ED5310

Feature Matching - Delaunay Triangulation

Fasith Ahamed F (ED20B019)
Sabare Greesh M (ME20B152)

Problem statement and a discussion on its relevance to the course

Feature matching of images using Delaunay Triangulation
Feature detection and matching is an important task in many computer vision applications, such as SLAM, structure-from-motion and more. Feature points are expressive in texture that can describe an image. The algorithm initiates by detecting feature points from image data using the FAST (Features from Accelerated Segment Test) approach. Subsequently, it performs Delaunay triangulation on these points, leveraging its uniqueness property to facilitate matching between any two sets of feature points. The similarity between 2 feature points of different sets is established using the nearest neighbours of the points.

Course Relevance: Delaunay Triangulation, 3D Convex Hull, Algorithms

State of the art relevant work details

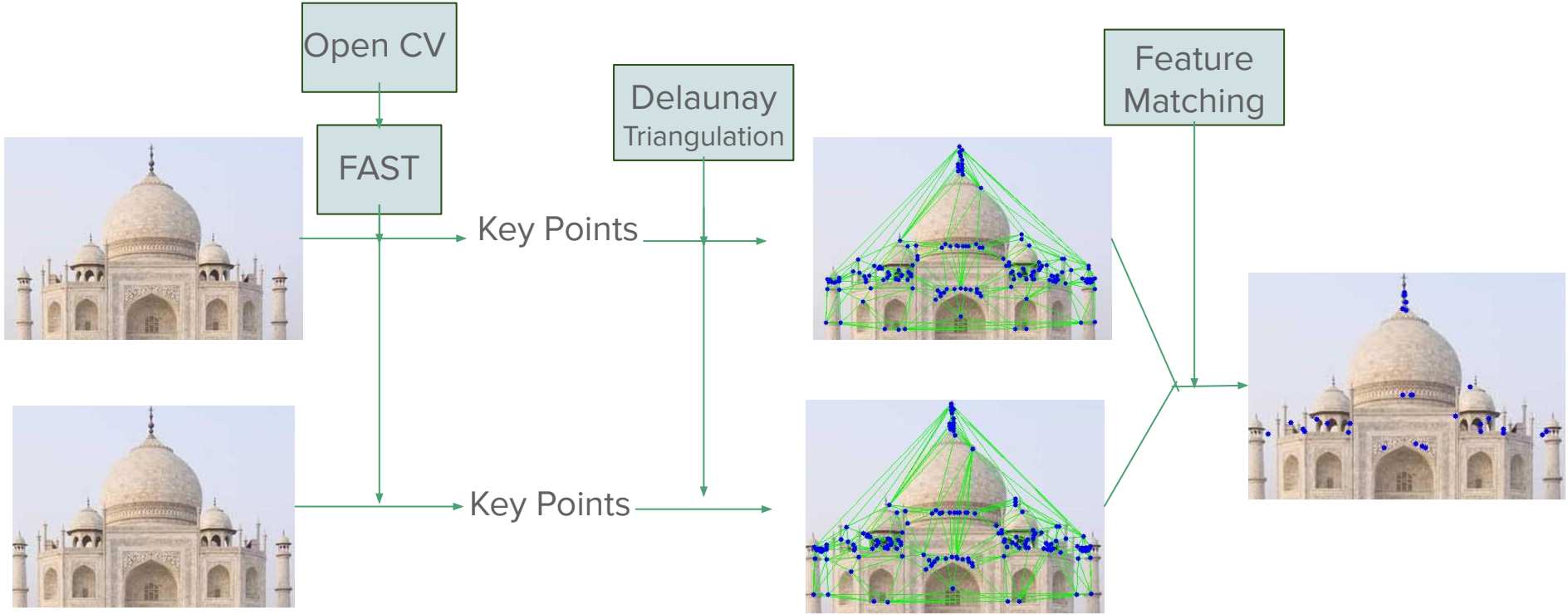
Proposed new feature matching technique

New algorithm for AR/VR SLAM feature detection

Challenge Addressed in work

Current feature detection algorithms such as FAST and SURF often yield an overwhelming number of features in an image, posing challenges for SLAM algorithms that utilize these features to map a 3D environment. Our objective is to refine this pool of features by isolating those that exhibit invariance to affine transformations—specifically, scaling, zooming, rotation, and similar operations. Delaunay Triangulation plays a pivotal role in achieving this goal due to its unique property of preserving affine transformation invariance.

