NXP Semiconductors
Reference Manual

Document Number: SVARM
Rev. 1, 07/2018

Surround View Application

1. Introduction

This document describes the software theory of the NXP's 360-degree Surround View system which contains several separate applications. This document also describes how to build each application from sources for a specific SoC using the Yocto project. A separate part of this document describes how to properly perform the system calibration settings and all necessary preparations before running the system (automatic) calibration. The Application Programming Interface (API) of the system calibration application is described in the final and the most detailed section of this document. It is the most important part of the whole Surround View system.

The Surround View software's ability can be simply demonstrated using its development kit. It is assumed that the interested user can build this kit themselves using the released hardware and software sources. For more information on how to properly run and set up the Surround View development kit using the i.MX 8QM device, see the separate application user's guide [1].

Contents

2. Software overview

From the software point of view, the whole Surround View 3D application (SV3D) is based on these parts:

- Linux OS. The current image can be created using the Yocto Linux OS project [2] for different target SoCs—the i.MX8 family device (default) or the i.MX6QP.
- The image-capturing application, whose sources are a part of the Surround View software package available in the repository [3]. It is a simple application used for the camera frames' capturing. These frames are used by the camera lens calibration tool consecutively.
- The camera lens calibration tool, which is used for the intrinsic camera parameters calibration. Originally, this is a third-party Matlab toolbox [4]. The detailed application theory of this tool is described on the official web page [4]. The tool is modified by NXP and compiled using the Matlab Runtime compiler [5]. The latest sources are available in the relevant repository [6] and the tool's executable file is available from the surround view software archive [3]. The use of the Surround View lens calibration is described in the *Surround View Application User's Guide* [1]. The intrinsic parameters represent the optical center and focal length of the camera.
- System (automatic) calibration application. This is a one-shot and off-line preprocessing application to calculate all the necessary extrinsic camera parameters, which represent the location of the camera in the 3D scene. The application executes all preprocessing calculations and generates the files for texture mapping (rendering). The calibration should be done only once when the camera system is fixed and the cameras don't move relatively to each other. Otherwise, the calibration process must be repeated. The sources of this application are a part of the Surround View software package available at the repository [3].
- The real-time rendering application, which maps the camera frames on a prepared 3D mesh using an internal GPU and blends them. The 3D rendering process uses real camera frames from the Surround View development kit (default mode) or static images in the *App/Content/camera_inputs* folder (optional mode). All sources of this application are a part of the Surround View software package available in the repository [3].

NOTE

Only the lens camera calibration tool runs on the Windows® OS platform. The other applications (image capturing, system calibration, rendering) run on the Yocto Linux OS platform.

Figure 1 shows the detailed application block diagram of the complete Surround View system and the Linux OS to Windows OS applications code flow.

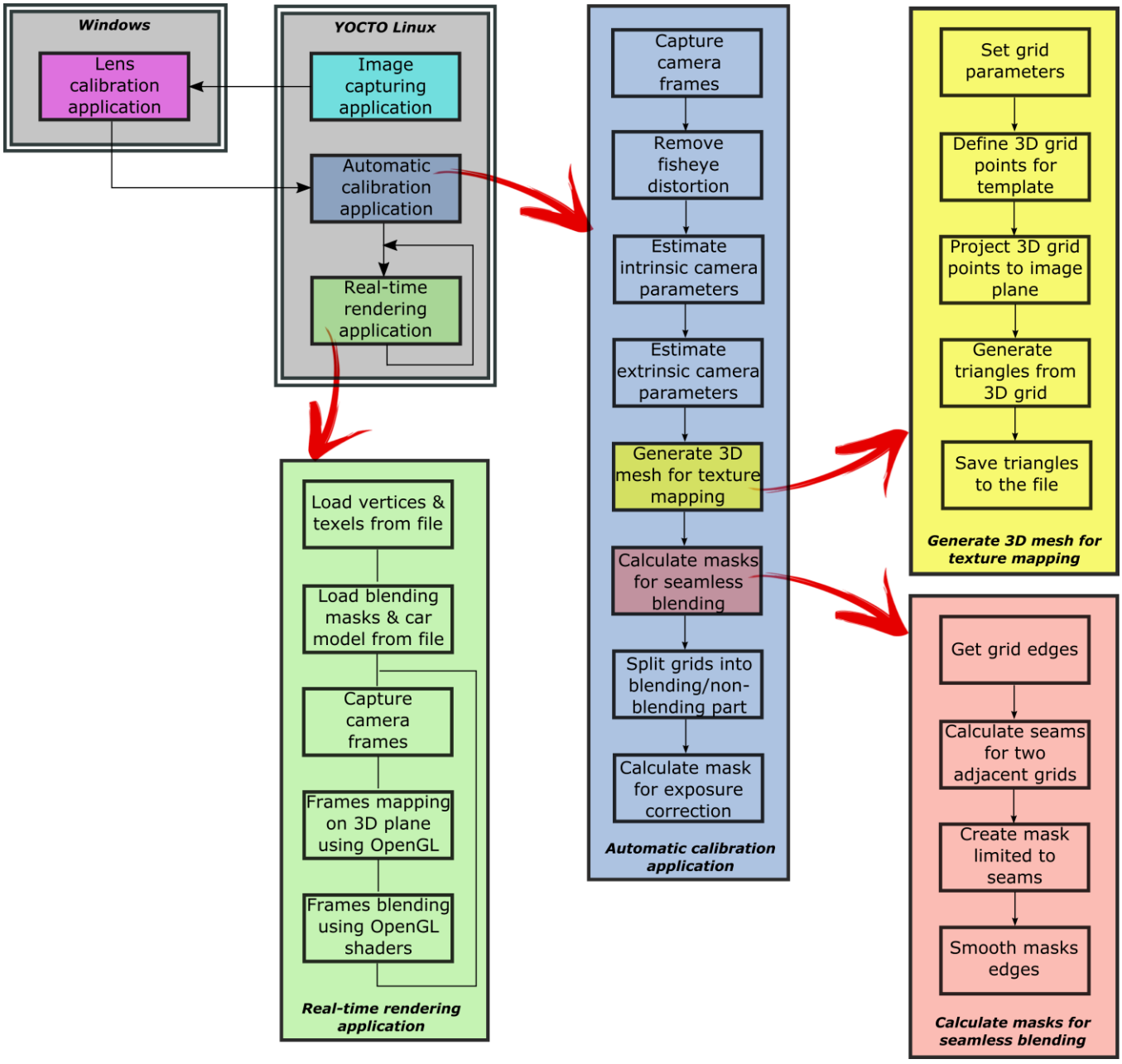

Figure 1. Surround view application software block diagram

3. Building applications from sources

All Yocto Linux OS Surround View applications can be built from sources. Here is how to do it:

