

Voxels, Point Clouds, and Registration

Richard (Hao) Zhang

CMPT 464/764: Geometric Modeling in Computer Graphics

Lecture 6

Outline

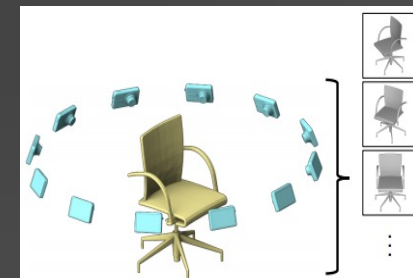
- Implicit representations
- Parametric representations
- Meshes (subdivision)
- Point clouds
- Voxels
- Multi-view representations
- Structured representations

Smooth curves and surfaces

Discrete representations

2D $\rightarrow$ 3D

Parts + relations = structures



Outline

- Implicit representations

- Parametric representations

- Meshes (subdivision)

Smooth curves and surfaces

- Point clouds

- Voxels

Discrete representations

- Multi-view representations

- Structured representations

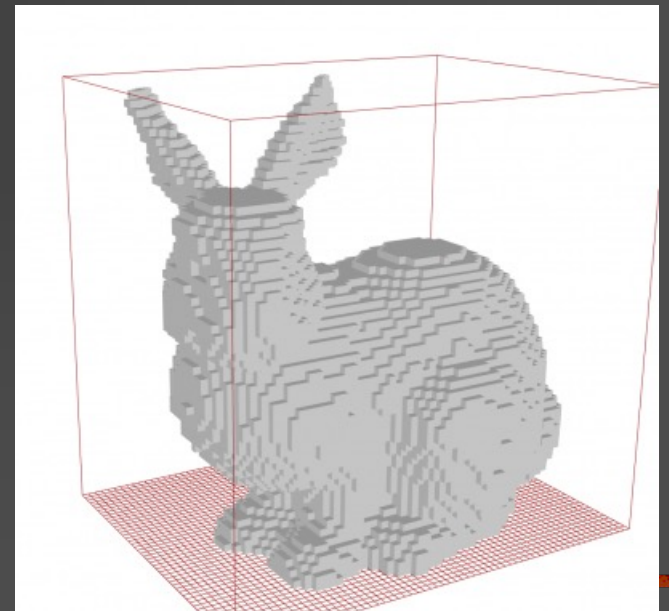
2D $\rightarrow$ 3D

Parts + relations = structures



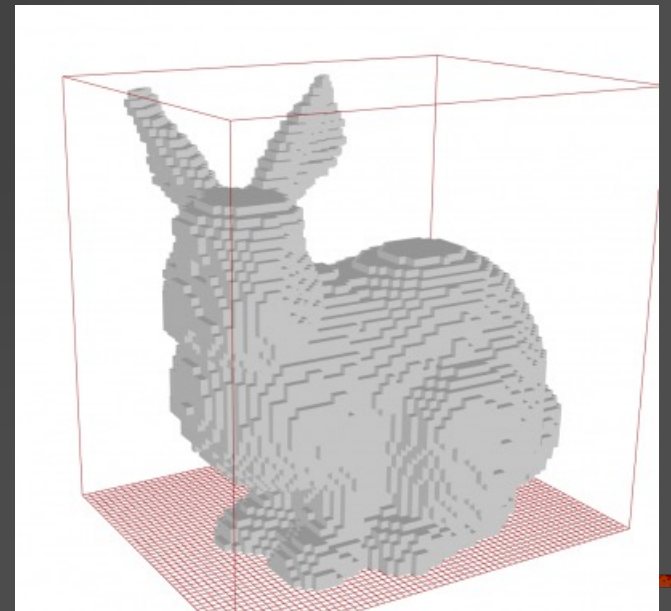
Volumetric or voxel representations

- Embed 3D shape in a regular volumetric grid: 3D shape given as a **volume** = set of all voxels that lie **on or inside** the shape
- Closely related to image and pixel representations: it is a **3D image**
- Closely tied to **implicit representations** and support similar operations



Volumetric or voxel representations

- Embed 3D shape in a regular volumetric grid: 3D shape given as a **volume** = set of all voxels that lie on or inside the shape
- Closely related to image and pixel representations: it is a **3D image**
- Closely tied to implicit representations and support similar operations
- Natural **“first choice” for neuralization** due to similarity to images
- But unlike images, internal patterns or structures mostly overlooked

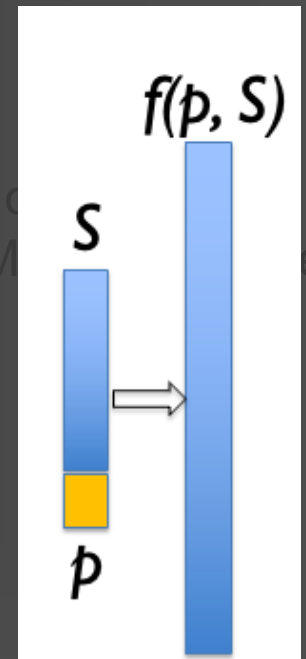


Voxels vs. implicit functions

- Voxels: intrinsically **discrete** representation, limited by resolutions
- Implicit functions: intrinsically **continuous** representation
 - There is an implicit field value for any (x, y, z)
 - It is when processing an implicit function (e.g., for rendering), one often need to discretize the function through voxelization, such as Marching Cubes

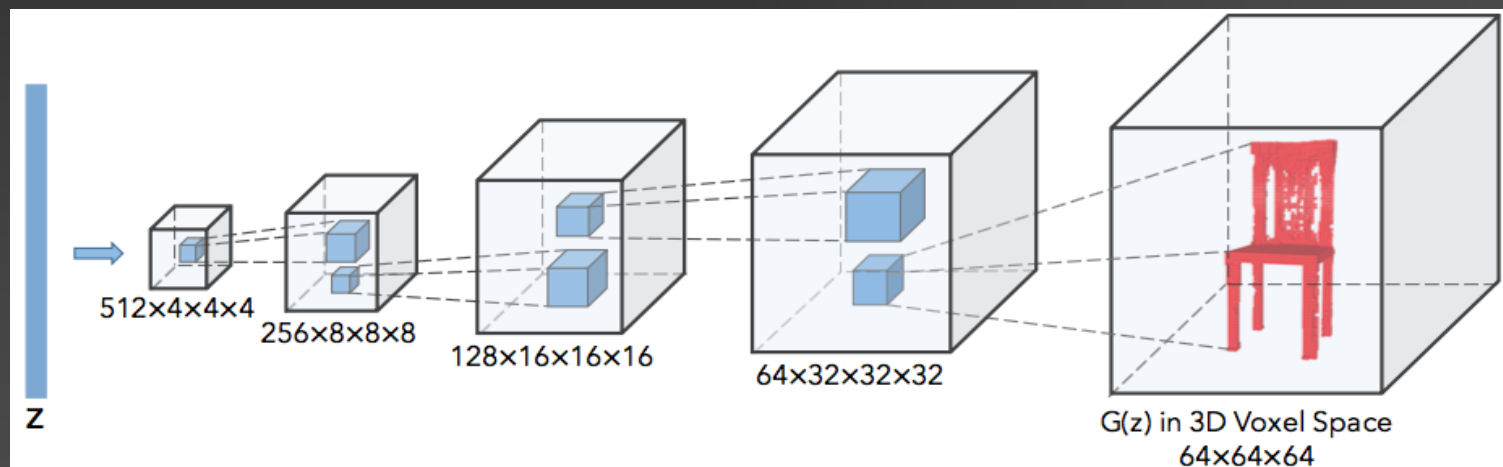
Voxels vs. implicit functions

- Voxels: intrinsically **discrete** representation, limited by resolutions
- Implicit functions: intrinsically **continuous** representation
 - There is a field value for any point
 - It is when processing an implicit function (e.g., for rendering), one needs to discretize the function through voxelization, such as Marching Cubes
- IM-Net: trained on voxel inputs, e.g., on 64^3 voxels, but can learn **continuous** outputs, $\forall p \in \mathbb{R}^3$



