 3DTK.
- The node in the upper left has three valid children, one of which is a leaf. Therefore, the child pointer only points to 3 nodes stored consecutively in memory. The leaf node in this example is a simple pointer to an array which stores both the number of points and the points with all their attributes.

Efficient Oc-Trees (3)

	8 byte				100 byte	
Leaf size (cm)	# Nodes	# Leaves	Mem. Size	Constr. time (ms)	Mem. Size	Constr. time (ms)
8560	6	12	192 B	1019.2	1.9 kB	557.1
4280	18	28	480 B	1321.2	4.8 kB	694.1
2140	46	86	1.4 kB	1535.7	13.9 kB	939.2
1070	129	296	4.5 kB	1706.1	45.3 kB	1165.7
535	408	800	12.8 kB	1909.7	130.1 kB	1279.3
267	1166	2595	40.4 kB	2081.8	405.2 kB	1529.4
133	3616	7993	124.8 kB	2299.7	1.25 MB	1656.5
66.8	11130	24587	384.1 kB	2473.3	3.85 MB	1895.5
33.4	33965	75999	1.18 MB	2687.5	11.91 MB	2002.3
16.7	102728	233413	3.62 MB	2873.3	36.65 MB	2146.5
8.35	302573	687529	10.67 MB	3134.9	109.53 MB	2290.3
4.17	814040	1808993	28.22 MB	3432.0	301.28 MB	2471.9
2.08	1927234	4166979	65.42 MB	3721.0	742.98 MB	2576.0
1.04	4031140	7783889	125.65 MB	3901.8	1.568 GB	2899.6
0.52	5592151	10142923	166.45 MB	4077.7	2.643 GB	3199.9

Efficient Oc-Trees (4)

- Comparison with other oc-trees

Library	Node size	Remarks
`xgrt`	$144 + x$	
octomap	$72 + x$	
PCL	$64 + x$	
PCL low memory base	$25 + x$	(since ver 1.1.1. Sept. 2011)
CloudCompare	16	size of leaf node
3DTK	8	

Lookup in an Oc-tree (1)

- Naive lookup implementations perform collision checks with the oc-tree planes.
- We use integer coordinates for an efficient traversal of the oc-tree with only a few bit operations.
- An oc-tree with depth d has integer coordinates 0 to 2d − 1 in each dimension.
- We assume the existence of a pre-computed array childBitDepth with childBitDepth[d] = 1 ≪ (maxDepth − d − 1).

Algorithm 2 lookup(Vector3i index, octree node, octree *parentTrace, int depth)

```
loop
    int childBit = childBitDepth[depth]
    int childIndex = (index.x & childBit ≠ 0) ≪ 2 | (index.y & childBit ≠ 0) ≪ 1 | (index.z & childBit ≠ 0)
    octree *parentTrace[depth] = &node
    depth++
    node = node.children[childIndex]
    if isLeaf(node) then
```

Here the integer coordinates are mapped to the index of the child that contains the given coordinates. The algorithm also shows how parent pointers are simulated by a simple trace that is extended during the traversal of the tree.

Lookup in an Oc-tree (2)

- The function to find a neighbor node is an extended lookup.
- To find a neighbor of a given node, the node in the parent trace is selected that is the deepest that still contains the desired index. This can be efficiently computed by comparing the current node index with the desired index
- Then a lookup starting at that parent is started to locate the corresponding neighbor.

Algorithm 3 findNeighbor(Vector3i dindex, Vector3i cindex, octree *parentTrace)

```
int depth = mostSignificantBit( (cindex.x^dindex.x) | (cindex.y^dindex.y)
| (cindex.z^dindex.z) );
lookup(dindex, parentTrace[depth], parentTrace, depth);
```

Using an Oc-tree for 3D Point Cloud Reduction

- Generate an oc-tree until you reached the desired voxel size
- Select the center point of each voxel for the reduced point cloud.

  Or

- Select n point randomly from each voxel.

- This is implemented in 3DTK (program `scan_red`)

```
bin/scan_red -s 0 -e 0 -f uos
        --reduction OCTREE --voxel 10 --octree 0 dat

bin/scan_red -s 0 -e 0 -f uos
        --reduction OCTREE --voxel 10 --octree 1 dat


bin/scan_red -s 0 -e 0 -f RIEGL_TXT
        --reduction OCTREE --voxel 10 --octree 0
        --reflectance ~/dat/bremen_city/
bin/show -s 0 -e 0 -f uosr ~/dat/bremen_city/reduced
```

3D Point Cloud as ...

… vector of *(x,y,z)*-values
… as range/intensity images
… as oc-trees

- Point Cloud reduction using Oc-trees
- Now: 3D Point Cloud reduction using range/intensity images
- How can one resize an image?

