i2CAT Research Centre, Barcelona
3D/ML Senior Research Engineer
Candidate: Dr Elena Camuffo
E-mail: elenacamuffo97@gmail.com

September 19, 2025
Technical Assignment

Point Cloud Transformation and Rendering

1 Point Cloud Normalization and Interactive Transformations

1.1 Preprocessing

The system loads the input PLY file (`inputPly.ply`) into a structure of points (x, y, z) with colors (r, g, b). We compute both the centroid $\mathbf{c} = \frac{1}{N}\sum_{i=1}^{N} \vec{p}_i$ and the axis-aligned bounding box (AABB), defined by coordinate-wise minima and maxima [1]. From the AABB we derive a scale factor $s = 2/\max\{x_{\max} - x_{\min},\, y_{\max} - y_{\min},\, z_{\max} - z_{\min}\}$, so that the normalized cloud fits in $[-1, 1]^3$. The cloud is translated by $-\mathbf{c}$ and scaled by s, centering it at the origin and normalizing its extent.

For later use, we cache centroid, AABB and approximate normals. Normals are estimated as $\mathbf{n}_i = (\vec{p}_i - \mathbf{c})/\|\vec{p}_i - \mathbf{c}\|$, i.e., radial vectors from the centroid [2]. Caching avoids repeated $O(N)$ passes, improving interactivity.

1.2 Transformations

We support rigid and non-rigid operations:

- **Translation:** $\vec{p}' = \vec{p} + \Delta\vec{t}$, applied with keys (WASD/RF).
- **Rotation:** Euler angles $(\theta_x, \theta_y, \theta_z)$, with standard axis matrices [3], e.g.

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}.$$

- **Displacement along normals:** $\vec{p}' = \vec{p} + d\,\mathbf{n}_i$, expanding/contracting the cloud [4].
- **Symmetric displacement:** offset proportional to horizontal deviation: $x_i' = x_i + \lambda\,|x_i - c_x|$, preserving symmetry w.r.t. the YZ plane.
- **Reset:** restores original loaded points, re-centering and re-scaling.

To increase efficiency, all rigid transformations (translation, rotation) are accumulated into a 4×4 model matrix. This matrix is applied lazily: transformed coordinates are computed on demand during rendering or before operations that require actual geometry changes (e.g., displacement, normals). In these cases, the transformation is baked into point data. This avoids redundant $O(N)$ updates.

1.3 Performance

Instead of updating every single point immediately, we store transformations in a matrix and apply them only when needed. This reduces the cost of rigid operations from scaling with the number of points ($O(N)$) to a constant-time update ($O(1)$) per frame. Rendering uses on-the-fly transformed coordinates without modifying the original cloud. This leads to markedly smoother interactivity, especially on large point clouds, and reduces CPU load during view manipulation.

2 Part 2: Particle Simulation and Collision Handling

In the second part of the assignment, we developed a real-time simulation of $N = 800$ particles constrained within a two-dimensional square domain. Each particle is defined by its position and velocity, initialized randomly, and updated at each time step using a fixed Δt. Reflective boundary conditions are enforced at the walls: when a particle reaches the boundary of the domain, the corresponding velocity component is inverted, simulating an elastic bounce. This ensures that all particles remain inside the domain while continuously moving.

Collision Detection via Spatial Hashing

Detecting collisions between moving particles is a central requirement. A naive implementation would compare every particle with every other, resulting in $O(N^2)$ distance checks per frame. For $N = 800$, this corresponds to more than 3×10^5 pairwise checks per update, which is impractical for interactive applications.

To address this, we implemented a **spatial hashing** scheme [6, 7]. The domain is divided into uniform square cells with edge length on the order of the particle diameter. Each particle is inserted into the hash table according to its current cell coordinates:

$$\text{cell}(x, y) = \left(\left\lfloor \tfrac{x}{\Delta} \right\rfloor , \ \left\lfloor \tfrac{y}{\Delta} \right\rfloor \right),$$

where Δ is the chosen cell size. Collisions are only tested among particles that reside in the same cell or in one of the eight neighboring cells. Since the number of particles per cell is typically small, each particle only needs to perform a handful of comparisons per update, reducing the average complexity to approximately $O(N)$.

Collision Resolution Strategy

When two particles are found to be within the collision distance (less than twice the radius), a two-step resolution is applied:

1. *Positional correction.* The particles are shifted along the line connecting their centers to eliminate overlap. Each particle is moved half the penetration depth in opposite directions.
2. *Velocity update.* Velocities are swapped between the two particles, which corresponds to an ideal elastic collision between equal masses. To avoid symmetric artifacts and ensure numerical robustness, a small random perturbation is added to each velocity after the swap [8].

This approach guarantees that particles remain separated while exhibiting realistic bounce-like behavior.

Performance Considerations

The efficiency gain from spatial hashing is significant. While a brute-force method grows quadratically with N, the hash grid keeps the number of neighbor checks proportional to the number of particles. With $N = 800$, collision detection and response can be performed in real time at interactive frame rates, whereas the naive method would already be computationally prohibitive. If the number of particles were doubled, the cost would approximately double as well, rather than quadruple. This scalability makes the method suitable for much denser point clouds.

Visualization

The simulation was integrated with a simple OpenGL visualization (using GLFW for window creation and GLEW for extension handling). Particles are rendered as points in a pop-up window, moving smoothly within the bounded square and bouncing against both walls and each other. The visual output confirms that the collision handling prevents overlaps while maintaining continuous motion.

Part 3: Rendering, Surface Inference, and View Synthesis

a) Where does the rendering step take place?