- Download the latest Yocto image from the repository [2]. Various image configurations are supported, depending on the graphical backend and the target device. Table 1 summarizes the versions of all Yocto images that were used to verify the Surround View application ability.

Table 1. Surround view application verified images

Version of	Target device	
	i.MX8 family device [1]	I.MX6QP [2]
Image	L4.9.51-8QM-beta2-rc2(119)	L4.1.15
GPU driver	6.2.4.138003	5.0.11.41671
OpenCV	3.4	3.1
Kernel	4.9.51-8qm_beta2_8qxp_beta+g423d9423	4.1.15-0001-g5bc7305

1. Tested on the XWayland and framebuffer graphical backends on the i.MX 8QM.
2. Tested on the X11 and framebuffer graphical backends. Additional MIPI camera patches are needed (see this link).

- Save this image to the SD card. The image contains both the root file system and kernel.
- Download the latest Surround View application source code from the repository [3] and save the sources to the */home/root/SV3D-1.1* folder on the system SD card.
- Go to the *App/Sources* subfolder and build the application using the make command. Use the right build command with regards to the current graphical backend, the target device, and the input you need (see also Table 2 for all possible build configurations). Alternatively, the instructions to build the application are included in the *README.md* file in the application root folder. The make command automatically creates the */App/Build* and */Tools/CamCapture* subfolders after its first use (if these directories don't exist already). The make command builds the whole Surround View project and creates these binary files (applications):
  - Image capturing binary file (*capturing*) in the */Tools/CamCapture/* folder.
  - Automatic (system) calibration binary file (*auto_calib_1.1*) in the */App/Build/* folder.
  - Rendering binary file (*SV3D-1.1*) in the */App/Build/* folder.

Table 2. Surround View application build commands

Inputs	Graphical backends	Target device	
		i.MX8 family device	I.MX6QP
Cameras	XWayland	make -f Makefile.wl DEVICE=IMX8QM	—
	Framebuffer	make -f Makefile.fb DEVICE=IMX8QM	make -f Makefile.fb DEVICE=IMX6QP
	X11	—	make -f Makefile.x11 DEVICE=IMX6QP
Images[1]	XWayland	make -f Makefile.wl INPUT=image	
	Framebuffer	make -f Makefile.fb INPUT=image	
	X11	make -f Makefile.x11 INPUT=image	
Videos[1]	XWayland	make -f Makefile.wl INPUT=videos	
	Framebuffer	make -f Makefile.fb INPUT=videos	
	X11	make -f Makefile.x11 INPUT=videos	
Raw[1]	XWayland	make -f Makefile.wl INPUT=raw	
	Framebuffer	make -f Makefile.fb INPUT=raw	
	X11	make -f Makefile.x11 INPUT=raw	

1. The input images (or videos) are saved to the /App/Content/camera_inputs folder.

4. System calibration arrangement

Before running the automatic (system) calibration application, perform these preparations:

1. Perform the calibration (system) settings for each camera and for the whole system.
2. Prepare the system calibration pattern and template definitions for each camera.
3. Capture the camera frames using the checkerboard calibration pattern and the image capture application.
4. Generate the radial models for each camera using the OcamCalib toolbox for Matlab [4], [6].

4.1. System calibration settings

The system parameters are defined in the *App/Content/settings.xml* file. These settings are used by all Surround View applications, but most of the settings' values are used by the automatic calibration application. There are several specific sections in this file:

- *<path>*
  - *<camera_inputs>*—the path to the static images or video files. These are used only when the automatic calibration (and rendering) is done from the static images or video files (not from the cameras).
  - *<camera_model>*—the path to the camera model files. This model is stitched to the final composite 3D surround view in the rendering application.
  - *<template>*—the path to the template point files.
- *<camera>*
  - *<number>*—the number of cameras (can be 1, 2, 3, or 4).
  - *<camerax> (x = 1…4)*—the *x* camera parameters.
  - *<height>—the camera input frame height.*
  - *<width>*—the camera input frame width.

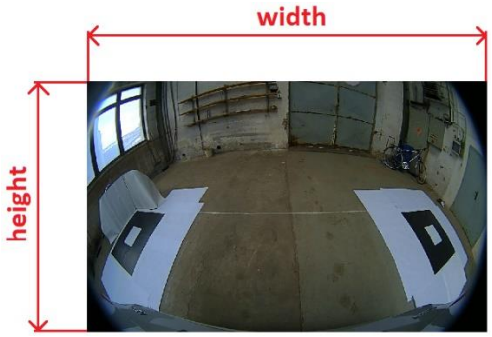

Figure 2. Fisheye camera frame resolution

  - *<sf>*—the scale factor for the lens undistorting.