Proposed methodology



Fast Detector

```python
import cv2

img = cv2.imread('Feature_Matching_DT\Fast_Detection\input2.jpeg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
fast = cv2.FastFeatureDetector_create(40)

kp = fast.detect(gray,None)
img2 = cv2.drawKeypoints(img, kp, None)
height, width = img.shape[:2]
scale_factor_x = 10 / width
scale_factor_y = 10 / height

for point in kp:
    point.pt = (point.pt[0] * scale_factor_x, point.pt[1] * scale_factor_y)

print("Threshold: ", fast.getThreshold())
print("nonmaxSuppression: ", fast.getNonmaxSuppression())
print("neighborhood: ", fast.getType())
print("Total Keypoints with nonmaxSuppression: ", kp[0].pt)

with open('Feature_Matching_DT\keypoints2.txt', 'w') as file:
    for point in kp:
        x, y = point.pt
        file.write(f'{x} {y}\n')
# display the image with keypoints drawn on it
cv2.imshow("Keypoints with nonmaxSuppression", img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Feature Matching Main()

```
int main()
{
  vector<Vertex_3D> tests_a = readKeypointsFromFile("keypoints.txt");
  vector<Vertex_3D> tests_b = readKeypointsFromFile("keypoints2.txt");
  Delaunay C_a(tests_a);
  unordered_map<Vertex_3D, vector<Vertex_3D>> neighbor_map_a = C_a.Map_Neigh();
  C_a.WriteVerticesToFile("output_vertices_edges_a.txt");
  Delaunay C_b(tests_b);
  unordered_map<Vertex_3D, vector<Vertex_3D>> neighbor_map_b = C_b.Map_Neigh();
  C_b.WriteVerticesToFile("output_vertices_edges_b.txt");
  Feature_Match Fm(neighbor_map_a, neighbor_map_b);
  unordered_map<Vertex_3D, vector<Vertex_3D>> feature_map = Fm.check();

  vector<Vertex_3D> features;
  for (const auto& pt : feature_map){
    features.push_back(pt.first);
  }
  WriteVerticesToFile("final_features.txt",features);
 return 0;
}
```

Read and Write to file function

```
vector<Vertex_3D> readKeypointsFromFile(const string& filename) {
    ifstream file(filename);
    vector<Vertex_3D> vertices;

    if (file.is_open()) {
        float x, y;
        while (file >> x && file.ignore() && file >> y && file.ignore()) {
            float z = x * x + y * y;  // Calculate z coordinate
            Vertex_3D vertex(x, y, z);
            vertices.push_back(vertex);
        }
        file.close();
    } else {
        cerr << "Unable to open file!" << endl;
    }

    return vertices;
}
void WriteVerticesToFile(const std::string& filename, vector<Vertex_3D> vertices) {
        std::ofstream outfile(filename);

        if (!outfile.is_open()) {
            std::cerr << "Unable to open file for writing: " << filename << std::endl;
            return;
        }

        // Write vertices to the file
        for (const auto& vertex : vertices) {
            outfile << "Vertex: (" << vertex.x << ", " << vertex.y << ", " << vertex.z << ")\n";
        }

        outfile.close();
        std::cout << "Vertices and edges written to file: " << filename << std::endl;
    }
```

Delaunay Triangulation

