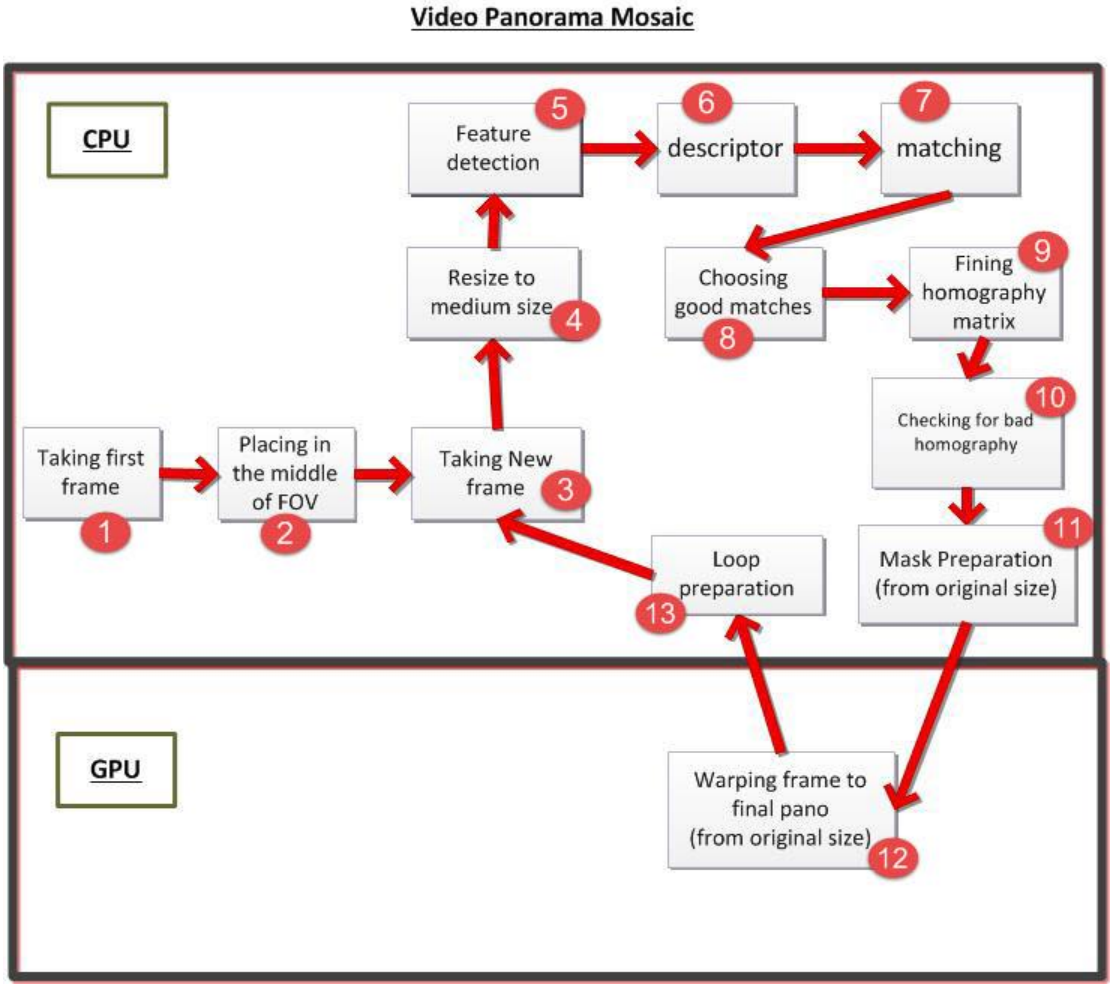
Code explanation:

Here is the whole process flowchart:



1. Taking first frame :

```
	if (device_num >= 0)
	{
		std::cout << "Opening camera device number " << device_num << ". Please wait..." << endl;
		if (!cap.open(device_num))
		{
			std::cout << "Camera device number " << device_num << " does not exst or installed. Please select another device or read from video file." << endl;
			return -1;
		}
```

```
}
else
{
	if (!cap.open(filename))
	{
		std::cout << "Bad file name. Can't read this file." << endl;
		return -1;
	}
}
if (device_num >= 0)
	for (int k = 0; k < 10; k++)
		cap >> img;
else cap >> img;

cv::Mat offset = cv::Mat::eye(3, 3, CV_64F);
int counter = 0;
```