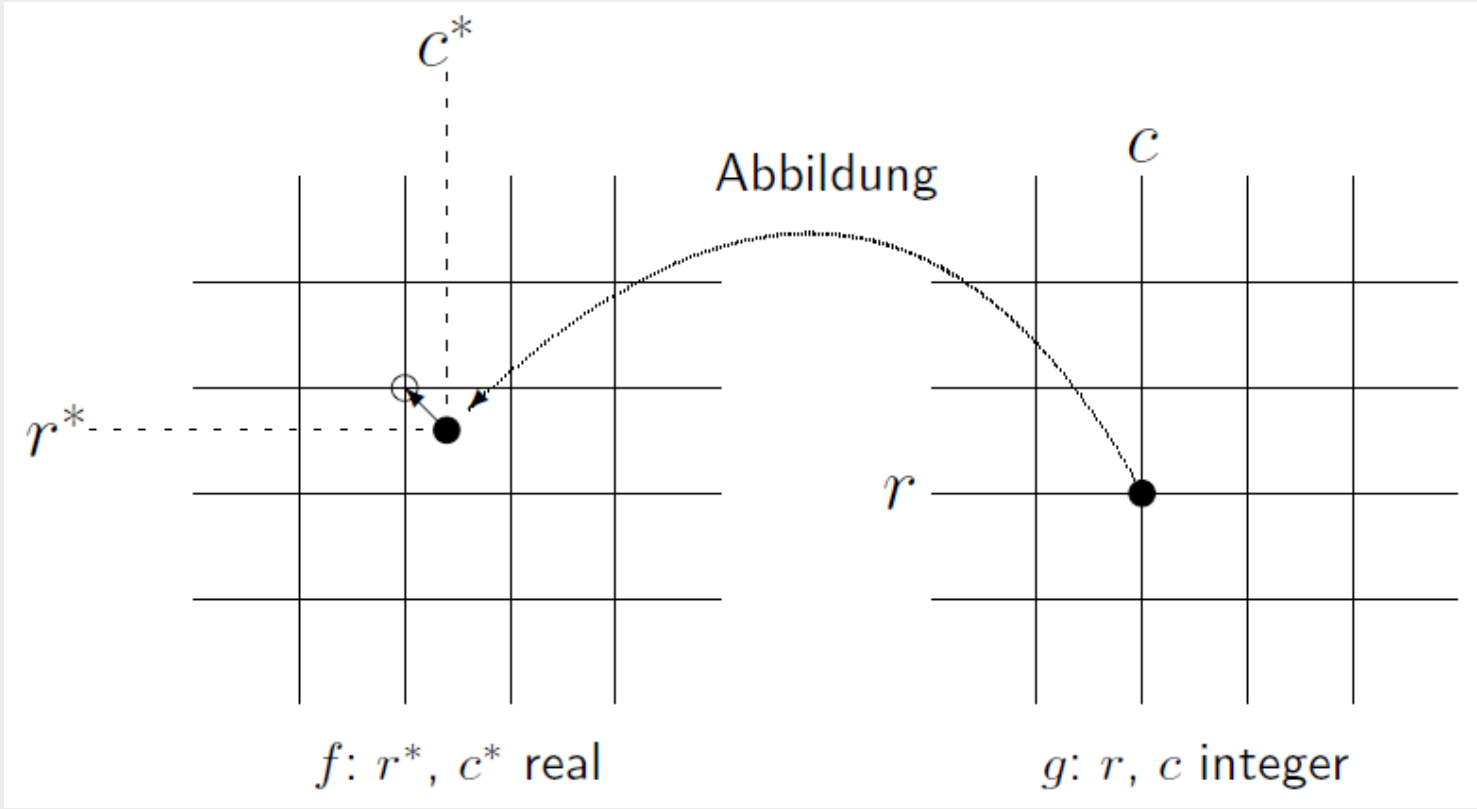
Scaling Images

- Images represent changes in intensity
- Depth images represent changes in depth
- Filters can pick out some changes and output “filtered images”
- Scales: things that change at fine scales are changing rapidly
- Idea: Build a representation that focuses on changes