In the `instant-ngp` repository, rendering is performed inside the C++/CUDA *Testbed* application, specifically in the NeRF path (`src/testbed_nerf.cu`). This module implements the ray-marching loop: it launches CUDA kernels that sample along camera rays, query the tiny-cuda-nn MLP for color and density, and accumulate results into per-pixel colors. The project README notes that the application can "train and render a MLP with multiresolution hash input encoding" [9], and the paper explicitly describes visualization using *ray marching* [13].

b) What algorithms do they use to infer the missing surfaces?

The NeRF model encodes only an implicit density field. To obtain explicit geometry, `instant-ngp` applies the classical *Marching Cubes* algorithm to a dense 3D grid of sampled densities (for NeRF→Mesh) or signed distances (for SDF→Mesh). This functionality is exposed in the GUI as the *"NeRF→Mesh / SDF→Mesh"* option. The maintainers confirm that Marching Cubes is the implemented method, although they caution that mesh quality depends on resolution and threshold parameters [9, 10].

c) How is the view synthesis computed?

Novel view synthesis in `instant-ngp` follows the standard NeRF volumetric rendering procedure. For each pixel in a novel camera view, the system:

1. Casts a ray through the scene volume,
2. Samples a sequence of 3D points along the ray,
3. Queries the neural network to predict density σ and emitted color $\mathbf{c}$ at each sample,
4. Integrates these values using alpha compositing:

$$C_{\text{ray}} = \sum_{k=1}^{N} T_k \alpha_k \mathbf{c}_k,$$

where $\alpha_k = 1 - \exp(-\sigma_k \delta_k)$ and T_k is the accumulated transmittance.

To achieve real-time performance, `instant-ngp` introduces two key optimizations:

- A multiscale **occupancy grid** that skips empty space and terminates rays early [13],
- A **fully-fused tiny MLP with multiresolution hash encoding**, implemented with `tiny-cuda-nn`, which reduces per-sample inference cost [13].

These techniques allow novel views to be rendered at interactive frame rates.

2.1 Point Cloud Creation

We reconstructed first the "Poster" example scene using the Nerfstudio framework [14] with the Instant-NGP backend. This scene consists of a textured poster on a flat wall, captured from many viewpoints (226 RGB images in our case) under controlled lighting. Our pipeline proceeds in four stages: preprocessing, NeRF training, geometry extraction, and postprocessing.

The input RGB images were first calibrated using a structure-from-motion (SfM) pipeline. In practice, we used COLMAP to estimate camera intrinsics and extrinsics from the unordered images [11]. The resulting camera poses and **sparse point cloud** were converted into Nerfstudio's data format for NeRF training. No additional filtering or masking was required beyond this standard SfM calibration.

We then trained a neural radiance field (NeRF) [12] using Nerfstudio's Instant-NGP implementation [13, 14]. Instant-NGP augments a compact MLP with a multi-resolution hash table of trainable feature vectors [13] and uses an explicit occupancy grid to skip empty space during ray marching.

After training, we extracted 3D geometry from the learned radiance field. Nerfstudio provides export tools to convert a trained NeRF into a dense point cloud or mesh representation [14]. We used the point-cloud exporter to sample the volume density and generate a **dense set of 3D points** in `.ply` format corresponding to the poster surface. Alternatively, surface reconstruction methods such as Poisson reconstruction can be applied to obtain a mesh.

The raw point cloud often contains artifacts, e.g., noisy points along sampled rays. To clean this, we applied standard point-cloud denoising techniques including statistical outlier removal. This step eliminates most spurious regions and yields a cleaner restricted geometry.

References

[1] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*. Springer, 2008.

[2] R. B. Rusu, S. Cousins, "3D is here: Point Cloud Library (PCL)," *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2011.

[3] R. Goldman, "Rotation matrices and quaternions," in *Graphics Gems*, Elsevier, 2011, pp. 472–475.

[4] M. Botsch, L. Kobbelt, M. Pauly, P. Alliez, B. Lévy, *Polygon Mesh Processing*. AK Peters, 2010.

[5] GLFW: Graphics Library Framework. Available: `https://www.glfw.org/`

[6] M. Teschner, B. Heidelberger, M. Müller, D. Pomeranets, and M. Gross, "Optimised Spatial Hashing for Collision Detection of Deformable Objects," in *Vision, Modeling, and Visualization*, 2003.

[7] N. Greene, "Spatial partitioning for fast ray tracing," in *Proceedings of the SIGGRAPH Symposium on Interactive 3D Graphics*, 1986.

[8] I. Millington and J. Funge, *Artificial Intelligence for Games*, 2nd ed., CRC Press, 2009.

[9] `instant-ngp` GitHub Repository, README. Available: `https://github.com/NVlabs/instant-ngp`

[10] GitHub Discussions, confirmation of Marching Cubes for mesh extraction. Available: `https://github.com/NVlabs/instant-ngp/discussions`

[11] J. L. Schönberger and J.-M. Frahm, "Structure-from-Motion Revisited," in *CVPR*, 2016.

[12] B. Mildenhall, P. P. Srinivasan, M. Tancik, J. T. Barron, R. Ramamoorthi, and R. Ng, "NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis," in *ECCV*, 2020.

[13] T. Müller, A. Evans, C. Schied, and A. Keller, "Instant Neural Graphics Primitives with a Multiresolution Hash Encoding," *ACM Trans. on Graphics (TOG)*, 2022.

[14] M. Tancik *et al.*, "Nerfstudio: A Modular Framework for Neural Radiance Field Development," *ACM SIGGRAPH 2023 Talks*, 2023.