2. Placing in the middle of FOV

```
//////////////////////////starting point
double start_width;
double start_height;

//if the x and y are not set from arguments they will be in the middle of FOV
if (x_start_scale < 0) start_width = img.cols*(XScale / 2 - 0.5);
else start_width = img.cols*(x_start_scale);
if (y_start_scale < 0) start_height = img.rows*(YScale / 2 - 0.5);
else start_height = img.rows*(y_start_scale);

// making the final image and copying the first frame in the middle of it
Mat final_img(Size(img.cols * XScale, img.rows * YScale), CV_8UC3);
Mat f_roi(final_img, Rect(start_width, start_height, img.cols, img.rows));
img.copyTo(f_roi);
img.copyTo(img_last);
```

3. Taking new frame:

```
//starting loop

	cap >> img_cur;

	if (img_cur.empty()) break;
	cvNamedWindow("current video", CV_WINDOW_NORMAL);
	imshow("current video", img_cur);
	waitKey(1);
```

4. Resize to Medium Size :

```
resize(img_cur, img_cur_scaled, Size(), work_scale, work_scale);
if (img_cur.empty()) break;
//convert to grayscale
cvtColor(img_cur_scaled, gray_curimage, CV_RGB2GRAY);
```

5. Feature detection:

```
//First step: feature extraction
t = getTickCount();
detector->detect(gray_curimage, keypoints_cur);
cout << "features= " << keypoints_cur.size() << endl;
cout << "detecting time: " << ((getTickCount() - t) / getTickFrequency())
<< endl;
```

6. Descriptor:

```
//Second step: descriptor extraction
t = getTickCount();

extractor->compute(gray_curimage, keypoints_cur, descriptors_cur);
cout << "descriptor time: " << ((getTickCount() - t) / getTickFrequency())
<< endl;
t = getTickCount();
```

7. Matching:

```
//Third step: match with BFMatcher
if (descriptors_last.type() != CV_32F) {
	descriptors_last.convertTo(descriptors_last, CV_32F);
}
if (descriptors_cur.type() != CV_32F) {
	descriptors_cur.convertTo(descriptors_cur, CV_32F);
}
vector< DMatch > matches;
BFMatcher matcher(NORM_L2, true);

matcher.match(descriptors_last, descriptors_cur, matches);
if (matches.empty()){
	last_err = false;
	continue;
}

vector< DMatch > good_matches;
vector< DMatch > good_matches2;

vector<Point2f> match1, match2;
sort(matches.begin(), matches.end(), cmpfun);
```