  → Fourier Transform

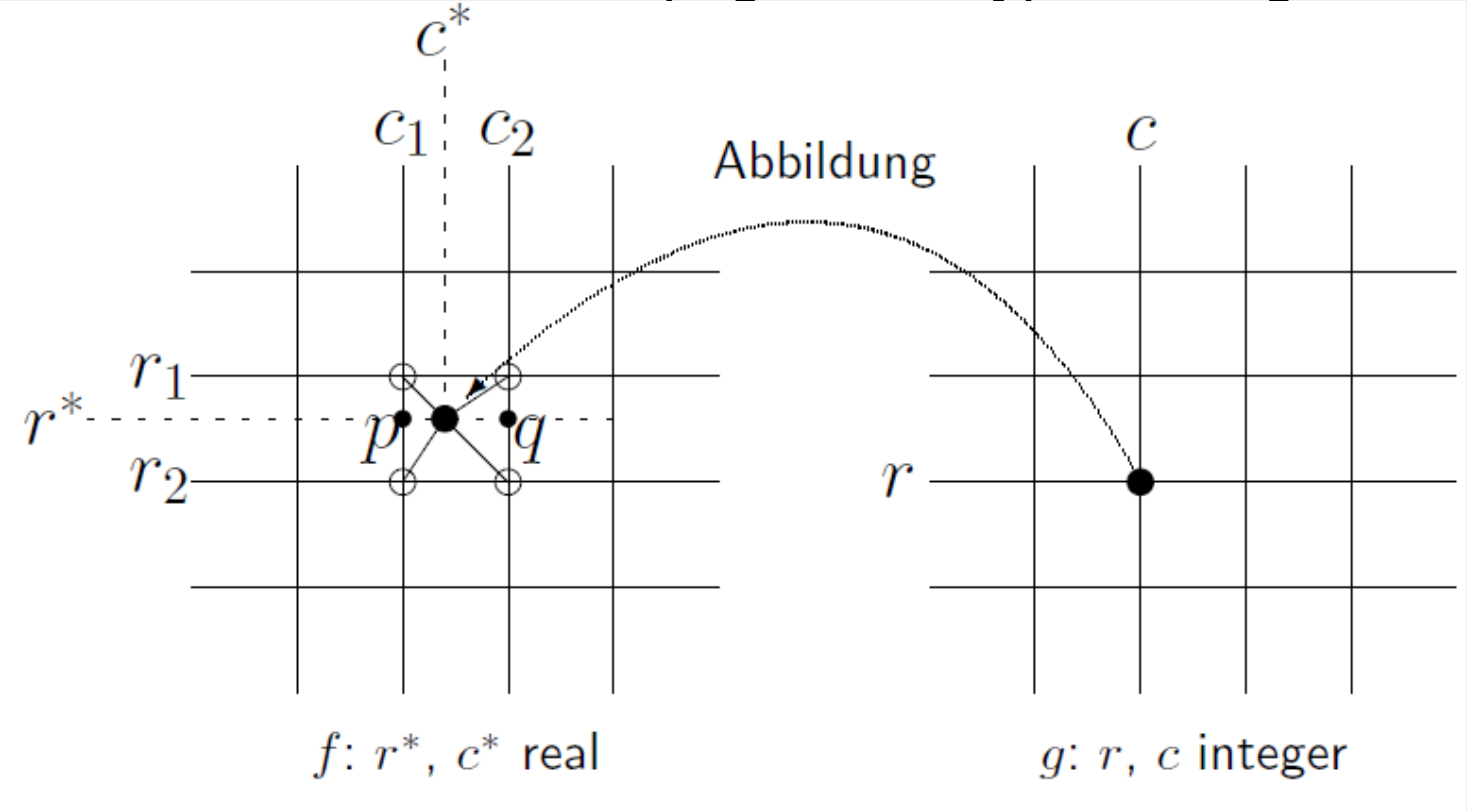
- standard scaling algorithms are **nearest-neighbor**, **bilinear** and **bicubic** interpolation

Nearest Neighbor Interpolation



Bilinear Interpolation

- is an extension of linear interpolation for interpolating functions of two variables (e.g., x and y) on a regular 2D grid.



- f is known at (0, 0), (0, 1), (1, 0), and (1, 1)

$$f(x,y) \approx \begin{bmatrix} 1-x & x \end{bmatrix} \begin{bmatrix} f(0,0) & f(0,1) \\ f(1,0) & f(1,1) \end{bmatrix} \begin{bmatrix} 1-y \\ y \end{bmatrix}.$$

Applications to 3D Point Clouds

- To Reduce an image we could
  - (1) Create a range image
  - (2) Downsample the range image (and the intensity image)
  - (3) Convert the range image back to a 3D Point Cloud
    - This implies implementing the inverse transformations of the image generation

```
bin/scan_red -s 0 -e 0 -f RIEGL_TXT
        --reduction RANGE --projection EQUIRECTANGULAR
        --scale 0.5 --width 3600
        --height 1000 ~/dat/bremen_city/
```

Either scale the range image and do the inverse mapping, or put the 3D points into a pixel and use inteprolation

```
bin/scan_red -s 0 -e 0 -f RIEGL_TXT
        --reduction INTERPOLATE --projection EQUIRECTANGULAR
        --scale 0.5 --width 3600
        --height 1000 ~/dat/bremen_city/
```