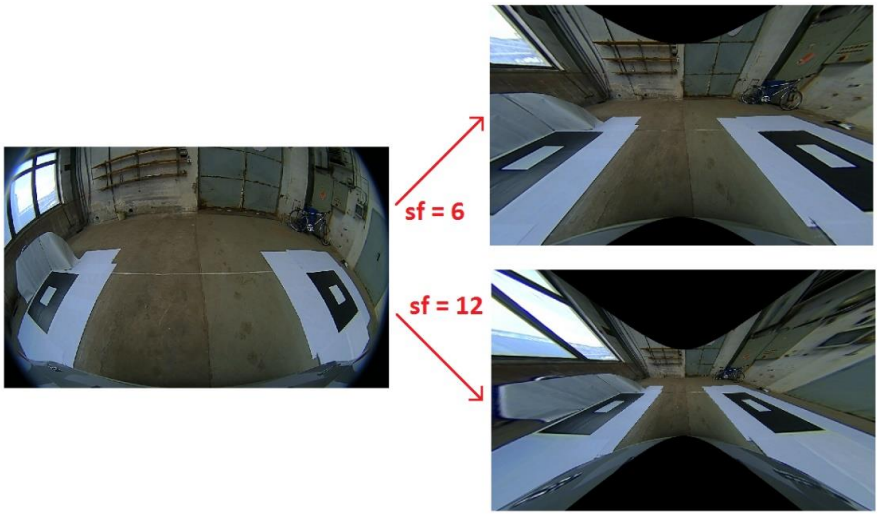

Figure 3. Fisheye camera frame scaling

– <*roi*>—the region of interest for contours' searching. The number is in the percentage of the input frame height from the bottom to the top.

Figure 4. Fisheye camera ROI definition

– <*contour_min_size*>—the minimum perimeter of the searched contour (in pixels). If there are contours with a perimeter smaller than <*contour_min_size*> on the pictures, they are ignored.

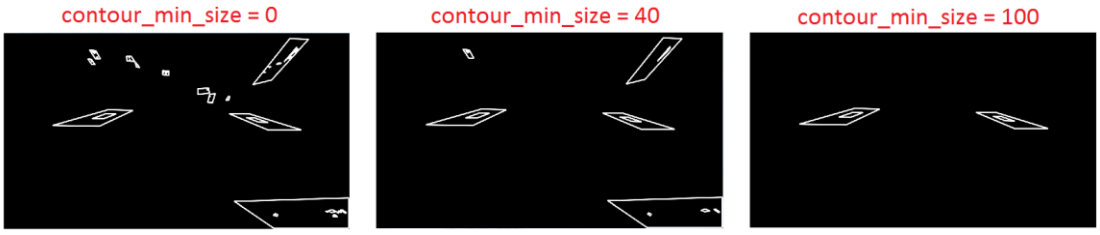

Figure 5. Search contours' definition

– <*chessboard_num*>—the number of checkerboard images that are used to estimate the intrinsic camera parameters. The checkerboards are in the *App/Content/camera_model/chessboard_x* folders (x = 1…4).

- <*display*>
  – <*height*>—display height.
  – <*width*>—display width.

- <*grid*>

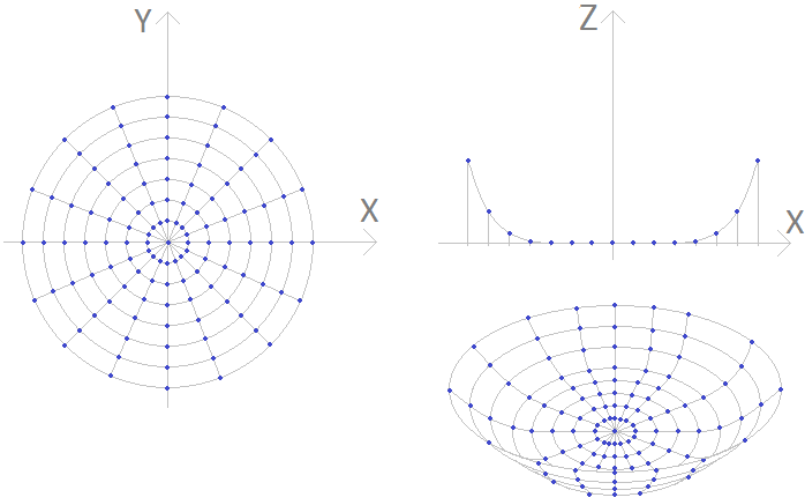

Figure 6. Grid definition

– <*angles*>—every quadrant of the circle is divided into this number of arcs.

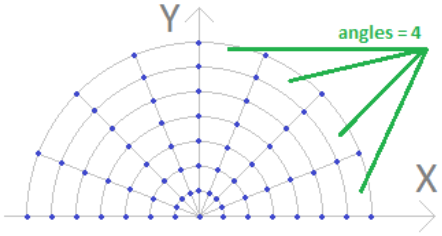

Figure 7. Angles definition

– <*start_angle*>—it is not necessary to define the grid through a whole semi-circle. The <*start_angle*> sets the circle sector for which the grid is generated.

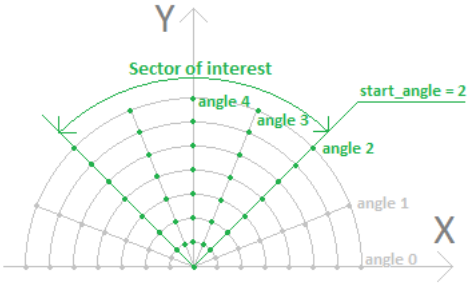

Figure 8. Start angle definition

– <*nop_z*>—the number of points in the *z*-axis.

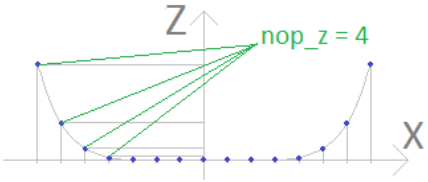

Figure 9. z-axis granularity definition

– <*step_x*>—the step in the *x*-axis that is used to define the grid points in the *z*-axis. The step in the *z*-axis is defined as: step_z[i]= (i * step_x)^2 for i = 1, 2, ... number of points.

