AutoVision X Hackathon

SVS – Surround View System

CHALLENGE: Image Stitching Application Using C++

Team 1Neuron – Akash Rai, Ankur

- 11th February 2024

Repo Link: <https://github.com/akashrai2003/ImageStitchingCPP-BoschProject/tree/main>

Abstract:

The Surround View System is a C++-based application that allows for the seamless stitching of an arbitrary number of photos captured from various devices. C++ was chosen as the major programming language because of its better execution speed, which is especially important for real-time applications like autonomous vehicles, where quick inference times are required to avoid negative repercussions. This application takes advantage of CPUs' parallel processing capabilities and, when available, GPUs. Notably, the system accepts both sequential and non-sequential image inputs and prioritizes matches within the images. Furthermore, the application is designed to work easily on x64 platforms using the MinGW compiler, assuring compatibility without requiring additional dependencies from end users like building OpenCV and related dependencies only by using the executable file. Let’s dive into it….

Algorithm Explanation:

Our technique is based on fundamental picture stitching concepts, including key components such as feature detectors, feature descriptors, feature matching, and the generation of graphs from matched features. A crucial component of our method is the application of homography-based estimators to extract accurate camera parameters, which enable efficient warping and the creation of high-quality stitched images. Recognizing the common obstacle of fluctuating exposures in photographs recorded under sunshine, our system handles this issue during the final phase with expert image mixing. This painstaking technique improves image quality by removing exposure-related anomalies, resulting in a refined and visually pleasing composite image. Additionally, we have incorporated an image inpainting feature to address the occurrence of black patches at the edges resulting from the warping of images captured from various angles. This optional functionality has been implemented to eliminate undesired artefacts, although it may not be suitable for all users as it gives rise to blurred edges of the image.

*Steps To Reproduce Our Results:*  To be able to obtain similar results as obtained by us you can follow these steps:

* Getting the OpenCV and opencv-contrib package(for other dependencies not in the basic zip file) from the official GitHub repository of OpenCV.
* Extracting these files and then we can either edit the CMakeLists.txt if on Debian based environments or use CMakeGUI on Windows for easy installation of files.
* Selecting the source file as the OpenCV file and the destination as an empty folder where we will build our libraries.
* After clicking configure we’ll be selecting our Visual Studio version and following the architecture for which we prefer to build the packages i.e. x64 in our case.
* Then searching for OPENCV\_EXTRA\_MODULES\_PATH and adding the modules directory of the opencv-contrib package.
* Also making sure that BUILD\_WITH\_STATIC\_CRT=OFF, if we want to statically link our libraries and also BUILD\_SHARED\_LIBS=OFF while selecting OPENCV\_ENABLE\_NONFREE=ON .
* Then clicking configure -> Generate -> Open Folder
* Further on selecting the Debug or Release version and then under the CMakeTargets directory we will first build the file named as ALL\_BUILD and after that we’ll go for building INSTALL.
* Creating a new C++ project and then adding a new cpp file to your source files.
* Going into the Project Properties -> VC++ -> Include Directories -> Adding include path from install folder and also Library Directories->.\install\x64\vc17\staticlib .
* Then going under C/C++->Code Generation->Runtime Library->Multi Threaded /MT(for static linking).
* Linker->Input->Additional Dependencies-> Adding all the filenames inside staticlib with .lib extensions.

Now you’re good to go and use any library required from the OpenCV package without having to care about also transferring DLL files with the program.

Basic Ideation Behind The Scenes:

The idea behind the whole application has being explained below by elaborating on the topics:

1. *Ensuring Enough number of images:*
   * For the application to run we should have atleast 2 images for stitching and we can usually set the threshold manually if we want to tweak in some parameters on our own.
   * Another good functionality of the application is that if there are completely different images present in the inputs given then it will just ask for input images and not give out the results.
2. *Feature Detection on the Input images:*

* Now we perform one of the most important aspects of our code i.e. feature detection, it grabs out all the features inside the image based on different techniques like SIFT(Scale-Invariant Feature Transform), SURF(Speeded-Up Robust Features), ORB(Oriented FAST and Rotated BRIEF), AKAZE(Accelerated-KAZE).
* These features serve as reference points for subsequent processing steps like matching and alignmen

*Fig.* SURF Detection

1. *Image Matching and Alignment:*

* Following feature detection, the application matches corresponding features across images to align them accurately.
* This step involves estimating homography-based transformations to account for camera motion and perspective changes between images. By aligning features, the application ensures seamless blending and stitching of the images.