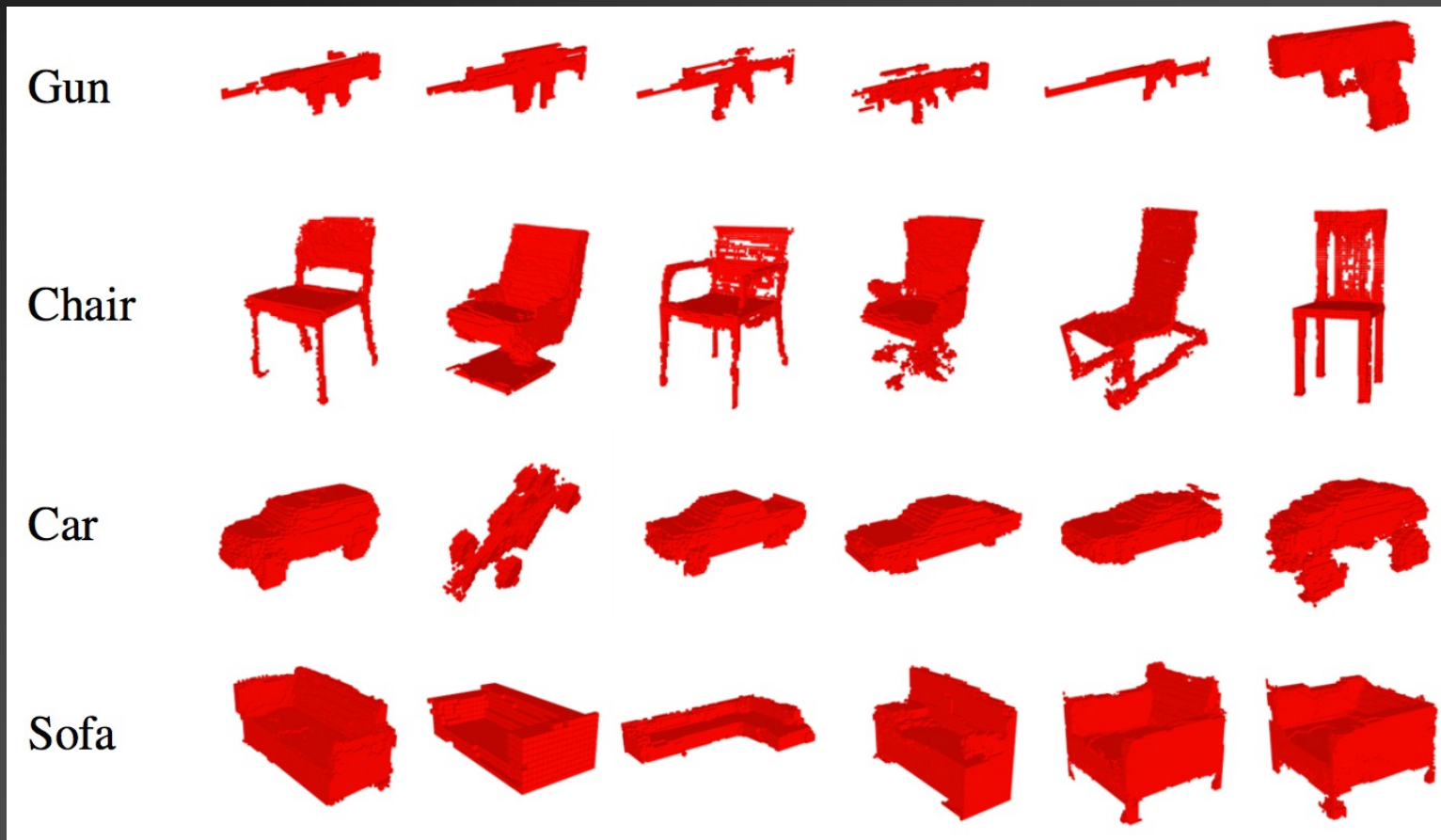
Voxel CNNs in a generative model

- 3D shape as voxels: combine volumetric CNN and GAN
- Generative network maps 200d vector to 64^3 volume



3D-GAN [Wu et al. NIPS 2016]

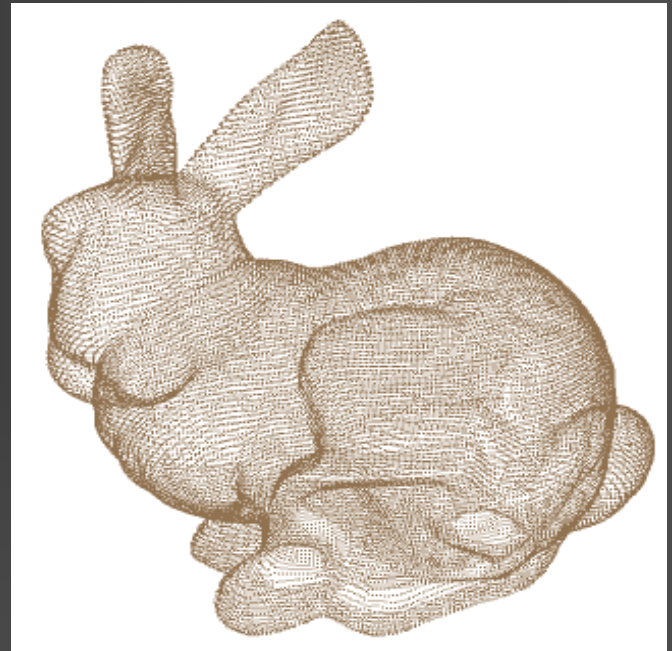
Shapes generated by 3D-GAN



3D volumetric shapes generated from random latent vectors

Point-based representation (PBR)

- A 3D surface model is represented using a set of **points near the surface**
- There is no (explicit) connectivity information between the points
- Typically need k NN – **k nearest neighbors** – during processing
- Point normals can also be specified or estimated for rendering



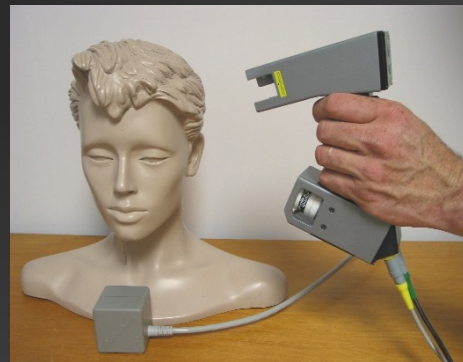
Point cloud acquisition



Cyberware



InSpeck



FastScan (\$23K)



NextEngine (< \$3K)



Roland DGA LPX-250 (\$10K)

Point cloud acquisition on mobile phones

You Prefer Which One? 🤔

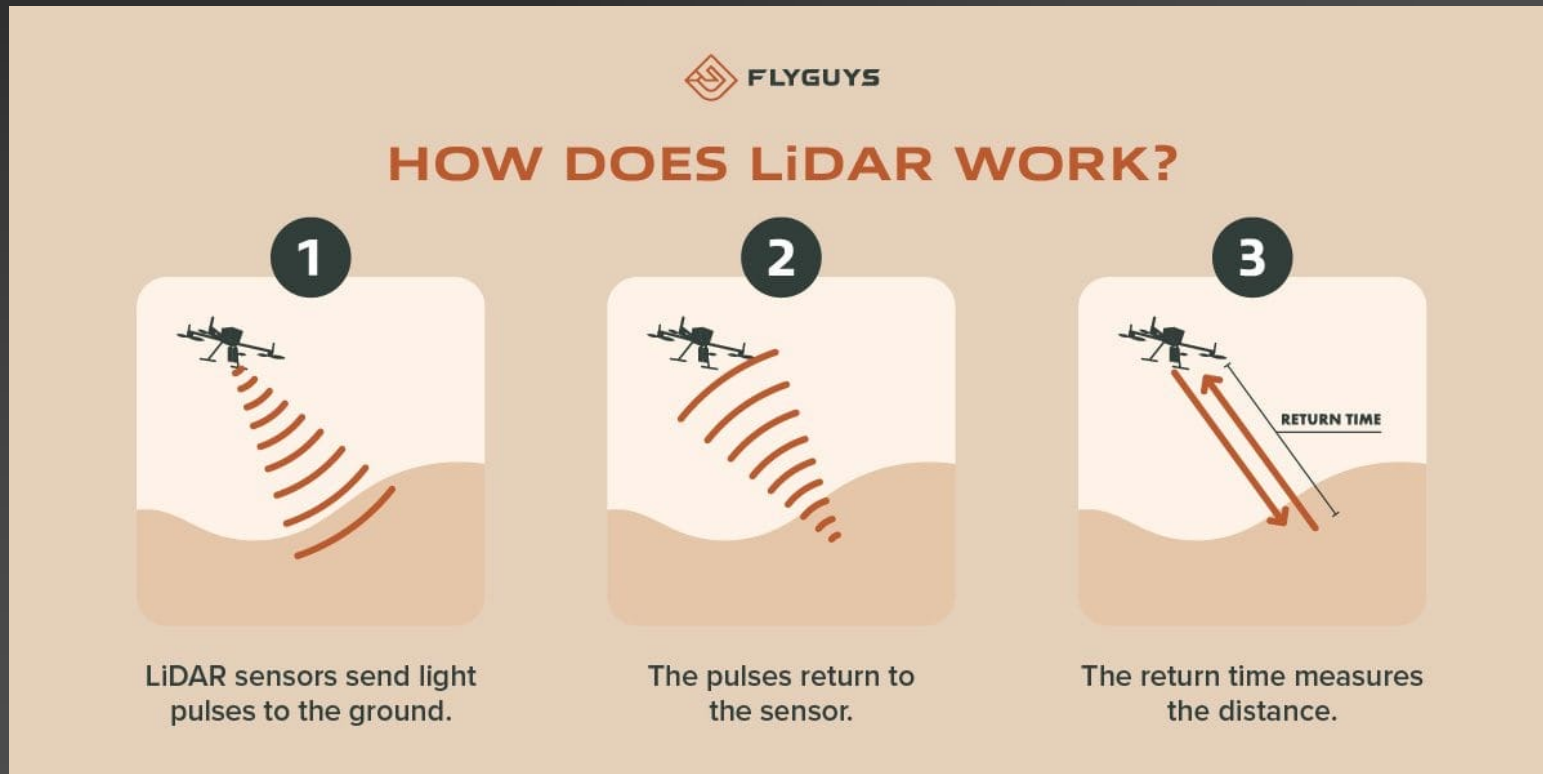
iPhone Pro CAMERA EVOLUTION

iPhone 16 Pro vs iPhone 17 Pro vs iPhone 18 Pro

Model	Ultra Wide	Telephoto	Main	Special Feature
iPhone 16 Pro	48 MP 1/2.2" 13mm FOV120° Macro	12 MP 1/2.8" 120mm 5x	48 MP 1/1.78" 24mm	LIDAR Scanner
iPhone 17 Pro	48 MP 1/2.2" 13mm FOV120° Macro	48 MP 1/2.8" 100mm 4x 3x Optical Quality Zoom	48 MP 1/1.78" 24mm	LIDAR Scanner
iPhone 18 Pro	48 MP 1/2.2" 13mm FOV120° Macro	48 MP 1/2.8" up to 8x Optical Zoom	48 MP Primary (Main) 1/4.0" up to 40x Digital Zoom, up to 18x Optical Zoom	LIDAR Scanner