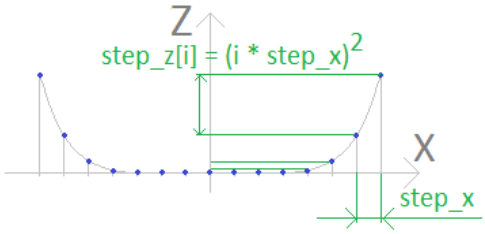

Figure 10. x-axis step definition

– <*radius*>—the scale of the flat bowl bottom area radius related to the template radius.

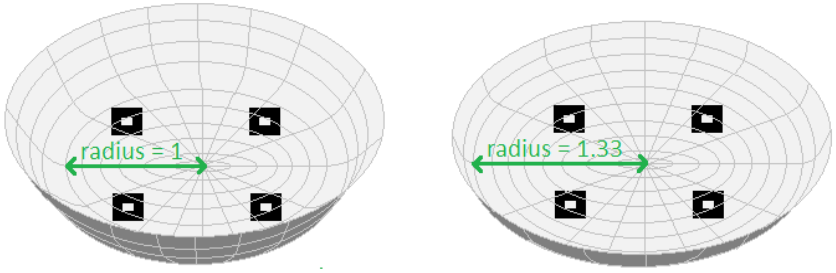

Figure 11. Flat bowl area definition

- <*mask*>

  To achieve a seamless blending of frames, the left and right edges of the masks are smoothed.

  – <*smooth_angle*>—defines the area in which the mask edges are smoothed. The original seam divides the smoothing angle into two angles with equal measures (angle bisector). The angle-based smoothing is applied only for a flat base. The seam at the bowl side is smoothed with a constant width.

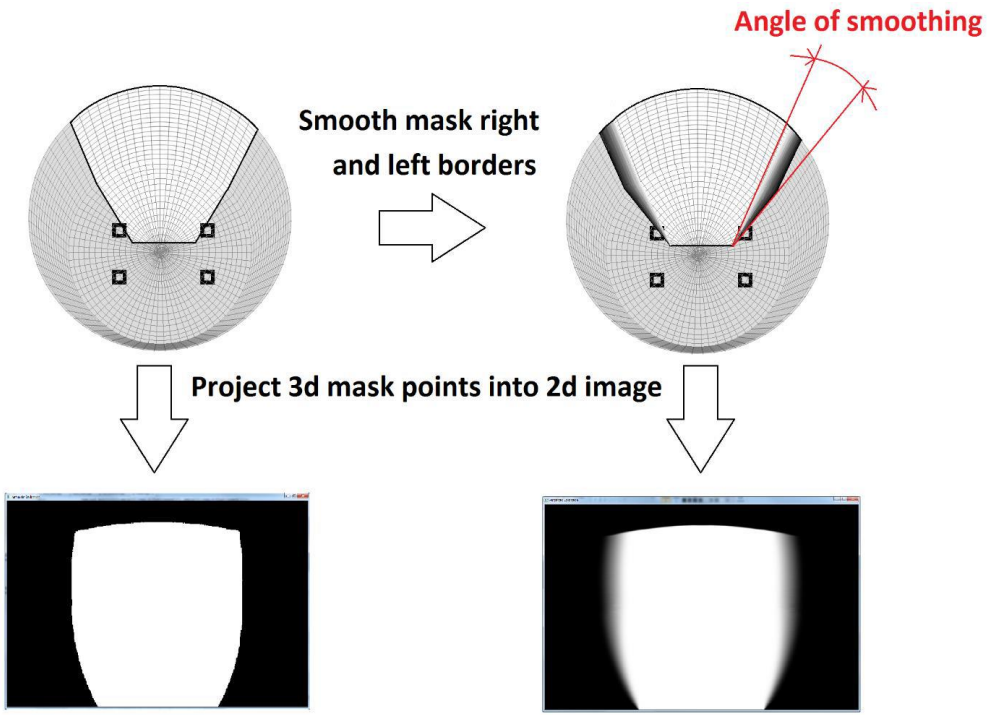

Figure 12. Smoothing area

- *<fb>*

  When the application runs in the framebuffer or XWayland graphical backends, both the keyboard and mouse events are read from the */dev/input/by-path/* devices and unique for the particular board. It is necessary to define the keyboard, mouse, and display devices. For the x11 graphical backend, these settings are ignored.
  - *<keyboard>*—the absolute path of a keyboard device.
  - *<mouse>*—the absolute path of a mouse device.
  - *<display>*—the absolute path of a display device.
- *<car_model>*
  - *<x_scale>*—the car model scale in the *x*-axis.
  - *<y_scale>*—the car model scale in the *y*-axis.
  - *<z_scale>*—the car model scale in the *z*-axis.

4.2. System calibration template

To calibrate the 4-camera system, the calibration pattern of a known size is used. See Figure 13 for an example of the rectangular pattern.

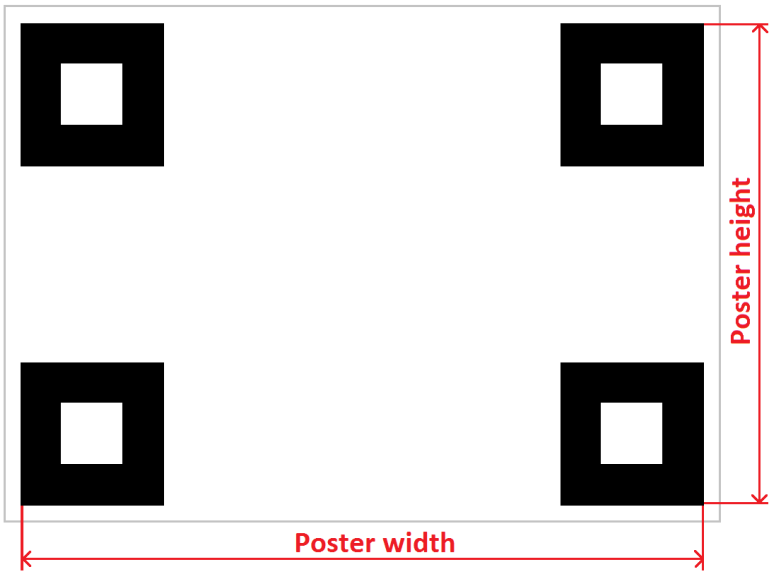

Figure 13. Extrinsic camera parameters' calibration pattern

Each camera can see two adjacent patterns. Therefore, the calibration poster is divided into four separate templates (or sectors); one for each camera (see Figure 14).

