Programming Assignments 3 and 4 – 600.445/645 Fall 2016

Score Sheet (hand in with report) Also, PLEASE INDICATE WHETHER YOU ARE IN 600.445 or 600.645

Name 1	Doran Walsten
Email	dwalste1@jhu.edu
Other contact information (optional)	
Name 2	Ravi Gaddipati
Email	ravigaddipati@gmail.com
Other contact information (optional)	
Signature (required)	I (we) have followed the rules in completing this assignment Doran Walsten Ravi Gaddipati

Grade Factor		
Program (40)		
Design and overall program structure	20	
Reusability and modularity	10	
Clarity of documentation and programming	10	
Results (20)		
Correctness and completeness	20	
Report (40)		
Description of formulation and algorithmic approach	15	
Overview of program	10	
Discussion of validation approach	5	
Discussion of results	10	
TOTAL	100	

600.645 Computer Integrated Surgery
Programming Assignment 4

Ravindra Gaddipati, Doran Walsten

December 1, 2016

Iterative Closest Point

In the previous assignment, we stopped at the moment where we were able to compute the closest point on a surface using a naive linear search of all triangles in the surface mesh. We also assumed that the registration transformation between the post reference frame and the CT reference frame was simply an identity matrix. In this assignment, our goal was to improve the search for the closest point on a surface by using a k-d tree to represent the surface as well as actually use the iterative closest point (ICP) algorithm to find the best registration transformation between the post reference frame and the CT reference frame.

1.1 Mathematical Approach

1.1.1 Bounding Spheres

Part of the process of generating a k-d tree involves defining bounding spheres around triangles in the surface mesh. This is done in order to represent a region is space that is close to the triangle. This is beneficial during the search algorithm down the road to identify potential triangles in the mesh to project onto. The math below was defined in our lecture slides.

The sphere is defined by first determining the center of the sphere. Given a triangle in 3-D space defined by the vectors $\vec{a}, \vec{b}, \vec{c}$ with edge $(\vec{a}, \vec{b})$ defining the longest edge,

$$\begin{aligned}\vec{f} &= (\vec{a} + \vec{b})/2\\ \vec{u} &= \vec{a} - \vec{f}\\ \vec{v} &= \vec{c} - \vec{f}\\ \vec{d} &= (\vec{u} \times \vec{v}) \times \vec{u}\end{aligned}$$

The center of the sphere lies somewhere along

$$\vec{q} = \vec{f} + \lambda\vec{d}$$

Given $(\lambda\vec{d} - \vec{v})^2 \leq (\lambda\vec{d} - \vec{u})^2$,

$$\lambda \geq \frac{\|\vec{v}\|^2 - \|\vec{u}\|^2}{2\vec{d}(\vec{v} - \vec{u})} = \gamma$$

If $\gamma \leq 0$, pick $\lambda = 0$. Else pick $\lambda = \gamma$.

Once the center is determined, the radius of the sphere is defined as $\|\vec{q} - \vec{a}\|$.

1.1.2 k-d tree search

A standard k-d tree was implemented, where the point space that was searched consisted of the centroids of the bounding spheres of each triangle. A few modifications to the standard search are made to ensure we account for any triangles that may cross the split plane.

1. When checking to see if a new closest point is found, we first check if the distance to the point is closer than the current best minus the bounding sphere diameter. If true, the query point is projected onto the triangle and checked to see if it is a new best.
2. After entering one sub-tree, we need to check if the closest point could be on the other side by intersecting the hypersphere of the best bound, and the split plane. The maximum triangle radius in the entire mesh files is added to the bound to account for possible overlapping triangles. Note we used the same max radius for all levels of the tree to prevent finding a new max for each level. This was done under the assumption that the mesh is fairly uniform in triangle sizes.

1.1.3 ICP

Iterative Closest Point uses matched point correspondences to slowly fit points to a surface. Initially, our registration transformation will be inaccurate and the point cloud will lie somewhere in space away from the actual mesh. However, there exist multiple methods to estimate the best rigid body transformation between point clouds. Each call to this optimization problem brings the transformed points closer to the actual mesh. Because we aren't provided a point cloud for the mesh, we must compute the closest point on the mesh for each point in the transformed point cloud. Repeated iterations of finding these matched pairs then finding the optimal transformation given those points will iteratively bring the clouds closer together until convergence. However, there may exist multiple local minima in the space of this optimization problem. This means that the initial guess for the registration transformation matters. Otherwise, the system may converge to a non-optimal solution.