```
class Delaunay
{
  public:
    Delaunay(const vector<Vertex_3D>& vertices_)
    {
      this->vertices = vertices_;
      this->Generate_PLY();
      this->Create_Hull();
      this->Generate_STL();
      //this->map = this->Map_Neigh();
    }
    unordered_map <Vertex_3D, vector<Vertex_3D>> Map_Neigh()
    {
      unordered_map <Vertex_3D, vector<Vertex_3D>> neigh_map;
      for (auto &vx : this->vertices)
      {
          for (auto &edg : this->edges)
              if (edg.end_vertices[0] == vx || edg.end_vertices
                  neigh_map[vx].push_back(edg.end_vertices[0] =
      }
      return neigh_map;
    }
```

```
vector<Vertex_3D> vertices = {};
vector<Vertex_3D> Hull_vertices = {};
list<Face> faces = {};
list<Edge> edges = {};
//unordered_map <Vertex_3D, vector<Vertex_3D>> map;
unordered_map<size_t, Edge*> edge_map;
```

```
void Create_Hull()
{
  vector<Vertex_3D>& vertices = this->vertices;
  if(this->Start_Hull(vertices)){
    for(const auto& vx : vertices)
    {
      if(!vx.looped)
      {
      this->Increment_Hull(vx);
      this->Clean_Up();
      }
    }
  this->Hull_Vertices();
 }
  else return;
}
```

```
void Increment_Hull(const Vertex_3D& vx)
{
  if(!Visibility_Check(vx)) return;
  for(auto it = this->edges.begin(); it != this->edges.end(); it++)
  {
    auto& edge = *it;
    auto& face1 = edge.face_linked1;
    auto& face2 = edge.face_linked2;
    if(face1 == NULL || face2 == NULL)
    {
      continue;
    }
    else if(face1->visible && face2->visible)
    {
      edge.remove = true;
    }
    else if(face1->visible|| face2->visible)
    {
      if(face1->visible) std::swap(face1, face2);
      auto inner_vx = this->Get_Inner_Vertex(face2, edge);
      edge.Erase(face2);
      this->Create_Face(edge.end_vertices[0], edge.end_vertices[1], vx, inner_vx);
    }
  }
}
```

Structures

```
struct Vertex_3D {
  float x,y,z;
  bool looped;
  Vertex_3D() = default;
  Vertex_3D(float _x, float _y, float _z):\
      x(_x), y(_y), z(_z), looped(false) {}

  bool operator !=(const Vertex_3D& vx) const
  {
      return x != vx.x || y != vx.y || z != vx.z;
  }
  bool operator ==(const Vertex_3D& vx) const
  {
      return x == vx.x && y == vx.y && z == vx.z;
  }
  bool operator <(const Vertex_3D& vx) const
  {
      return x < vx.x && y < vx.y && z < vx.z;
  }
  double dist(Vertex_3D& vx){

          return sqrt(((vx.x-x)*(vx.x-x)) + ((vx.y-y)*(vx.y-y)));
  }
      size_t operator()(const Vertex_3D& v) const {
        // Create a hash combining the individual hashes of x, y, and z
        size_t seed = 0;
        hash_combine(seed, std::hash<float>{}(v.x));
        hash_combine(seed, std::hash<float>{}(v.y));
        hash_combine(seed, std::hash<float>{}(v.z));
        return seed;
    }
```

```
struct Face
{
  bool visible;
  Vertex_3D vertices[3];
  Face(const Vertex_3D& v1, const Vertex_3D& v2, const Vertex_3D& v3): visible(false)
      { vertices[0] = v1; vertices[1] = v2; vertices[2] = v3;};

  void Reverse(){swap(vertices[0], vertices[2]); };
};

struct Edge
{
  bool remove;
  Face *face_linked1, *face_linked2;
  Vertex_3D end_vertices[2];
  Edge(const Vertex_3D& v1, const Vertex_3D& v2):
      face_linked1(nullptr), face_linked2(nullptr), remove(false)
      { end_vertices[0] = v1; end_vertices[1] = v2; };

  void Face_linked(Face* face)
  {
    if( face_linked1 == NULL || face_linked2 == NULL )
      (face_linked1 == NULL ? face_linked1 : face_linked2)= face;
  };

  void Erase(Face* face)
  {
    if(face_linked1 == face || face_linked2 == face)
      (face_linked1 == face ? face_linked1 : face_linked2) = nullptr;
    else return;
  };
```