8. Choosing good matches:

```
		//matching filter number 0. It will calculate the distance of matches and
the first 50 ones which has less 4 times distance than max distance will be considered.
		if (match_filter == 0)
		{

			double max_dist = 0; double min_dist = 100;

			for (int i = 0; i < matches.size(); i++)
			{
				double dist = matches[i].distance;
				if (dist < min_dist) min_dist = dist;
				if (dist > max_dist) max_dist = dist;
			}
			for (int i = 0; i < matches.size() && i < 50; i++)
			{
				if (matches[i].distance <= 4 * min_dist)
				{
					good_matches2.push_back(matches[i]);
				}
			}
		}



		//matching filter number 1. It will calculate the norms od distances and
the it has the same matching filter number 0 at the end to reduce and find the best the
matching features.
		else if (match_filter == 1) {
			int counterx;
			float res;
			for (int i = 0; i < (int)matches.size(); i++){
				counterx = 0;
				for (int j = 0; j < (int)matches.size(); j++){
					if (i != j){
						res =
cv::norm(keypoints_last[matches[i].queryIdx].pt - keypoints_last[matches[j].queryIdx].pt)
- cv::norm(keypoints_cur[matches[i].trainIdx].pt -
keypoints_cur[matches[j].trainIdx].pt);
						if (abs(res) < (img.rows*0.03 + 3)){ //this
value(0.03) has to be adjusted
							counterx++;
						}
					}
				}
				if (counterx > (matches.size() / 10)){
					good_matches.push_back(matches[i]);
				}
			}

			double max_dist = 0; double min_dist = 100;
			for (int i = 0; i < good_matches.size(); i++)
			{
				double dist = good_matches[i].distance;
				if (dist < min_dist) min_dist = dist;
				if (dist > max_dist) max_dist = dist;
```

```
                }

                cout << "max_dist:" << max_dist << endl;
                cout << "min_dist:" << min_dist << endl;
                //take just the good points
                if ((max_dist == 0) && (min_dist == 0))
                {
                        last_err = false;
                        continue;
                }
                sort(good_matches.begin(), good_matches.end(), cmpfun);
                for (int i = 0; i < good_matches.size() && i < 50; i++)
                {
                        if (good_matches[i].distance <= 4 * min_dist)
                        {
                                good_matches2.push_back(good_matches[i]);
                        }
                }
        }
        //no filter
        else if (match_filter == 2)
                good_matches2 = matches;



        cout << "goodmatches features=" << good_matches2.size() << endl;

        vector< Point2f > obj_last;
        vector< Point2f > scene_cur;

        //take the keypoints

        for (int i = 0; i < good_matches2.size(); i++)
        {
                obj_last.push_back(keypoints_last[good_matches2[i].queryIdx].pt);
                scene_cur.push_back(keypoints_cur[good_matches2[i].trainIdx].pt);
        }
        cout << "match time: " << ((getTickCount() - t) / getTickFrequency()) <<
endl;
        t = getTickCount();
        Mat mat_match;

        if (scene_cur.size() >= 4)
        {
                first_time_4pointfound = true;
                //drawing some features and matches
                drawMatches(img_last, keypoints_last, img_cur, keypoints_cur,
good_matches2, mat_match);
                cvNamedWindow("match", WINDOW_NORMAL);
                imshow("match", mat_match);
                if (counter == 1) waitKey(0);
```

9. Finding Homography

```
// finding homography matrix

if (warp_type == "affine")
{
	H2 = estimateRigidTransform(scene_cur, obj_last, 0);
	if (H2.data == NULL) {
		last_err = false;
		good_matches.clear();
		good_matches2.clear();
		scene_cur.clear();
		obj_last.clear();
		continue;
	}

	H.at<double>(0, 0) = H2.at<double>(0, 0);
	H.at<double>(0, 1) = H2.at<double>(0, 1);
	H.at<double>(0, 2) = H2.at<double>(0, 2);
	H.at<double>(1, 0) = H2.at<double>(1, 0);
	H.at<double>(1, 1) = H2.at<double>(1, 1);
	H.at<double>(1, 2) = H2.at<double>(1, 2);
	cout << "H=" << H2 << endl;
}

else if (warp_type == "perspective")
{
	H = findHomography(scene_cur, obj_last, CV_RANSAC, 3);
	if (H.empty()){
		good_matches.clear();
		good_matches2.clear();
		scene_cur.clear();
		obj_last.clear();
		continue;
	}
	cout << "H=" << H << endl;

}
```

10. Checking for bad Homography

```
// using corelations and norms to find the errors and skip that
frame. Bad matching can lead to bad homography matrix
H.convertTo(H, CV_32F);
H_old.convertTo(H_old, CV_32F);
Mat correlation;
matchTemplate(H_old, H, correlation, CV_TM_CCOEFF_NORMED);
cout << "correlation:" << correlation << endl;
double nownorms = norm(H - H_old, 2);
H.convertTo(H, CV_64F);
H_old.convertTo(H_old, CV_64F);
cout << "now norm:" << nownorms << endl;
cout << "miangin norm:" << norms << endl;

if (norms == 0)	norms = nownorms;
```

```
			if ((nownorms > 2 * norms) && (abs(correlation.at<float>(0)) < 0.8)
|| (nownorms > 10 * norms))
			{
				if (!last_err && (counter != 1))
				{
					cout <<
"Errorrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr" << endl;
					if (log_flag){
						string name = "Error_frame_" +
to_string(counter) + ".jpg";
						imwrite(name, panorama_temp);
							}
					last_err = true;
					good_matches.clear();
					good_matches2.clear();
					scene_cur.clear();
					obj_last.clear();
					continue;
				}
				else last_err = false;
			}
			else{
				norms = (norms* (counter - 1) + nownorms) / counter;
				last_err = false;

			}
```