1.2 Algorithmic Approach

1.2.1 k-d tree

The previous naive closest point search was converted to one using a k-d tree. The tree is declared in `surface.h`, where `Surface::Division` represents a node in the tree. A k-d is a form of a binary search tree, where each level splits the input points based on a single axis, determined by the tree depth. The main algorithmic decision for the k-d tree was median selection; we chose to use `std::nth_element()`. This STL algorithm implements introselect, a hybrid of quickselect and median of medians. This algorithm partially sorts a range such that the nth element is as such if the entire range was sorted. Every element before the nth element is gaurenteed to be less than every element after the nth element. This alogrithm is ideal for this application since we're not interested in a fully sorted range, but partitioning in left and right. This gives us O(n) performance on average when constructing the tree, and O(log n) performance when finding the closest point.

The actual points stored in the k-d tree were the centroids of each triangles' bounding sphere. As described in the mathematical approach, the radius of the bounding sphere was used to determine if a triangle could possibly have a closer point than the current bound. This has the advantage of avoiding a full projection on each triangle.

1.2.2 ICP

We followed the practical ICP method described in the lecture notes. The following are the parameters used during the algorithm

1. n - Iteration number

2. $F_{reg,n}$ - The current estimate of the registration transformation on iteration n. η_n - The current estimate of the match distance threshold. (Any closest point which is greater than this distance away from the original point is ignored to reduce effect on transformation update

3. $E = ..., \vec{e}_k, ...$ - Error of existing registration transformation for each valid matched point pair. Used to determine the terminating condition for the algorithm

This approach consists of 5 steps.

1. **Initialization**
   During this step, we load in all of the relevant data including the original points in the rigid body frame as well as the surface mesh. We also initialize the threshold value η.

2. **Matching**
   Compute the closest point on the surface mesh $\vec{c}_k$ for each point in the rigid body frame $\vec{q}_k$ using the current value of $F_{reg,n}$ and compute the distance $\vec{d}_k = \|\vec{c}_k - F_{reg,n-1} * \vec{q}_k\|$. If $\vec{d}_k < \eta_n$, store the matched pair of points $(\vec{q}_k, \vec{c}_k)$.

3. **Transformation update**
   Compute a rigid body transformation between the point clouds of matched points to generate a new estimate for $F_{reg,n}$. Using this estimate, compute the residual error e_k. Using e_k, determine the mean magnitude of the error $\bar{\epsilon}_n$.

4. **Adjustment** Update the value of $\eta = 8\bar{\epsilon}_n$.

5. **Iterate** We will use the terminating condition that $0.96 \leq \frac{\bar{\epsilon}_n}{\bar{\epsilon}_{n-4}} \leq 1$. If this is not satisfied, the iterations will continue between steps 2 through 4. We experimented with this threshold, and found it to preform the most consistently.

1.3 Software Overview

The provided source files included are:

- `main.cpp`: Main driver, also includes a function to compare debug files to outputs.
- `files.h`: File parsers for the input data files.
- `surface.h`: Represents a surface mesh as a k-d tree, where each node is a `cis::Surface::Division`.
- `horn.h`: From a previous assignment, performs cloud to cloud registration.
- `icp.h`: Implements the iterative closest point algorithm.
- `pointcloud.h`: From a previous assignment, defines a class to represent a cloud of points.
- `registration.h`: Transforms the pointer body to the fixed reference frame.
- `utils.h`: Common functions such as string splitting.

1.4 Verification

1.4.1 Closest Point using k-d tree

To verify k-d tree construction, we checked that all of the points were preserved. We wanted to verify that our search of the k-d tree for the closest point on the mesh was functioning properly. To do so, we created a prism defined with the vertices

$$\{(0,0,0),(1,0,0),(1,1,0),(0,1,0),(0,0,2),(1,0,2),(1,1,2),(0,1,2)\}$$

and two triangles defining each face. This test also allows us to test the case of overlapping bounding spheres. This geometry was chosen because the simple faces had predictable projections given random points around the object. We considered the motif of two faces coming together with one edge to create 3 test cases: project onto face 1, project onto the edge, project onto face 2. These regions in space are defined by

$$x \geq 1, 1 \geq y \geq 0, 2 \geq z \geq 0$$

$$1 \geq x \geq 0, y \geq 1, 2 \geq z \geq 0$$

$$x \geq 1, y \geq 1, 2 \geq z \geq 0$$

We generated 5 random points in each of these region and ensured that the projections were in the form of

$$(1,y,z),(x,1,z),(1,1,z)$$

. we observed that these test cases passed, and in addition to comparison with the naive search results, are confident that we developed a correct search of the k-d tree/closest point determination.