*A collage of a garden

Description automatically generated*

All the Features Detected by ORB detector

1. *Pairwise Matching:*

* The program performs pairwise matching of features between images, comparing detected keypoints and descriptors to establish correspondences.
* The matcher type is chosen based on the matcher\_type variable, which could be Affine, BestOf2Nearest, or BestOf2NearestRange.
* After matching, only images that contribute to the largest connected component are retained, ensuring coherent stitching and alignment*.*
* *Why not GoodFeatures:*

1. Good Features + Geometric Transformation:

* Pros:

Distinctive Points: Focuses on identifying distinctive and repeatable points across images.

Explicit Geometric Information: Provides explicit spatial relationship information through geometric transformation (e.g., homography).

Robustness: Handles significant viewpoint changes, rotations, and scaling.

* Cons:

Limited to Detected Features: Relies on the quality and distribution of detected keypoints.

Manual Parameter Tuning: Performance can be sensitive to feature detector and matching algorithm parameters.

1. Pairwise Matching:
   * + Pros:

Flexible Feature Types: Works with various feature types (e.g., ORB, AKAZE, SIFT) and descriptors.

Global Consistency: Considers information from multiple images simultaneously, potentially leading to more global consistency.

Works with Any Point: Not limited to specific features; any distinctive point can be used for matching.

* + - Cons:

Computational Complexity: Can be computationally expensive, especially for a large number of images. Sensitivity to Outliers: Outliers or incorrect matches may impact overall alignment.