LiDAR

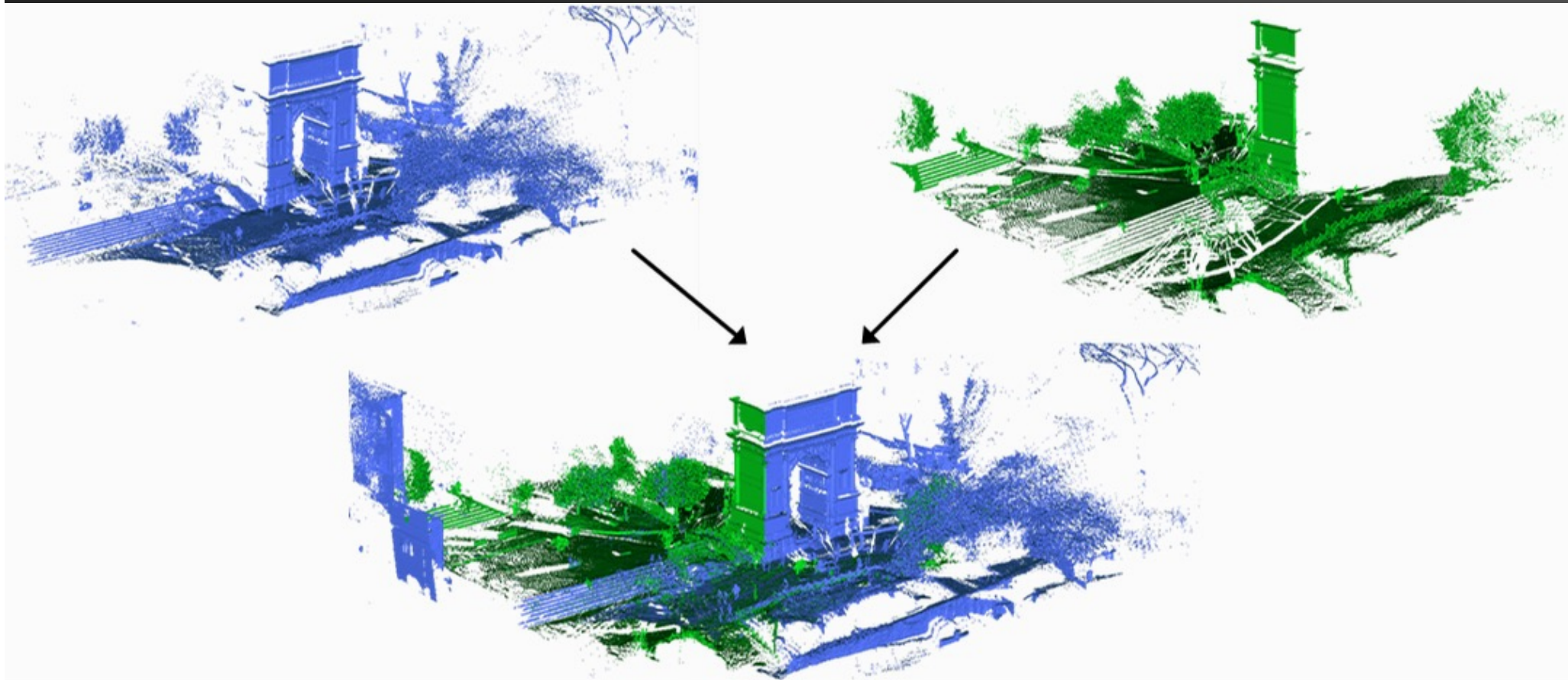
- LiDAR = Light Detection and Ranging



Use of LiDAR in autonomous driving

Company/system	Uses LiDAR?	Sensor philosophy
Waymo	Yes	LiDAR + cameras + radar
Zoox	Yes	LiDAR + cameras + radar
Aurora	Yes	Proprietary long-range LiDAR + cameras + imaging radar
Baidu Apollo / Apollo Go	Yes	LiDAR-centric multi-sensor fusion
Pony.ai	Yes	LiDAR + cameras + radar
WeRide	Yes	LiDAR + cameras + radar
Mobileye Drive / Chauffeur	Yes	Camera system plus an independent radar/LiDAR system
Mercedes DRIVE PILOT	Yes	LiDAR + radar + cameras, with substantial redundancy
Tesla FSD	No	Primarily cameras + neural networks ("Tesla Vision")

Point cloud registration

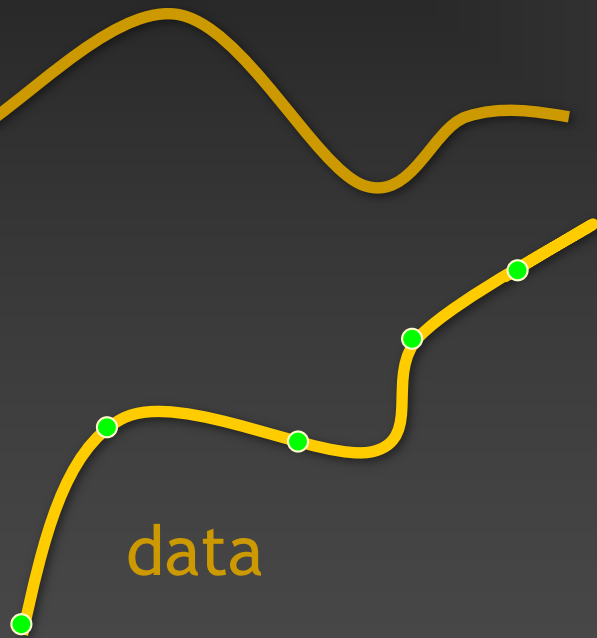


Iterative closest point (ICP) algorithm

- A classic registration/correspondence schemes
 - Input: **data** and **model shapes**
 - Objective:
 - Rigid transform = rotation + translation
 - **Minimize mean squared error** from data points to closest points in model [Besl and McKay 92]
 - Correspondence obtained by Euclidean proximity
-

ICP algorithm

model

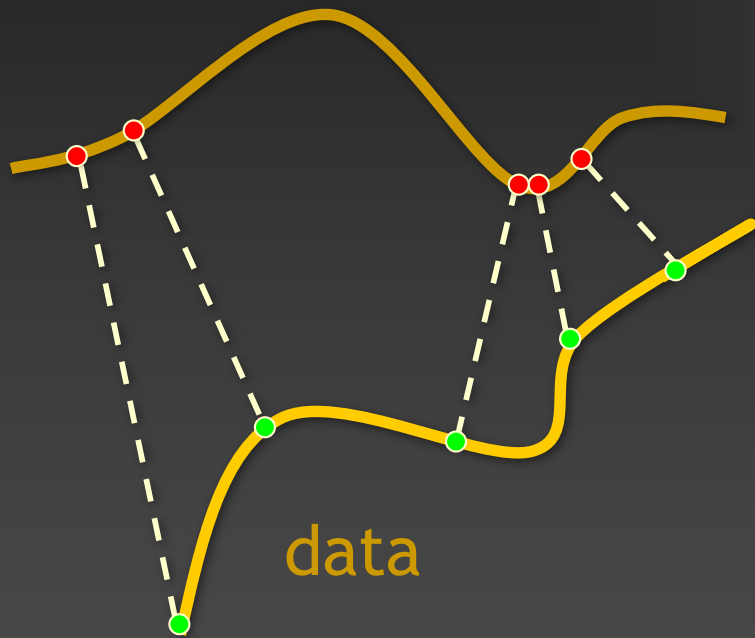


data

Model and data shapes (point samples)

ICP algorithm

model

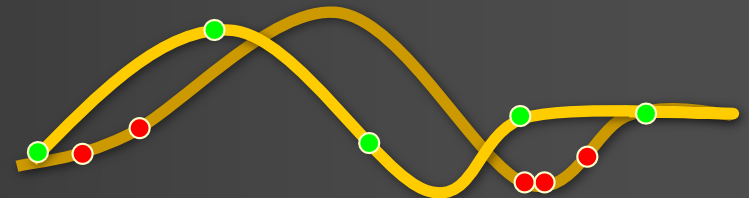
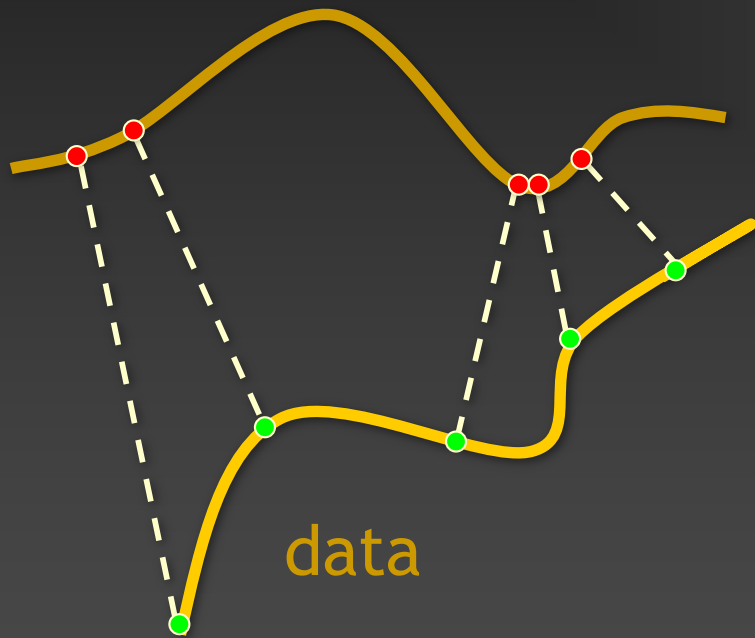