```
struct Vertex_Hash
{
  size_t operator() (const Vertex_3D& v) const
  {
      return hash<string>{}(to_string(v.x)+to_string(v.y)+to_string(v.z));
  }
};
```

Feature Match

```
class Feature_Match{
  public:
    Feature_Match(unordered_map<Vertex_3D, vector<Vertex_3D>> a, unordered_map<Vertex_3D, vector<Vertex_3D>> b)
    {
      this->map_a = a;
      this->map_b = b;
    }
    unordered_map<Vertex_3D, vector<Vertex_3D>> check(){
          unordered_map<Vertex_3D, vector<Vertex_3D>> fp = this->map_a;
          for(auto pt: this->map_a){
            bool ans;
            for(auto ptb: this->map_b){


              vector<float>ap, bp;
              Vertex_3D point = pt.first;
              Vertex_3D pointb = ptb.first;
              ap = this->dist_pt(1, point);
              bp = this->dist_pt(2, pointb);

              ans = this->is_equal(ap, bp);

            }
            if (ans == 0){
                fp.erase(pt.first);
              }
          }

          return fp;
        }
```

```
unordered_map<Vertex_3D, vector<Vertex_3D>> map_a;
unordered_map<Vertex_3D, vector<Vertex_3D>> map_b;

bool is_equal(vector<float> ap,vector<float> bp){

    sort(ap.begin(), ap.end());
    sort(bp.begin(), bp.end());

    if (ap.size() != bp.size()){
      return false;
    }
    int sz = bp.size();
    int threshold = 1;
    int max_thres = 0.8 * sz;

    int ans = 0;
    for(int i = 0;i<sz;i++){

        ans+=(abs(ap[i]-bp[i])<threshold);
    }


    return ans>=max_thres ? 1 : 0;
}
```

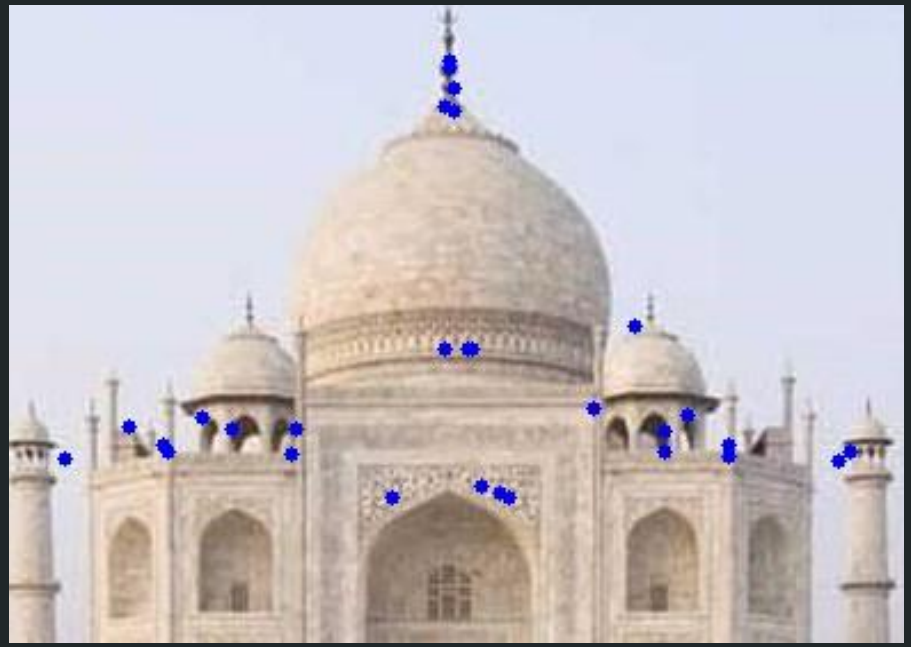
```
vector<float> dist_pt(const int num, Vertex_3D& pt){
    vector<float> ans;
    if(num==1){
        for(auto p: this->map_a[pt]){