Homography Calculations :

```
			//take the x_offset and y_offset
			/*the offset matrix is of the type

			|1 0 x_offset |
			|0 1 y_offset |
			|0 0 1        |

			*/
			offset.at<double>(0, 2) = start_width;
			offset.at<double>(1, 2) = start_height;
			if (first_time_4pointfound == true && second_time_4pointfound ==
false)
			{
				H_kol = offset*H;
				second_time_4pointfound = true;

			}
			else
			{
				H_kol = H_kol*H;
			}
			cout << "Homography time: " << ((getTickCount() - t) /
getTickFrequency()) << endl;
			t = getTickCount();
```

Warping Image:

```
                    //using gpu for applying homography matrix to images
                    gpu::GpuMat rImg_gpu, img_cur_gpu;
                    img_cur_gpu.upload(img_cur);
                    gpu::warpPerspective(img_cur_gpu, rImg_gpu, H_kol, size_wrap,
INTER_NEAREST);
                    rImg_gpu.download(rImg);


                    cout << "warpPerspective time: " << ((getTickCount() - t) /
getTickFrequency()) << endl;
                    t = getTickCount();

                    //ROI for img1
                    t = getTickCount();
                    mask = cv::Mat::ones(final_img.size(), CV_8U) * 0;

                    vector<Point2f> corners(4), corner_trans(4);

                    corners[0] = Point2f(0, 0);
                    corners[1] = Point2f(0, img.rows);
                    corners[2] = Point2f(img.cols, 0);
                    corners[3] = Point2f(img.cols, img.rows);
                    perspectiveTransform(corners, corner_trans, H_kol);
```

11. Mask Preparation:

```
                    //making mask
                    vector<Point> line1;
                    int Most_top_x_corrner = 0;

                    line1.push_back(corner_trans[0]);
                    line1.push_back(corner_trans[2]);
                    line1.push_back(corner_trans[3]);
                    line1.push_back(corner_trans[1]);
                    fillConvexPoly(mask, line1, Scalar::all(255), 4);
                    line(mask, corner_trans[0], corner_trans[2], Scalar::all(0), 5, 4);
                    line(mask, corner_trans[2], corner_trans[3], Scalar::all(0), 5, 4);
                    line(mask, corner_trans[3], corner_trans[1], Scalar::all(0), 5, 4);
                    line(mask, corner_trans[1], corner_trans[0], Scalar::all(0), 5, 4);
                    cout << "mask time: " << ((getTickCount() - t) / getTickFrequency())
<< endl;
```

12. Warping frame to final pano (from original size):

```
                    rImg.copyTo(final_img, mask);
                    final_img.copyTo(panorama_temp);
```

GUI text and rectangle :

```
                    line(final_img, corner_trans[0], corner_trans[2], CV_RGB(255, 0, 0),
5, 4);
                    line(final_img, corner_trans[2], corner_trans[3], CV_RGB(255, 0, 0),
5, 4);
                    line(final_img, corner_trans[3], corner_trans[1], CV_RGB(255, 0, 0),
5, 4);
                    line(final_img, corner_trans[1], corner_trans[0], CV_RGB(255, 0, 0),
5, 4);

                    cout << "GUI time: " << ((getTickCount() - t) / getTickFrequency())
<< endl;
                    fps = 1 / ((getTickCount() - start_app) / getTickFrequency());
                    putText(final_img, "fps=" + std::to_string(fps), Point2f(100, 100),
CV_FONT_NORMAL, 1.5, CV_RGB(255, 0, 0), 4, 8);
                    putText(final_img, "frame=" + std::to_string(counter), Point2f(100,
150), CV_FONT_NORMAL, 1.5, CV_RGB(255, 0, 0), 4, 8);
                    namedWindow("Img", WINDOW_NORMAL);
                    imshow("Img", final_img);
```

13. Loop preparation

```
            ////looop preparation
                    gray_curimage.copyTo(gray_lastimage);
                    img_cur.copyTo(img_last);
                    keypoints_last = keypoints_cur;
                    descriptors_last = descriptors_cur;
                    H.copyTo(H_old);
                    last_err = false;
                    t = getTickCount();

                    cout << "loop prepration time: " << ((getTickCount() - t) /
getTickFrequency()) << endl;
```