data

Find closest points from data to model

ICP algorithm

model



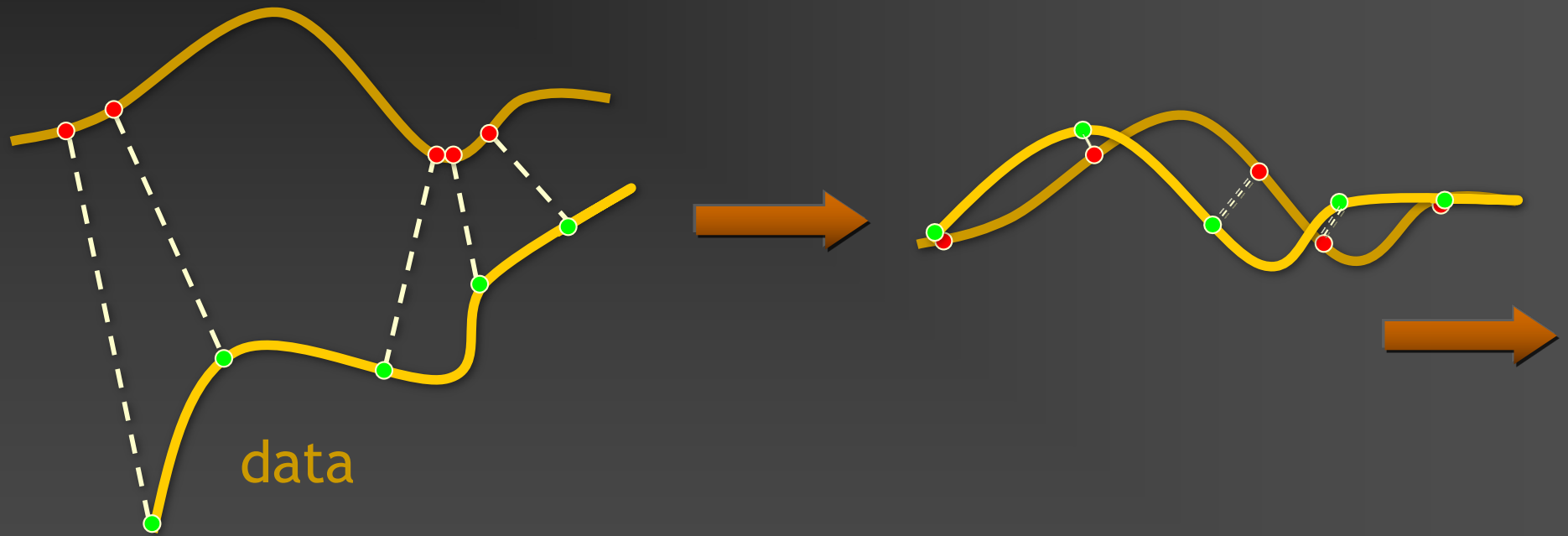
Find best rigid transform to align the corresponding points

ICP algorithm

model

data

Iterate ...



A historical note on PBRs

- “As the visual complexity of computer-generated scenes continue to increase, the use of classical modeling primitives (polygons) as display primitives becomes less appealing.”

Levoy and Whitted,
“The Use of Points as a Display Primitive”, 1985

- Use of points traces back to modeling of smoke, fire, and cloud around the late 70's [Csuri et al. 79, Blinn 82]

First renewed interest

- PBR was witnessing a revival around 2002 - 2006
- Points are directly available via **laser scanning**
 - Substantial advances in 3D digitizing and laser scanning and acquisition technology
 - High quality points (color and texture) easily obtained
 - Relatively inexpensive laser scanners available

Point-based Graphics Resource

Symposium on Point-Based Graphics

[Symposium on Point-Based Graphics 2006 \(On-line Papers\)](#) ***
[Symposium on Point-Based Graphics 2006 \(On-line Papers\)](#)
[Symposium on Point-Based Graphics 2005 \(On-line Papers\)](#)
[Symposium on Point-Based Graphics 2004 \(On-line Papers\)](#)

Books

Point-Based Graphics

Edited By [Markus Gross](#), Swiss Federal Institute of Technology (ETH), Zurich, Switzerland [Hanspeter Pfister](#), MERL (Mitsubishi Electric Research Laboratories), Cambridge, Massachusetts, U.S.A.

Tutorials/Course Notes/Survey

Point-Based Computer Graphics



[Marc Alexa](#), [Markus Gross](#), [Mark Pauly](#), [Hanspeter Pfister](#), [Marc Stamminger](#), [Matthias Zwicker](#)
SIGGRAPH 2004 Course Notes

Point-Based Computer Graphics



[Marc Alexa](#), [Carsten Dachsbacher](#), [Markus Gross](#), [Mark Pauly](#), [Jeroen van Baar](#), [Matthias Zwicker](#)
Eurographics 2003 Tutorial Notes

Point-Based Computer Graphics



[Marc Alexa](#), [Markus Gross](#), [Mark Pauly](#), [Hanspeter Pfister](#), [Marc Stamminger](#), [Matthias Zwicker](#)
Eurographics 2002 Tutorial Notes

A Survey of Point-Based Techniques in Computer Graphics



[Leif Kobbelt](#), [Mario Botsch](#)
to appear in Computers & Graphics

Dissertation/Thesis

Point-Based Modeling, Animation and Rendering of Dynamic Objects



[Bart Adams](#)
PhD Thesis, Katholieke Universiteit Leuven, May 2006

Point-Based Multi-Resolution Rendering



[Michael Wand](#)
PhD Thesis, University of Tübingen, Department of Computer Science and Cognitive Science, 2004

Dynamic Point Samples as Primitives for Free-Viewpoint Video



[Stephan Wurmlin](#)
PhD Thesis, ETH Zurich, 2004

Reconstruction and Rendering of Implicit Surfaces from Large Unorganized Point Sets



[Patrick Reuter](#)
PhD Thesis, LaBRI - Université Bordeaux-1, 2003

Point Primitives for Interactive Modeling and Processing of 3D Geometry



[Mark Pauly](#)
PhD Thesis, ETH Zurich, 2003

Continuous Reconstruction, Rendering, and Editing of Point-Sampled Surfaces



[Matthias Zwicker](#)
PhD Thesis, ETH Zurich, 2003

Point Set Surfaces

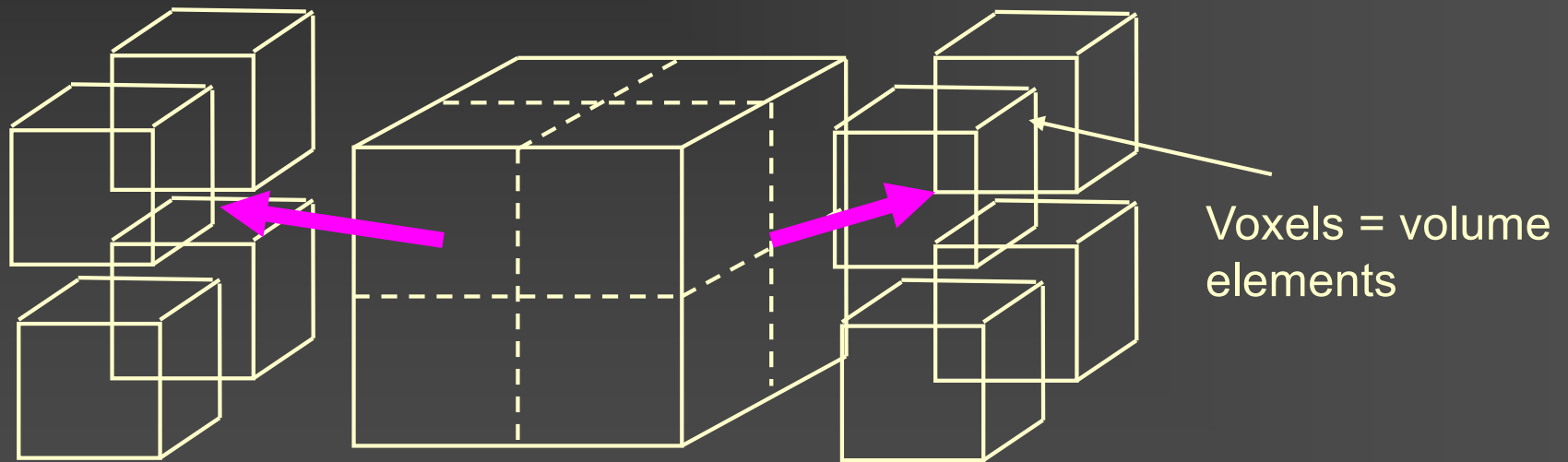


[Shachar Fleishman](#)
PhD Thesis, 2003

<https://www.realtimerendering.com/kesen/PointBasedPaper.html>

Hierarchical PBRs

- Use of **octree** – a spatial data structure
- 3D generalization of binary and quad tree

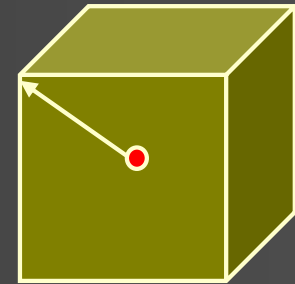


- Hierarchy leads to compact PBR [Botsch et al. 02]

Compact PBR: setup and key point

- Setup:
 - Points located at **voxel centers** – fixed increments from one sample to the next
 - Use of **binary volume grids**
- Key: most significant bits are **reused** by many nearby samples → memory efficiency
- To achieve real efficiency, e.g., 2.67 bits/sample, we require **sufficiently dense** point set!