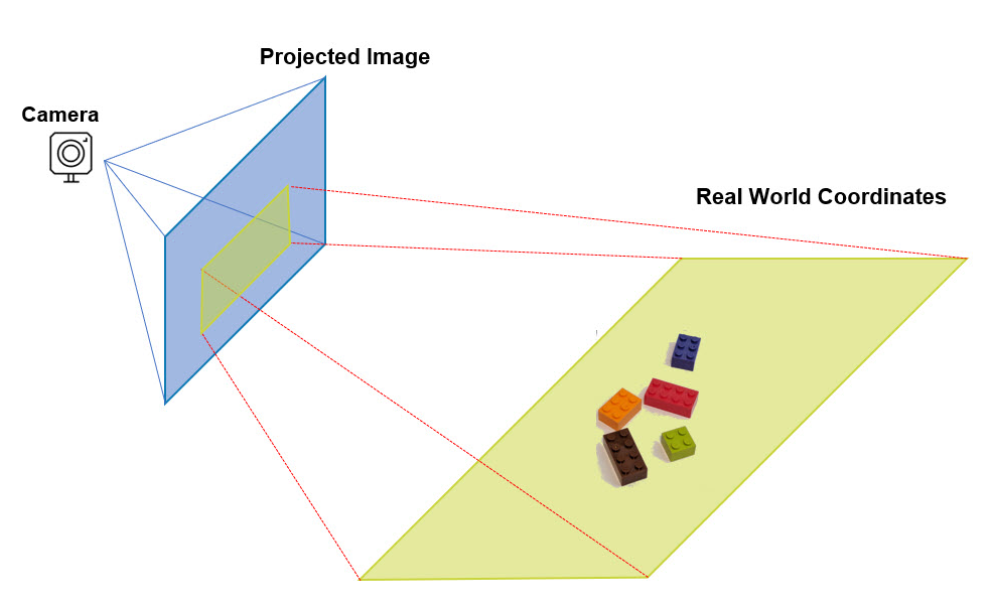
*A collage of a garden

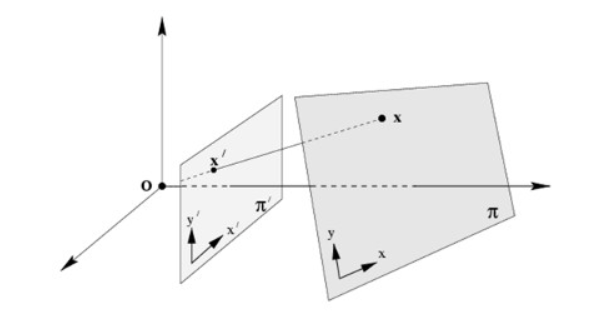
Description automatically generated*

*Fig.* All the Pairwise matching outputs

1. *Camera Estimation:*

* Camera Estimation involves determining intrinsic and extrinsic parameters using mathematical models. The choice between Affine and Homography estimators affects complexity and accuracy:
* Affine Estimator: Assumes linear mapping with minimal distortion, suitable for simpler camera geometries.
* Homography Estimator: Represents general mapping with non-linear distortions, ideal for complex scenes and camera geometries.





* + *WHY NOT RANSAC?*
    - Advantages of RANSAC:

Simplicity in implementation. Robustness to outliers in feature matching. Can handle non-linear transformations.

Advantages of Camera Estimation and Bundle Adjustment:

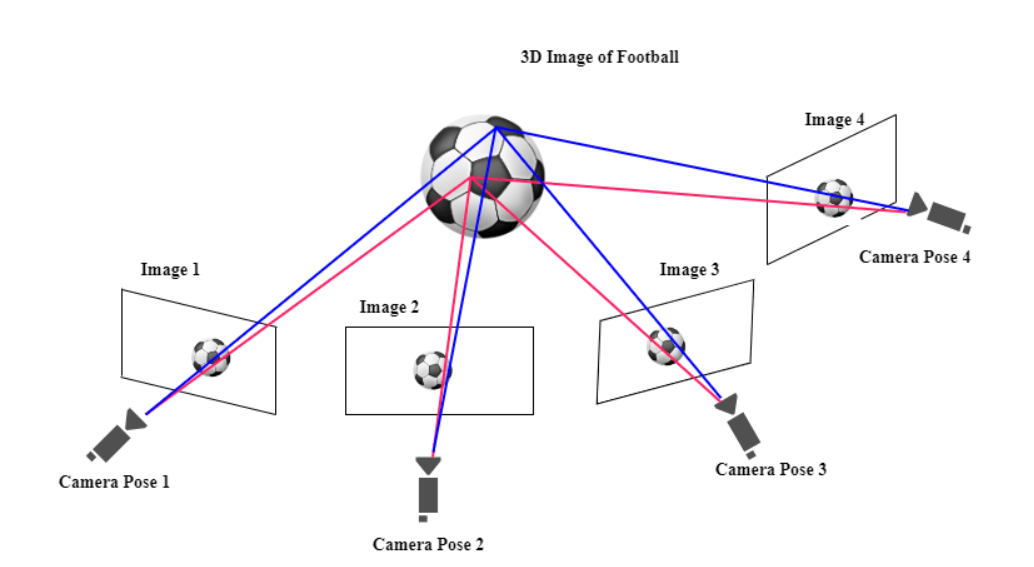
* + - Global optimization leads to better overall results. Consideration of inter-image constraints improves accuracy. Can handle non-linear distortions and lens effects.
    - Considerations:

RANSAC is suitable for simpler cases and may be faster.

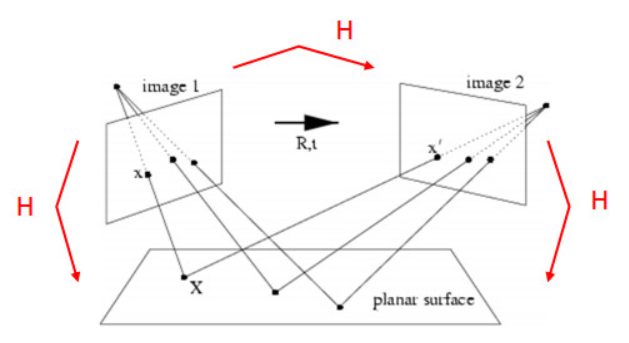
Camera estimation and bundle adjustment provide more accurate and globally optimized results but might be computationally more intensive.

