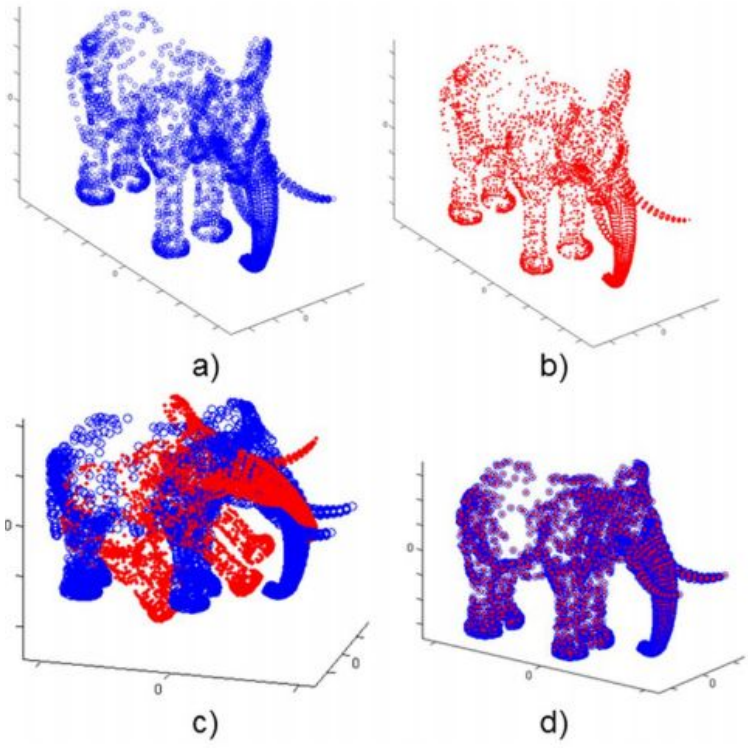
Scan Matching via the Iterative Closest Point Algorithm



An intuitive understanding Scan Matching via Iterative Closest Point is best illustrated in this 11 second video.

Overall Pipeline

More formally, we define our pipeline as follows:

1. **INPUT:** Two 2D/3D point clouds of arbitrary size (*a, b).* For this, we can expect our inputs to be either:
    a. 3D arrays, where each cell represents a point in space. We have ***1*** if there is an item in that x, y, z, and a ***0*** if there isn't. (Not likely, as this will be very memory intensive for sparse pointclouds)
    b. 2D array of glm::vec3's representing the position of each point. This is very similar in spirit to the CUDA Flocking Boids projects (More likely).

2. **ALGORITHM:** We use the *Iterative Closest Point* algorithm. While there are many variants, this algorithm essentially seeks to do the following:

a. For each point in the **target** pointcloud (*a),* find the closest point in the scene pointcloud *(b).*
b. **Compute** a 3D transformation matrix that best aligns the point using Least Squares Regression (this is where most of the CUDA goes, a deep dive on the variations here)
c. Update all points in the target by the transformation matrix
d. Repeat steps 1-3 until some epsilon convergence

3. **OUTPUT:** Some Affine 3D Transformation matrix that defines the Rotation + Translation required to get from *(a) -> (b).* This will be visually illustrated via a gif or pictures of both point clouds iteratively aligning with each other.

CUDA & Project Specs

Baseline

1. **CPU Implementation:** Just like Stream Compaction, offers a good launching point of comparison.

2. **Efficient Data Structure for NN Search:** Naively, a scene with **N** points with a target of **M** points has complexity **O(N*M)** due to the nearest neighbor search. However, unlike the Flocking Boids project, we do not have the benefit of presuming some 'search radius' by which to split our space. In order to overcome this challenge, we must implement some efficient data structure (KD-Tree, Oct-Tree, or some cleverer method) by which to perform a nearest neighbor search.

3. **Matrix Multiplications, Singular Value Decomposition, other Optimization Strategies:** Depending on the specific variant of ICP/Scan Matching we choose to attack, we may have to implement matrix multiplication, SVD, or another optimization strategy. This is required to solve the least squares problem
4. **Final Visualization of Matching Point Clouds**

Extended

5. **More recent implementations of Scan Matching**: ICP is the industry standard, but there are many variants of it. A more recent implementation is linked here. Most implementations of the algorithm are only slightly different from each other, so it would be interesting to see how different methods perform on the GPU.
6. **Running on an Embedded Device:** I initially proposed this project because I wanted to run it using an Intel Realsense point cloud running on a Jetson TX2. While this isn't directly CUDA related, it will still require some non trivial setup, and would be a great exercise in performance analysis.

Libraries and API's

1. For some implementations, a Convex Optimization solver is required. There aren't great open source CVX solvers on the GPU, and Ceres Solver is the best on the CPU. CVX solvers are highly non trivial to write. This does not make life any easier for the CUDA portion of the project though. A CVX solver will simply solve an optimization equations, whereas a lot of the CUDA will go into taking a raw point cloud and turning it into something useful for the CVX solver.
2. Point Cloud Library, ROS, and the f1tenth.com api. This is simply in case we want to run on an embedded device with an actual camera.
3. There are loads of point cloud datasets online. You can even easily create your own eg. try matching a 3D sine curve to a 3D cosine curve.