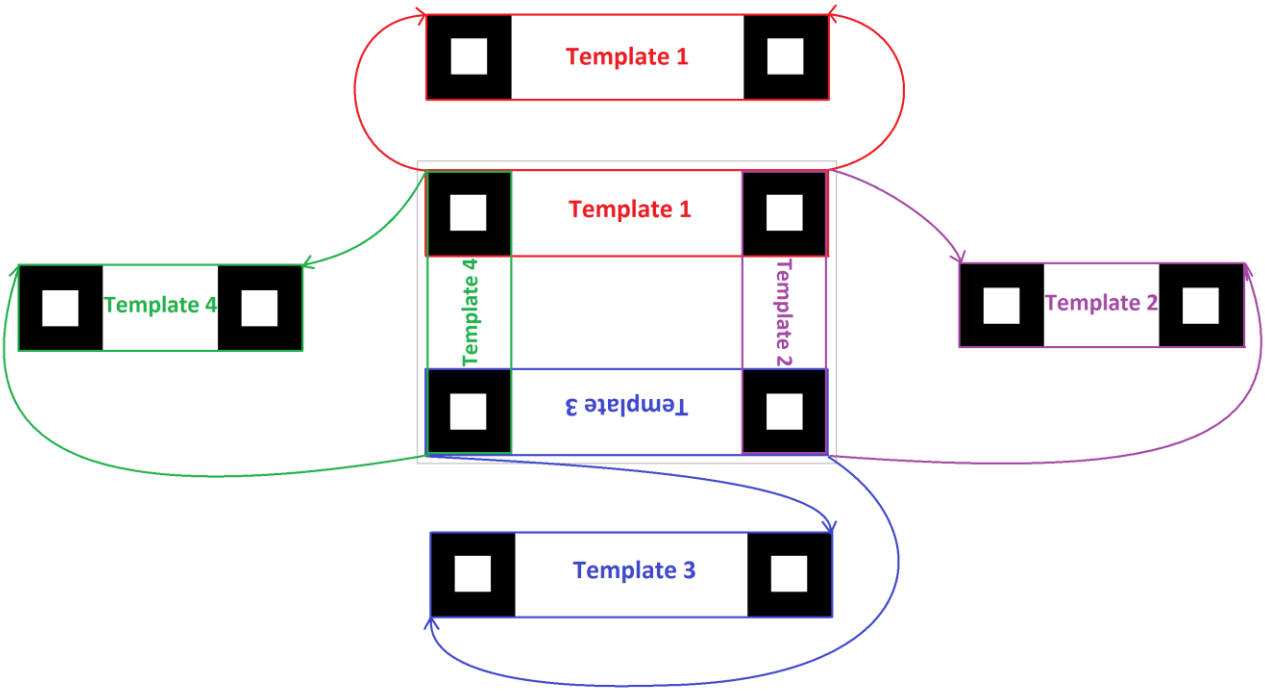

Figure 14. Calibration pattern sectors

The application estimates the extrinsic camera position in relation to the calibration template and the coordinates of the template corners (16 points per template). It identifies all pattern corners in the captured camera images and establishes a correspondence with the real-world distances of these corners. Using these correspondences, the extrinsic camera parameters (rotation and translation vectors) are estimated.

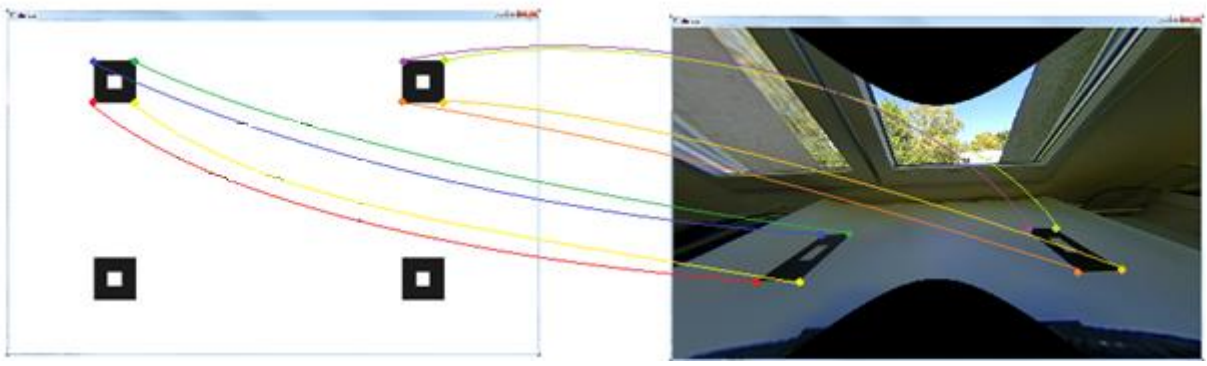

Figure 15. Pattern-to-camera frame corners' mapping

Before running the automatic calibration application, it is necessary to determine the template points and save the points' coordinates to the *App/Content/template/template_y.txt* files, where y = 1…4 is the camera index. The coordinates in these files are divided into two columns: the first column contains the *x*-coordinates and the second column contains the *y*-coordinates (see Figure 16). The points must be defined in the right-ascending order, that means from point 1 to point 16.