1. *Bundle Adjustment:*

* Camera parameters are further refined using bundle adjustment, which optimizes parameters to minimize reprojection errors.
* The bundle adjustment cost function is selected based on the ba\_cost\_func variable, influencing the optimization process.



1. *Warp Images:*

* Images are warped based on the estimated camera parameters using the chosen warping method (e.g., Plane, Affine, Cylindrical, or Spherical).
* This step ensures that images are properly aligned and stitched together to form a cohesive panorama.

1. *Exposure Compensation and Inpainting:*

* To address exposure variations and artifacts resulting from image warping, the application employs exposure compensation techniques during the final phase.
* Additionally, an optional image inpainting feature is incorporated to eliminate black patches at image edges caused by perspective distortion. While this may introduce slight blurriness, it enhances the overall visual coherence of the stitched image.

**

1. *Seam Finding:*

* Seams between adjacent images are identified to facilitate seamless blending.
* The seam finder type is chosen based on the seam\_find\_type variable, allowing for customization of the seam finding process.



1. *Image Blending:*

* Images are blended together using the selected blending type (e.g., linear blending, multi-band blending) to create a visually appealing final panorama.
* Optionally, intermediate results can be saved if timelapse mode is enabled, allowing for further analysis or visualization of the stitching process. 

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Advantages Over Other Applications:  Some of the unique things we’ve tried in our application are described below:   * *Parallel Computing:* It ensures that our application works faster on CPU’s too and we’d done it by 2 major methods where:  1. Using OpenMP: As suggested by OpenCV it is one of the most user friendly and also can increase the speed of execution as compared to normal executions. As OpenCV is highly optimized in itself thus gaining a few seconds is good enough.   #pragma omp parallel for schedule(dynamic) ordered  for (int img\_idx = 0; img\_idx < num\_images; ++img\_idx)  {  #pragma omp ordered  {……}  }   1. Multi-Threading: We’d manually used threading to finally compost our images into one and thus enhancing the speed of execution faster as compared to OpenMP too.   for (int img\_idx = 0; img\_idx < num\_images; ++img\_idx){  {  std::lock\_guard<std::mutex> lock(orderMutex);  }  }  for (auto& thread : threads){  thread.join();  }   |  |  |  | | --- | --- | --- | | Parallel Processing using OpenMP: | Without OpenMP | Using threading | | Compositing, time: 2.33728 sec  Finished, total time: 4.16571 sec | Compositing, time: 2.43665 sec  Finished, total time: 4.38755 sec | Compositing, time: 2.2924 sec  Finished, total time: 4.15271 sec |   *Fig.* Table Comparing the Parallel Processing Speeds of the Application   * *Memory Optimization:* The code frees all the allocated memory accordingly when it’s not needed any further and thus memory usage is lower than usual. The implementation was made much more comfortable by OpenCV itself. * A pond in a garden    Description automatically generated*Inpainting of the Images:* The images when stitched are usually warped if they are taken from different angles thus it leaves a black patch at the edges which can sometimes be not very suitable. So to fill those spaces blending wasn’t a good idea with its neighboring pixels as it gave a dark demeanor. So we used image inpainting which could actually produce blurry edges and would just look like the interpolation of the image.       *Fig.* Comparing the 3 different types of output that can be obtained (a) Original O/P (b) Inpainted O/P (c) Blended O/P   * *Static Linking:* The project was made by statically linking the OpenCV libraries to the executable which in turn increases the size of the application but saves the user from the hectic installation of OpenCV from source. Another way to run the application is to provide all the DLLs in the same directory as the application so that while executing it can find the appropriate files to look up to. This was done by building the package using some major commands like BUILD\_SHARED\_LIBS=OFF and BUILD\_WITH\_STATIC\_CRT=OFF. Thus to run the application we’ve made a command line tool or you can open up the terminal and just run the executable with all of the flags you want to provide and the paths of images.   Example COMMAND: As you can see order of images doesn’t matter here!  .\Project3.exe img3.jpg img1.jpg img4.jpg img2.jpg --features orb --matcher homography --save\_graph match.dot |
| THANK YOU   * *Team 1Neuron* |
|  |