Binary volume grid

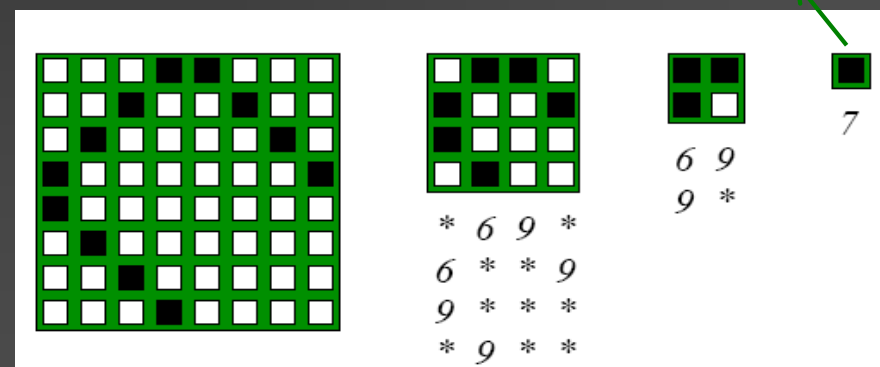
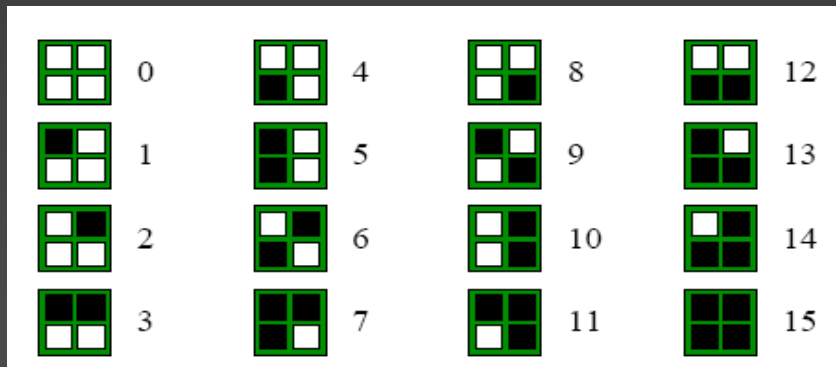


- Divide bounding cube of surface into n^3 grids
- At most one point sample per voxel is located at center
- Approximation error is bounded by $\sqrt{3}h/2$ – note points are clamped to nearest voxel centers
- Each voxel is either full (1) or empty (0)
- Voxel is 1 if and only if surface intersects the voxel

Octree rep: use quadtrees to illustrate

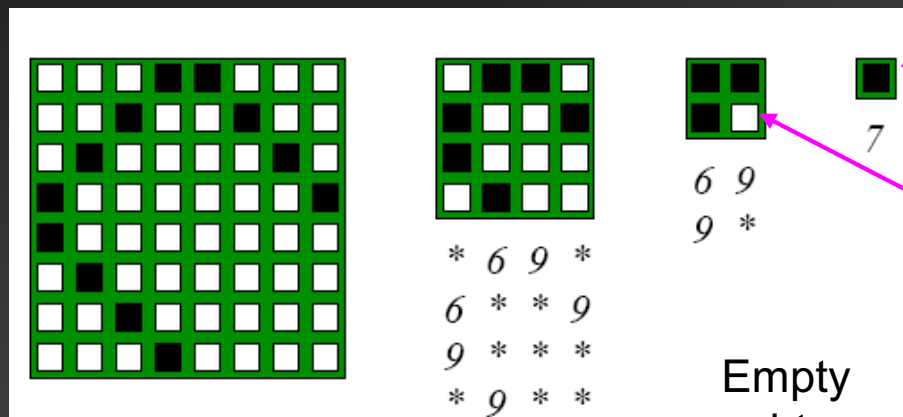
- Naïve storage needs n^3 bits in a 3D array, but only $O(n^2)$ 1's for a typical 2-manifold surface
- Utilize the usual octree/quadtree representation
 - 1 byte per octree node indicating status of children
 - 1 $\leftrightarrow$ at least one full voxel in child and 0 otherwise

Four children, one is empty. So three 1's and one 0. Pattern formation gives index 7



Quad-tree needs a half-byte per node

Example: shown as quad-tree



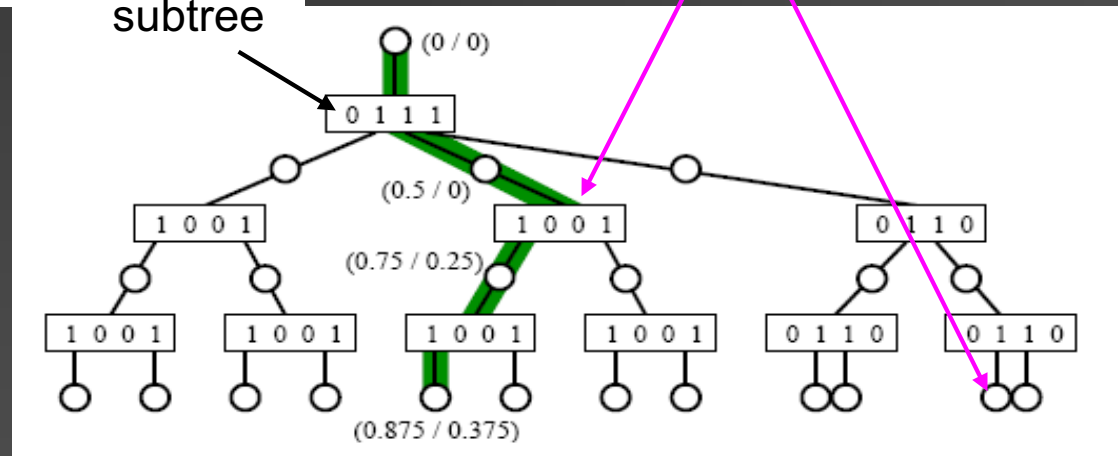
Every step along the path from root adds additional precision bit to a coordinate

On = 1

Naïve: 64 bits to store info about any binary 8x8 grid; 12 points, so $64/12 = 5.3 \text{ b/s}$

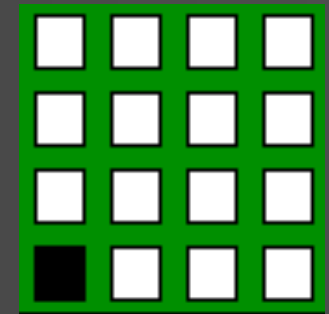
Off = 0

10 q-tree nodes: $4 \cdot 10/12 = 3.3 \text{ b/s}$



How efficient is the tree rep? (aside)

- Let quantization level be q , e.g., $q = 12$ (bits) if the grid is $2^{12} \times 2^{12} \times 2^{12}$
- One sample (a chain), q levels, 8 bits per level
 $8q$ b/s – not very good
- All voxels are full = full tree:
 - $2^q \times 2^q \times 2^q = 8^q$ samples
 - Number of bits: $8 + 8^2 + 8^3 \dots + 8^q = 8(8^q - 1)/7$
 - About $8/7 = 1.142857$ b/s
 - That is pretty good!



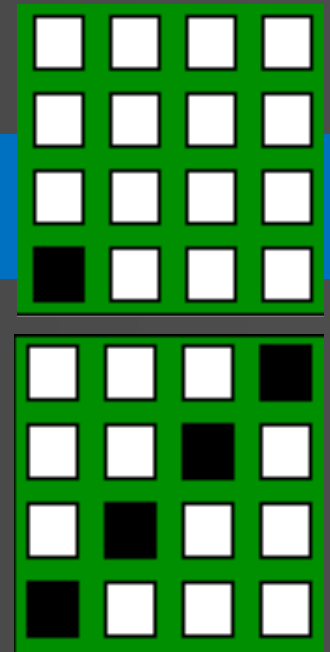
How efficient is the tree rep? (aside)

- Each $2 \times 2 \times 2$ block has 1 sample – kind of sparse
 - Use the same number of bytes as before
 - But only 1/8 of the points
 - $64/7 = 9.142857$ b/s – oh oh!
- Bit rate may **not** depend on quantization levels
- Density of point samples is a contributing factor

For 2-manifold surface ... (aside)

- Expected number of samples m = the expected number of voxels intersected by a 2-manifold surface is $O(n^2)$ – acknowledge this ☺
- Expected number of voxels in a non-empty $2 \times 2 \times 2$ block intersected by a 2-manifold surface is 4. To represent m samples, need $m/4$ full voxels, each using 8 bits to encode status of a $2 \times 2 \times 2$ block.
- Going to next coarser level, # full voxels is expected to be reduced by a factor of $2^2 = 4$, as n is decreased by 2

For 2-manifold surface ... (aside)