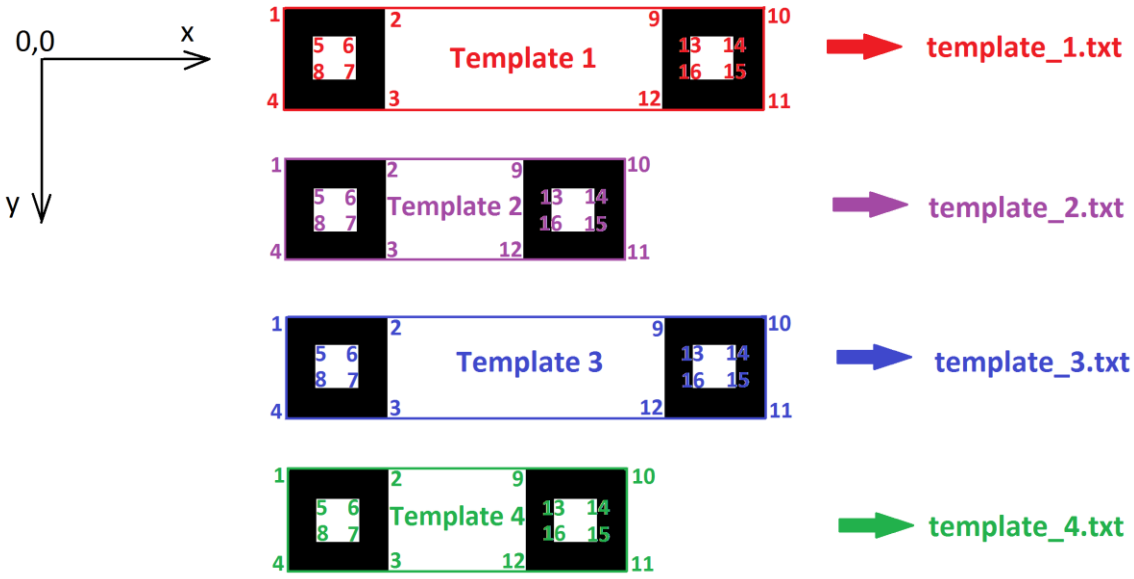

Figure 16. Pattern coordinates' definition

It is not necessary to define the template points for a real template size, but it is necessary to define all points with a right proportion between them. For example, the point coordinates can be defined in pixels or millimeters.

NOTE

For a square-sized calibration pattern, all four templates are the same.

4.3. Image capturing

A special checkerboard calibration pattern is used for the image capturing before running the intrinsic camera calibration (see Figure 17).

Figure 17. Intrinsic camera parameters' calibration pattern

Use the image capturing binary file in the *Tools/CamCapture* folder and follow all steps described in the "Image capturing" section of the *Surround View Application User's Guide* [1]. This application returns several images per camera in a JPEG format, which are then used by the lens calibration tool.

Table 3. Image capturing application dependency

Application inputs	Real-time camera frames
Application outputs	/Tools/CamCapture/frameX_Y.jpg 1)
Application settings	/App/Content/settings.xml
Binary file name	/Tools/CamCapture/capturing
Target platform	i.MX8 family device, i.MX6QP
Target OS	Yocto Linux OS

1. With subsequent copying to the relevant App/Content/camera_models/ folder

NOTE

The captured images are used for the intrinsic camera parameters' calculation by both the lens and automatic calibration applications.

4.4. Camera model generation

To generate the radial camera models, the Scaramuzza toolbox for Matlab is used. Alternatively, the pre-built binary file can be used for that. The application generates text files with radial camera model coefficients, which are used by the automatic calibration application consecutively. The text files are called *calib_result_y.txt*, where y = 1…4 is the camera index. These files are in the *App/Content/camera_models* folder. Follow all the steps described in the "Camera intrinsic parameters estimation tool" section of the *Surround View Application User's Guide* [1]. This is a simplified version of the application usage summary:

1. Copy all input (captured) JPEG images from the target platform to the Windows OS PC.
2. Perform the lens (intrinsic) calibration using the OCamCalib toolbox for Matlab or OCamCalib executable file for Matlab Runtime.
3. Copy all generated model text files for the target platform to its relevant folder.

Table 4. Lens calibration application dependency

Application inputs	frameX_Y.jpg 1)
Application outputs	calib_results_X.txt 2)
Application name	OCamCalib Matlab toolbox v3.0 or ocam_calib.exe for Matlab Runtime
Target OS	64-bit Windows OS PC with Matlab

1. Copy from the target-platform Tools/CamCapture/ folder
2. With subsequent copying to the relevant /App/Content/camera_models/ folder

5. Automatic calibration application

The automatic (system) calibration is a one-shot, stand-alone application which performs the whole preprocessing calculation and generates the files for texture mapping. It uses only the CPU, while the

final rendering application uses the GPU. To estimate the extrinsic camera parameters, a special calibration pattern of a known size is used (see Figure 13). The process of the automatic (system) camera calibration using the i.MX8 family device is described in the "System calibration" section of the *Surround View Application User's Guide* [1].

Before running the automatic calibration, make sure that all files are in the correct folders (see Figure 18).

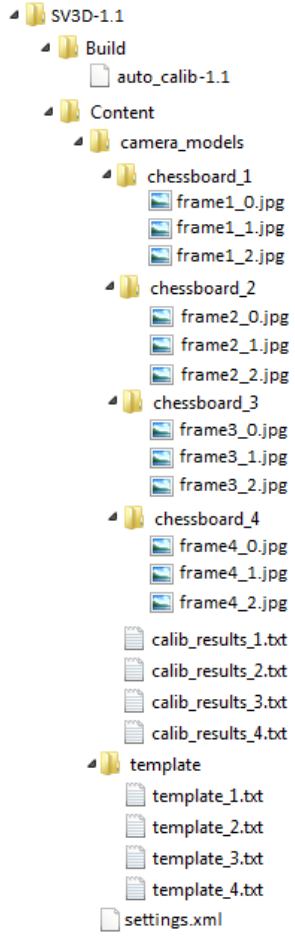

Figure 18. System calibration application dependency

Table 5. Automatic calibration application dependency

Application inputs	/App/Content/camera_models/calib_results_X.txt
	/App/Content/camera_models/chessboard_X/frameX_Y.jpg
	/App/Content/template/template_X.jpg
Application outputs	/App/Build/arrayN
	/App/Build/maskN.jpg
	/App/Build/compensator/
Application settings	/App/Content/settings.xml
Binary file name	/App/Build/auto_calib_1.1
Target platform	i.MX8 family device, i.MX6QP
Target OS	Yocto Linux OS

6. Automatic calibration API definition

There are five main steps in the automatic (system) calibration process:

1. Read the calibration parameters from the *settings.xml* file located in the */App/Content* folder.
2. Create all camera objects and set the objects' properties (camera objects are independent of each other).
3. Create a grid for each camera (the grid objects are independent of each other).
4. Create the masks for all cameras at once. Split the grids according to these masks.
5. Create the exposure compensator for all cameras at once.