After construction, we also verified that every point was equal to or less than the split point of its' parent node, and that the total number of points in the tree remained constant upon splitting into child nodes.

1.5 Results

We compared the speed of the programs on the debug files, using the naive search and the k-d tree. Five trials for each file was done. The results are as follows. The figures are conservative, given that the k-d tree runtime is small enough that overhead could be significant.

```
Average time in seconds.
Input   naive     k-d tree   speedup
------------------------------------
A       0.98      0.038      25.79
B       2.93      0.058      50.52
C       3.536     0.052      68
D       2.722     0.04       68.05
E       1.596     0.03       53.2
F       1.482     0.026      57
------------------------------------
```

The error for the provided debug files is as follows:

```
Comparing "OUTPUT4/PA4-A-Output.txt" and "INPUT4/PA4-A-Debug-Answer.txt"
s_k average error: 0.00630512
c_k average error: 0.00515087
```

```
Difference average error: 0.00204

Comparing "OUTPUT4/PA4-B-Output.txt" and "INPUT4/PA4-B-Debug-Answer.txt"
s_k average error: 0.00632635
c_k average error: 0.00417279
Difference average error: 0.00198

Comparing "OUTPUT4/PA4-C-Output.txt" and "INPUT4/PA4-C-Debug-Answer.txt"
s_k average error: 0.00536797
c_k average error: 0.00390102
Difference average error: 0.002045

Comparing "OUTPUT4/PA4-D-Output.txt" and "INPUT4/PA4-D-Debug-Answer.txt"
s_k average error: 0.00841151
c_k average error: 0.0070447
Difference average error: 0.00345

Comparing "OUTPUT4/PA4-E-Output.txt" and "INPUT4/PA4-E-Debug-Answer.txt"
s_k average error: 0.0300523
c_k average error: 0.025546
Difference average error: 0.011925

Comparing "OUTPUT4/PA4-F-Output.txt" and "INPUT4/PA4-F-Debug-Answer.txt"
s_k average error: 0.0232257
c_k average error: 0.0185215
Difference average error: 0.010455
```

The output files for the unknown data are included. In general, we found that the convergence was very quick. Waiting for the error to stabilize between two spaced intervals (here spaced by 5) had us execute some extra iterations but helped us be more confident in the convergence.

1.6 Discussion

When either the naive linear or the k-d tree search method is used, our ICP method works very well. The error between our results and the provided debug data is minimal as is the error of the registration itself for all of the debug trials. However, the runtime was heavily reduced when the k-d tree was used. Speedup was over 50x for most of the files. This suggests that for larger and larger meshes, using a k-d tree is a better alternative which reduces time without losing accuracy since search time is reduced to O(log n).

Early iterations of our k-d tree data structure passed our verification test but would fail to converge to a satisfactory solution during ICP. We determined that the k-d tree was not determining the same closest point on the mesh as the naive linear search for a collection of points in the data. This was frustrating as we could not picture a verification test where we could increase the complexity of the surface and still estimate the expected projection in order to find where the algorithm was going wrong. We determined that the choice of splitting plane (X,Y,Z) was off by a level within the tree. Because the prism was a symmetric object, the consequences of this issue could not be observed. In future endeavors, we may need to come up with a better approach to verification which still uses a simple object but is not symmetric in order to catch issues such as this.

We played with the initial transformation guess, noticing it had a large impact. While this was less of an issue once we fixed the above bugs, we used an initial transformation of a translation to the surface centroid as the seed.

1.7 Usage

The codebase is built with cmake. Inside the project directory,

```
mkdir build && cd build
cmake -DCMAKE_BUILD_TYPE=Release .. && make -j4
```

The executable is also included.

```
e.x:

bin/cisicp INPUT/Problem4MeshFile.sur INPUT/Problem4-BodyA.txt
    INPUT/Problem4-BodyB.txt INPUT/PA4-G-Debug-SampleReadingsTest.txt
    OUTPUT/PA4-G-Output.txt

bin/cisicp test
```

Two executables are built, one which uses the naive closest point search (`cisicp_naive`) and the other using the k-d tree (`cisicp`). The CLI usage information is printed if no arguments are passed.

This program uses Eigen (eigen.tuxfamily.org) for linear algebra and numerical methods as well as doctest (github.com/onqtam/doctest) to implement test cases.

1.8 Contributions

Ravi primarily worked on the k-d tree, and Doran mainly on the ICP algorithm. Both contributed to developing the methods and testing.