- Let m be number of samples. Total number of full voxels =

$$m/4 + m/4^2 + \dots + m/4^q \leq m/3$$

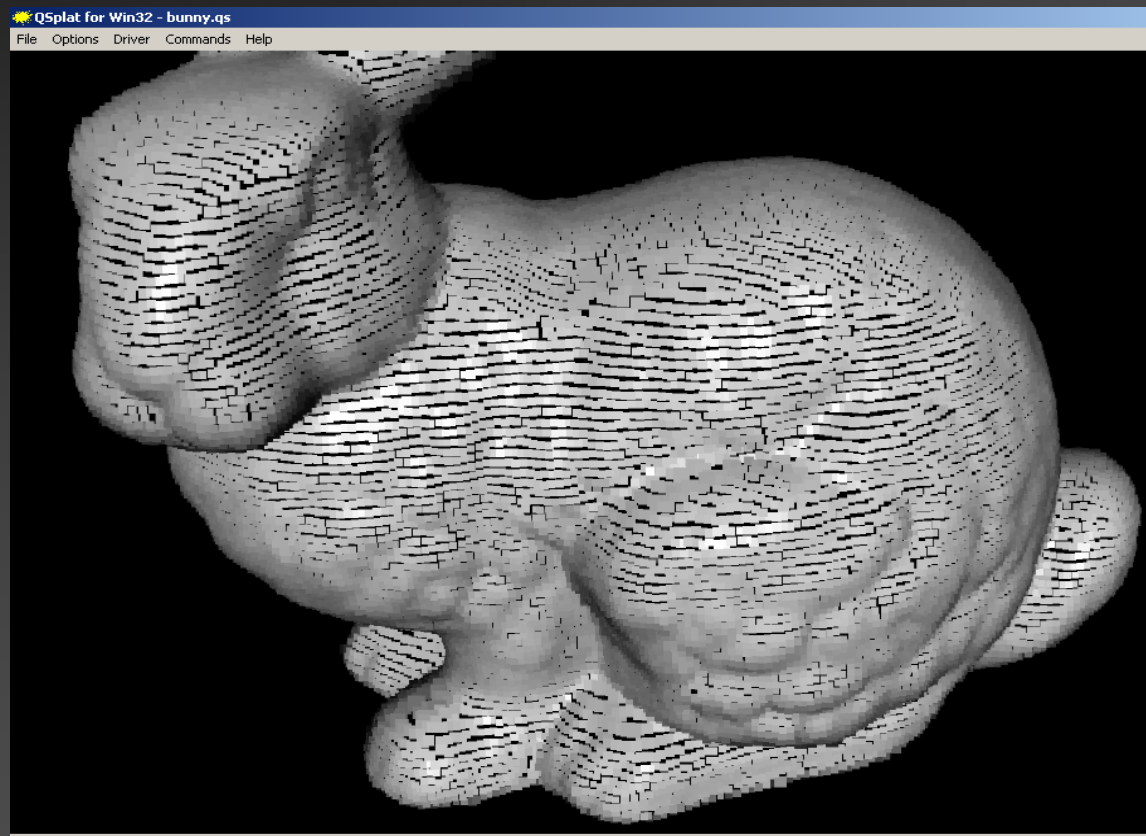
- With 8b to encode each full voxel: bit rate $\leq 8 \times (m/3) / m = 2.67$ b/s!
- Further compression:
 - Exploit **nonuniform distribution** of byte-code for the full pixels (4 out of 8 occupied is more likely)
 - Recall Huffman coding: less bits for the more frequent

Hierarchical PBR rendering algorithm

```
TraverseHierarchy(node)
{
    if (node not visible)
        skip this branch of the tree
    else if (node is a leaf node)
        draw a splat
    else if (benefit of recursing further is too low)
        draw a splat
    else
        for each child in children(node)
            TraverseHierarchy(child)
}
```

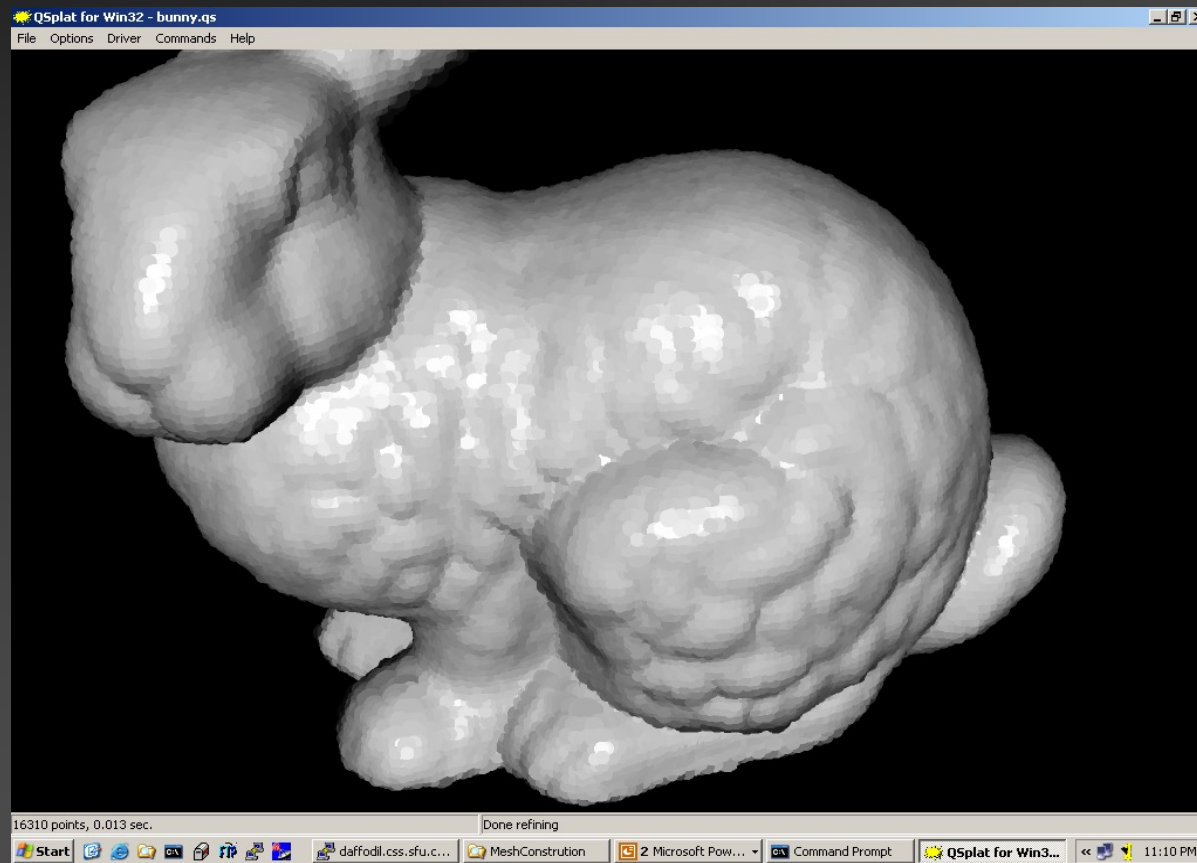
Qsplat

PBR rendering via splatting: QSplat



Splat = OpenGL
points

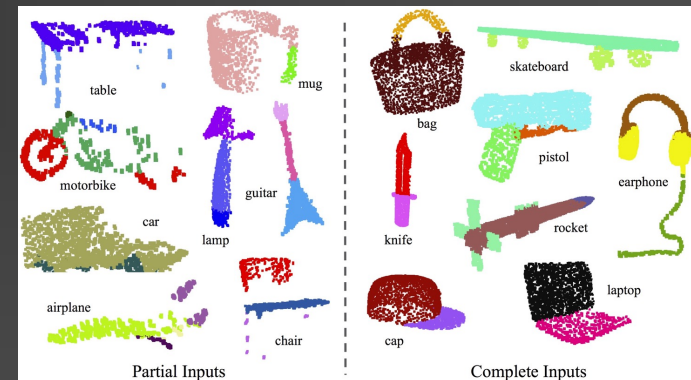
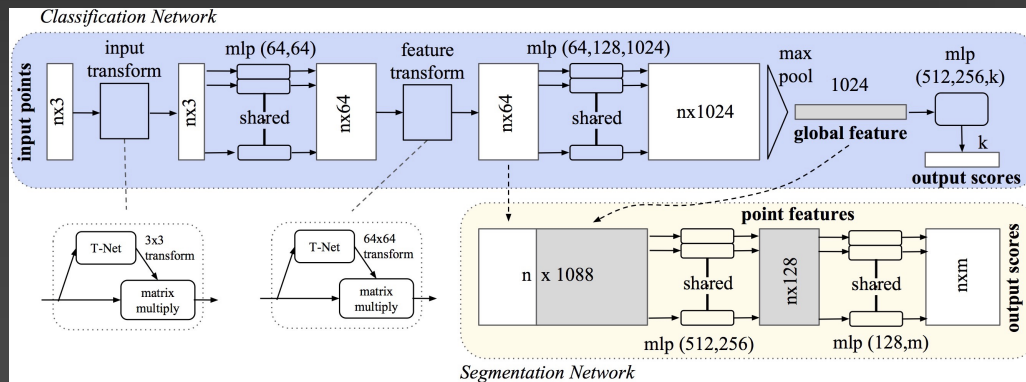
PBR rendering via splatting: QSplat



Splat = circles

Second revival: deep learning

- PointNet and PointNET++, since 2016/17
- Deep neural network to **encode and aggregate point features** for shape recognition, segmentation, etc. (more later)





Leonidas Guibas

Professor of Computer Science, [Stanford University](#)
Verified email at cs.stanford.edu

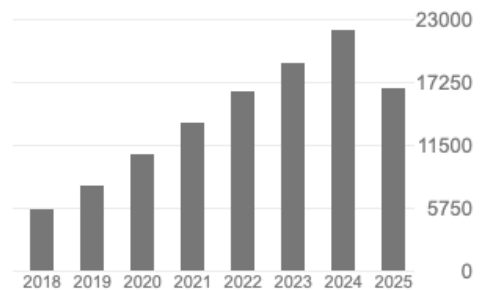
[geometric computing](#) [computer vision](#) [computer graphics](#) [machine learning](#) [robotics](#)



Cited by

[VIEW ALL](#)

	All	Since 2020
Citations	164209	98953
h-index	158	102
i10-index	592	390



Public access

[VIEW ALL](#)

4 articles

189 articles

not available

available

Based on funding mandates

Co-authors

[VIEW ALL](#)