Individual classes are created for each step: XMLParameters, Camera, Grid, Mask, and Compensator classes. See the following subsections for their description.

6.1. XMLParameters class

The XMLParameters object contains different parameters which are used to define the attributes of other classes and described in Section 4.1, "System calibration settings".

Public attributes (accessible from outside the class):

- string camera_inputs—the path to the camera calibration frame files.
- string camera_ model—the path to the camera model files.
- string tmplt—the path to the template points' files.
- **int** camera_num—the number of cameras.
- CamParam cameras[4]—the cameras' parameters.
- **int** height—the camera frame height in pixels.
- **int** width—the camera frame width in pixels.
- **float** sf—the de-fisheye scale factor.
- **int** roi—the region of interest for the contours' searching (in percentage of the input frame height from the bottom to the top).
- **int** cntr_min_size—the minimum length of the contour in pixels.
- **int** disp_height—display height.
- **int** disp_width—display width.
- **bool** show_debug_img—debug mode.
- **int** grid_angles—every quadrant of the circle is going to be divided into this number of arcs.
- **int** grid_start_angle—this parameter sets a circle segment for which the grid is generated.
- **int** grid_nop_z—the number of points in the z-axis.
- **float** grid_step_x—a step in the x-axis for the bowl side which is used to define the grid points in the z-axis.
- **float** bowl_radius—the bowl radius scale.
- **float** smooth_angle—the mask angle of smoothing.
- string keyboard—the keyboard device.

- string mouse—the mouse device.
- string out_disp—the display device.
- **float** model_scale[3]—the car model scale.

Public methods (accessible from outside the class):

- Read the calibration parameters from an *.xml* file.

```
/*************************************************************************
* @brief        Read the calibration settings from an .xml file.
* @param  in    const char* filename—the name of the .xml file with the project settings.
* @return       The function returns 0 if all settings are read successfully. Otherwise, it returns -1.
*               The bublic properties are set.
* @remarks      The function reads the settings from the .xml file and writes them to
*               the public properties using the libxml2 library.
*************************************************************************/
int readXML(const char* filename);
```

- Get the maximum value of the x-coordinate from a *.tmplt* file with the template points' definition.

```
/*************************************************************************
* @brief        Get the template width (the maximum value of the template vertices
*               x-coordinate).
* @param in     const char* filename—the template file name.
*          out  int* val—the pointer to the output value.
* @return       The function returns -1 if the template file is not found. Otherwise, it returns 0.
* @remarks      The function searches for the maximum value of the template vertices
*               x-coordinate and writes this value to the val.
*************************************************************************/
int getTmpMaxVal(const char* filename, int* val);
```

- Print the XML parameters to the screen.

```
/*************************************************************************
* @brief        Write all the public parameters' values.
* @param        —
* @return       —
* @remarks      The function writes all public parameters values to the screen.
*************************************************************************/
void printParam();
```

6.2. Camera class

The camera object is created for each camera and contains all information about the camera, that is the camera model, the intrinsic and extrinsic parameters, the LUT for the fisheye distortion removal, and so on. The camera objects are independent of each other.

To estimate a camera model, the Scaramuzza toolbox for Matlab is used. The toolbox generates a polynomial camera model and saves it into a *.txt* file. These camera polynomial coefficients are used to remove the fisheye distortion.

The estimation of both the extrinsic and intrinsic camera parameters is made after the fisheye distortion is removed.

Checkerboard images are used to estimate the intrinsic camera parameters. The application searches the checkerboard corners and calculates the camera matrix, knowing that all checkerboard squares are equal.

To estimate the extrinsic parameters, use a special template with patterns of a known size.
The application identifies all pattern corners in the captured camera images and establishes a correspondence with the real-world distance of these corners. Using these correspondences, the extrinsic camera parameters (rotation and translation vectors) are estimated.

It is necessary to set the template parameters and the intrinsic camera parameters before the extrinsic camera parameters are calculated.

Public attributes (accessible from outside the class):

- **int** index—camera index. The camera index defines the angle at which the camera input is going to be rotated in the result view (rotation: 1—without rotation, 2—rotated left by 90 degrees, 3—rotated by 180 degrees, 4—rotated right by 90 degrees).
- ocam_model model—radial camera model. This camera model is used to remove the fisheye distortion and must be prepared before running the automatic calibration (see Section 4.4, “Camera model generation”). The path to the camera model must be written to the *settings.xml* file (see Section 4.1, “System calibration settings”).
- **float** sf—scale factor. The scale factor defines the camera FOV after removing the de-fisheye transformation and its value must be written into the *settings.xml* file (see Section 4.1, “System calibration settings”).
- Mat xmap, ymap—x LUT, y LUT. These LUTs are used to remove the fisheye distortion. They are calculated using the radial camera model, scale factor, and camera frame size.
- Size poster—poster size. It is the size of a whole poster with four square patterns (see Section 4.2, “System calibration template”).
- **struct** CameraTemplate tmp—template properties:
  - **char** filename[50]—the name of the file that contains the points’ coordinates definition.
  - Size size—template size. It is not the size of a whole poster but the size of a separate template which is seeable for the camera (it consists of two square patterns). The template size is calculated from the points’ coordinates in a camera class method.
  - uint pt_count—points’ number. The points’ number is obtained from the file with the points’ coordinates which must be prepared before running the automatic calibration (see Section 4.2, “System calibration template”).
  - vector<Point3f> ref_points—points’ coordinates. The points’ coordinates are read from a file which must be prepared before running the automatic calibration (see Section 4.2, “System calibration template”).