            ans.push_back(p.dist(pt));
        }
    }
    else{

            for(auto p: this->map_b[pt]){

            ans.push_back(p.dist(pt));
        }
    }
    return ans;
}
```

Display

```python
import cv2
import shutil


# Read the output file cinng vertices and edges
with open("Feature_Matching_DT\Delaunay_Triangulation\output_vertices_edges_b.txt", "r") as file:
    lines = file.readlines()

base_image = cv2.imread("Feature_Matching_DT\Fast_Detection\input2.jpeg")  # Replace with your existing image
height, width = base_image.shape[:2]
# Draw edges on the existing image
# Parse vertices and edges data
vertices = []
edges = []
for line in lines:
    if line.startswith("Vertex"):
        vertex_str = line.split("(")[1].split(")")[0]
        x, y, z = map(float, vertex_str.split(","))
        vertices.append((int(x*width/10), int(y*height/10)))  # Assuming 2D po fr image display
    elif line.startswith("Edge"):
        start_index = line.index("[")
        end_index = line.index("]")
```

```python
# Draw edges on the copied image
for edge in edges:
    cv2.line(copied_image, edge[0], edge[1], (0, 255, 0), 1)  # Green lines, adjust thickness if needed
for vertex in vertices:
    cv2.circle(copied_image, vertex, 3, (255, 0, 0), -1)
# Save the copied image with the triangulation
shutil.copyfile("Feature_Matching_DT\Fast_Detection\input2.jpeg", "Feature_Matching_DT\Fast_Detection\output2.jpeg")  # Make a copy of the original image
cv2.imwrite("Feature_Matching_DT\Fast_Detection\output2.jpeg", copied_image)  # Save the modified image


# Display the copied image with the triangulation
cv2.imshow("Delaunay Triangulation", copied_image)

with open(r"Feature_Matching_DT\Delaunay_Triangulation\final_features.txt", "r") as file2:  # Replace with your second vertices file
    lines2 = file2.readlines()

# Parse the second set of vertices data
vertices2 = []
for line in lines2:
    if line.startswith("Vertex"):
        vertex_str = line.split("(")[1].split(")")[0]
        x, y, z = map(float, vertex_str.split(","))
        vertices2.append((int(x*width/10), int(y*height/10)))  # Assuming 2D points for image display

# Create another copy of the image for drawing the second set of vertices
copied_image2 = base_image.copy()

# Draw vertices from the second set on the second copied image
for vertex in vertices2:
    cv2.circle(copied_image2, vertex, 3, (255, 0, 0), -1)  # Blue circles for second set of vertices
shutil.copyfile("Feature_Matching_DT\Fast_Detection\input1.jpeg", r"Feature_Matching_DT\Fast_Detection\features.jpeg")  # Make a copy of the original image
cv2.imwrite(r"Feature_Matching_DT\Fast_Detection\features.jpeg", copied_image2)

cv2.imshow("Features final", copied_image2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Results



Tools Used

- OpenCV - Fast Detector
- OpenCV - displaying image

References

1. https://arxiv.org/abs/0810.2434
2. https://www.mdpi.com/1424-8220/23/1/146
3. https://medium.com/@deepanshut041/introduction-to-feature-detection-and-matching-65e27179885d
4. https://www.sciencedirect.com/science/article/pii/S0031320398000867
5. https://www.sciencedirect.com/science/article/pii/S0030402613009030?via%3Dihub

Team Members and contributions

Fasith Ahamed F :
(ED20B019)

Sabare Greesh :
(ME20B152)