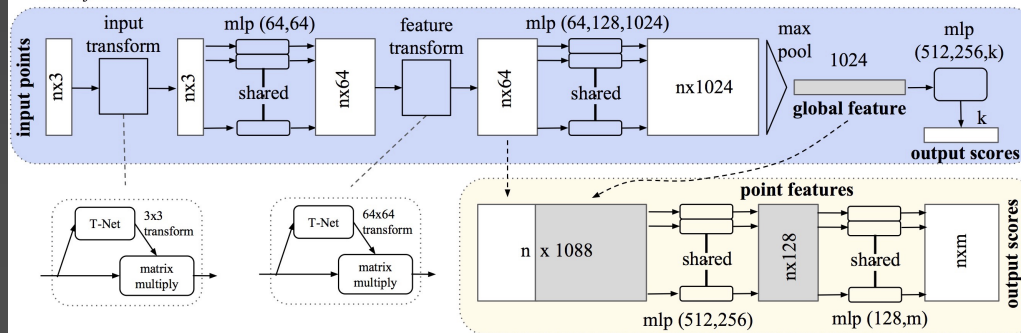


Hao Su
UCSD CSE

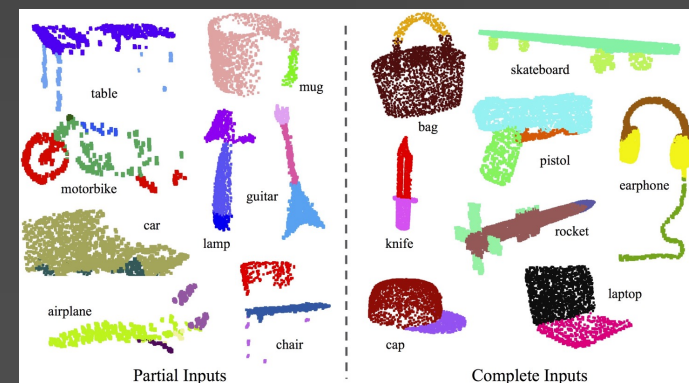


TITLE	CITED BY	YEAR
Pointnet: Deep learning on point sets for 3d classification and segmentation CR Qi, H Su, K Mo, LJ Guibas Proceedings of the IEEE conference on computer vision and pattern ...	21564	2017
Pointnet++: Deep hierarchical feature learning on point sets in a metric space CR Qi, L Yi, H Su, LJ Guibas Advances in neural information processing systems 30	16028	2017
Shapenet: An information-rich 3d model repository AX Chang, T Funkhouser, L Guibas, P Hanrahan, Q Huang, Z Li, ... arXiv preprint arXiv:1512.03012	7141	2015
The earth mover's distance as a metric for image retrieval Y Rubner, C Tomasi, LJ Guibas International journal of computer vision 40 (2), 99-121	7138	2000
Kpconv: Flexible and deformable convolution for point clouds H Thomas, CR Qi, JE Deschaud, B Marcotegui, F Goulette, LJ Guibas Proceedings of the IEEE/CVF international conference on computer vision ...	3725	2019
Frustum pointnets for 3d object detection from rgb-d data CR Qi, W Liu, C Wu, H Su, LJ Guibas Proceedings of the IEEE conference on computer vision and pattern ...	3184	2018
A point set generation network for 3d object reconstruction from a single image H Fan, H Su, LJ Guibas Proceedings of the IEEE conference on computer vision and pattern ...	3014	2017

Classification Network



Segmentation Network



Third revival: 3D Gaussian splatting (3DGS)

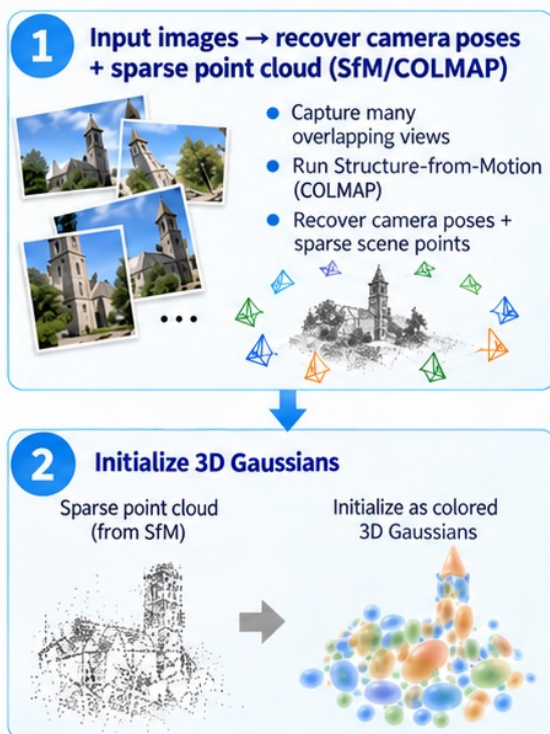
- 3DGS [Kerbl et al. SIG 2023] after NeRF [Mildenhall et al. ECCV 2000]





3D Gaussian Splatting (3DGS): Pipeline Overview

Key steps from images to real-time rendering

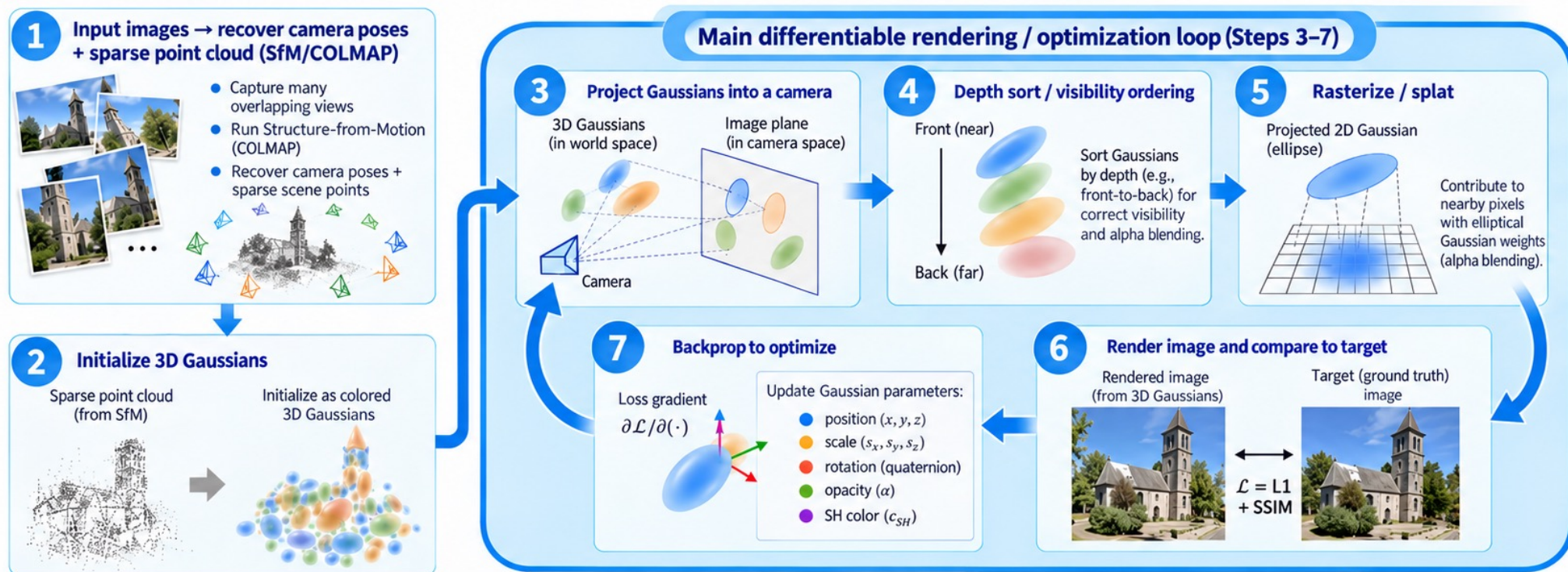


Good online source: <https://learnopencv.com/3d-gaussian-splatting/>



3D Gaussian Splatting (3DGS): Pipeline Overview

Key steps from images to real-time rendering

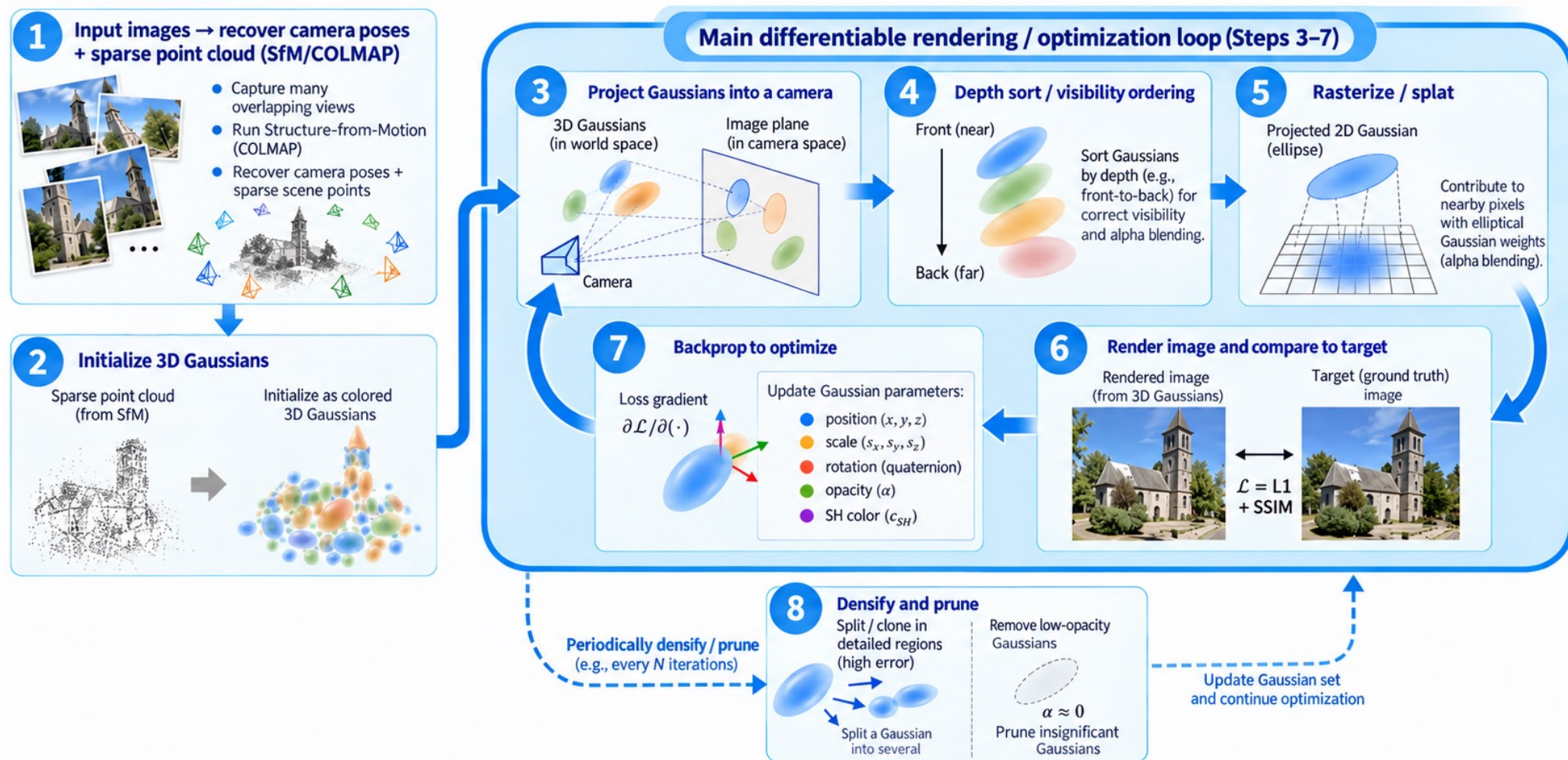


Good online source: <https://learnopencv.com/3d-gaussian-splatting/>



3D Gaussian Splatting (3DGS): Pipeline Overview

Key steps from images to real-time rendering



Good online source: <https://learnopencv.com/3d-gaussian-splatting/>



3DGS Step 1–3: From Images to Projected Gaussians (aside)

Initialization and camera-space projection

1

Input images + camera poses
(SfM/COLMAP)



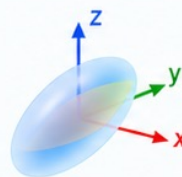
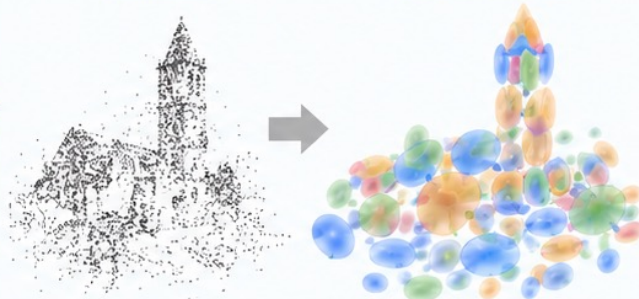
- 1 Capture many **calibrated views**
- 2 Run **Structure-from-Motion**
- 3 Estimate **camera poses + sparse scene points**

2

Initialize 3D Gaussians

Sparse point cloud
(from SfM)

Initialize as colored
3D Gaussians



Gaussian parameters:

- position (x, y, z)
- scale (s_x, s_y, s_z)
- rotation (quaternion)
- opacity (α)
- SH color (c_{SH})

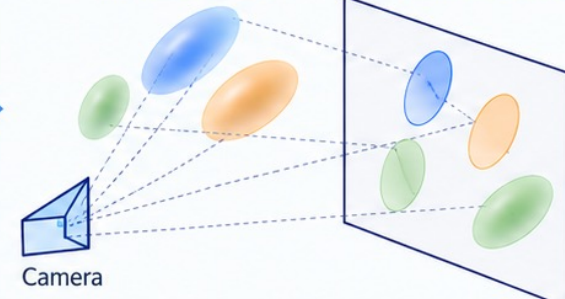
- 1 Each point becomes a **learnable 3D Gaussian**
- 2 Anisotropic covariance gives **ellipsoidal shape**
- 3 Color is view-dependent via **spherical harmonics**

3

Project into the camera

3D Gaussians
(in world space)

Image plane
(in camera space)



3D anisotropic Gaussian $\rightarrow$ 2D elliptical footprint

$$\mathcal{N}(\mu, \Sigma) \xrightarrow{\text{proj.}} \mathcal{N}(u, \Sigma_{2D})$$

- 1 Transform to **camera space**
- 2 Approximate **screen-space covariance**
- 3 Get a **2D ellipse** per Gaussian



Key idea: 3DGS starts from SfM geometry, then turns each point into an **optimized explicit primitive**.



3DGS Step 4–7: Rendering and Optimization (aside)

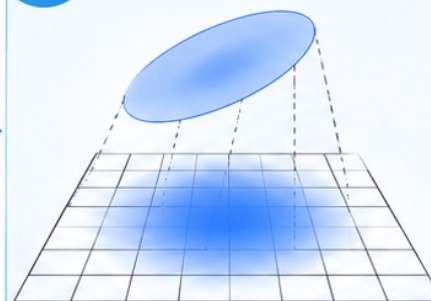
How projected Gaussians become a reconstructed image

4 Depth sort / visibility ordering



- Sort Gaussians approximately by depth
- Front-to-back compositing handles visibility
- Closer splats attenuate farther ones

5 Rasterize / splat



$$\text{weight} \propto \exp\left(-\frac{1}{2} \mathbf{x}^T \Sigma^{-1} \mathbf{x}\right)$$

- Each Gaussian splats onto nearby pixels
- Elliptical footprint gives smooth coverage
- Efficient tile-based rasterization

6 Render image and compute loss



$$\mathcal{L} = \mathcal{L}_1 + \lambda \text{SSIM}$$



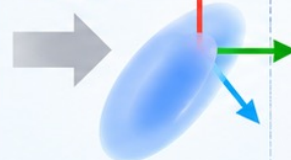
Rendered image
(from 3D Gaussians)

Target (ground truth)
image

- Composite colors with alpha blending
- Compare rendered and ground-truth images
- Use photometric reconstruction loss

7 Backprop to optimize

Loss gradient
 $\partial \mathcal{L} / \partial (\cdot)$



Update Gaussian parameters:

- position (x, y, z)
- scale (s_x, s_y, s_z)
- rotation (quaternion)
- opacity (α)
- SH coefficients (c_{SH})

- Differentiate through the rasterizer
- Update all Gaussian parameters
- Improve image reconstruction



Key idea: rendering is differentiable, so Gaussians can be optimized directly from image reconstruction error.



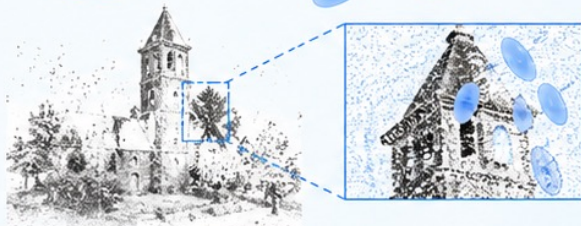
3DGS Step 8 + Training Loop (aside)

Adaptive densification, pruning, and final real-time rendering

1 Densify

Large Gaussian in high-error region

Split / clone into smaller Gaussians



- Add capacity where detail is missing
- Split or clone Gaussians with large gradients
- Helps recover thin structures and sharp detail

2 Prune

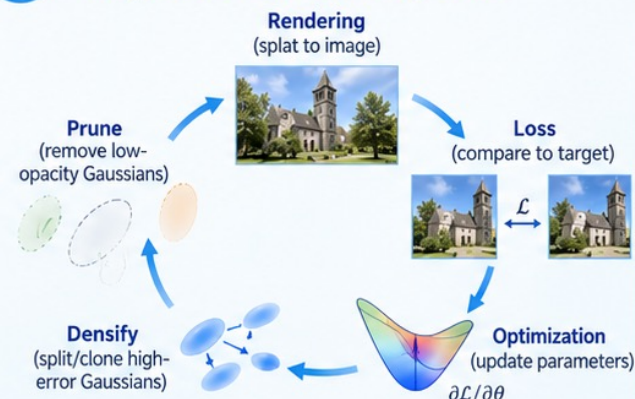
Low-opacity / insignificant Gaussians

After pruning



- Remove insignificant or nearly transparent Gaussians
- Keep the representation compact
- Improves speed and memory efficiency

3 Repeat until convergence



- Alternate optimization and structural updates
- The Gaussian set evolves during training
- Converges to an explicit scene representation

A Final representation

Millions of optimized 3D Gaussians representing the scene



B Fast novel-view rendering

Render new camera views in real time (from explicit Gaussians)



C Why 3DGS is popular



High visual quality

Reconstructs detailed, photorealistic scenes.



Explicit and editable

Direct 3D representation (Gaussians) that can be inspected and modified.



Much faster than NeRF at inference

Real-time novel-view rendering on a GPU (without per-scene optimization).



Overall recipe: initialize → project → splat → compare → optimize → densify/prune → repeat.

Good online source: <https://learnopencv.com/3d-gaussian-splatting/>