Private attributes (inaccessible from outside the class):

- **float** roi—the ROI defines the region in which the pattern contours are searched. It is defined in

the percentage of the input frame height from the bottom to the top and its value is set in the *settings.xml* file (see Section 4.1, “System calibration settings”).

- **int** cntr_min_size—minimum contour size. This attribute defines the minimum perimeter of the searched contour in pixels. If there are contours with a perimeter smaller than this value on the pictures, then they are ignored. The attributes’ value is set in the *settings.xml* file (see Section 4.1, “System calibration settings”).
- **double** radius—template radius. It is the radius of the circle that is circumscribed around the calibration poster in pixels. The value is calculated from the poster size as a half of its diagonal.
- **struct** CameraParameters param—camera parameters:
  - Mat K—the camera matrix (intrinsic camera parameters) is calculated from the checkerboard images (see Section 4.3, “Image capturing”).
  - Mat distCoeffs—the distortion coefficients (extrinsic camera parameters) are calculated from the correspondence between the template points and the points from the calibration image of the template from the camera.
  - Mat rvec—the rotation matrix (extrinsic camera parameter) is calculated from the correspondence between the template points and the points from the calibration image of the template from the camera.
  - Mat tvec—the translation vector (extrinsic camera parameter) is calculated from the correspondence between the template points and the points from the calibration image of the template from the camera.

Public methods (accessible from outside the class):

- Create a camera object.

```
/*****************************************************************************
* @brief          Camera class creator.
* @param   in     char *filename—the .txt file with the polynomial camera model from the
*                                Scaramuzza toolbox for Matlab.
*          in     float sf—scale factor.
*          in     int index—camera index.
* @return         The function returns the Camera* object if the camera model is loaded
*                 successfully; a return value of NULL indicates an error.
*                 The public properties of the Camera object are set: sf, index.
*                 The ROI value is set to the default value—50 %.
* @remarks        The class creator reads the polynomial camera model from the file.
*                 If any problems occur, the creator returns NULL.
*                 Otherwise, it returns the pointer to the Camera object.
*****************************************************************************/
static Camera* create(const char *filename, float sf, int index);
```

- Set the template parameters.

```
/*****************************************************************************
* @brief          Set the template parameters.
* @param  in      char *filename—the .txt file with the coordinates of the reference points.
```

```
*                   Size poster_size—the size of the calibrating poster.
* @return           The function returns 0 if the file exists.
*                   If the file is not found, the function returns -1.
* @remarks          The function calculates and sets the template parameters (points number and
*                   template width and height) from the template points' coordinates which are
*                   defined in the file.
*                   The poster size is used for further normalization of the reference points'
*                   coordinates into [-1, 1].
*********************************************************************************/
int setTemplate(const char *filename, Size poster_size);
```

- Set the intrinsic camera parameters.

```
/*********************************************************************************\
* @brief            Set the camera intrinsic parameters (camera matrix and distortion coefficients).
* @param  in        char *filename—the path to the folder which contains the checkerboard images.
*                   char *filename—the checkerboard .jpg file base name.
*                   int img_num—the number of the calibrating images.
*                   Size patternSize—the number of checkerboard corners in the horizontal and
*                                    vertical directions.
* @return           The function returns 0 if all checkerboard corners are found and they
*                   are placed in a certain order (row by row, left to right in every row).
*                   Otherwise, -1 is returned.
*                   The private property param and public properties xmap and ymap of the Camera
*                   object are set.
* @remarks          The intrinsic camera parameters are calculated for the camera after removing
*                   the fisheye transformation. Therefore, the estimation of the intrinsic camera
*                   parameters is made after the fisheye distortion is removed.
*                   The function calculates the camera matrix K using the checkerboard images.
*                        |fx   0   cx |
*                   K = |0    fy  cy |        (cx, cy)—the principal point at the image center.
*                        |0    0    0 |        fx, fy—the focal lengths in the x- and y-axes.
*                   The distCoeffs distortion coefficients are set to 0 after the defisheye
*                   transformation LUTs are calculated.
*********************************************************************************/
int setIntrinsic(const char *filepath, const char *filename, int img_num, Size patternSize);
```

- Set the extrinsic camera parameters.

```
/*********************************************************************************
* @brief            Set the extrinsic camera parameters (rotation and translation vectors).
* @param  in        const Mat &img—the captured calibrating frame from the camera.
* @return           The function returns 0 if the extrinsic camera parameters are set.
*                   A return value of -1 indicates an error.
*                   The param and radius private properties of the Camera object are set.
* @remarks          The function calculates the extrinsic camera parameters to find an object pose
```

```
*                from the 3D-2D point correspondences. To estimate the extrinsic parameters, a
*                special template with patterns of a known size must be used. The application
*                identifies all pattern corners in the captured camera images and establishes a
*                correspondence with the real-world distance of these corners. Using these
*                correspondences, the extrinsic camera parameters (rotation and translation
*                vectors) are estimated.
*                The estimation of the extrinsic parameters is done after the fisheye distortion
*                is removed.
*                Set the templates' parameters and the intrinsic camera parameters before
*                the extrinsic camera parameters are calculated.
*                Important: the calibration patterns must be visible on the captured frame.
*********************************************************************************/
int setExtrinsic(const Mat &img);
```

- Get the maximum possible bowl height.

```
/*********************************************************************************
* @brief        Get the maximum number of grid rows in the z-axis.
* @param  in    double radius—the radius of the flat circle bottom of the bowl. The radius must be
*               defined in pixels and bigger than the half value of the template diagonal.
*               double step_x—the step in the x-axis which is used to define the grid points in the
*               z-axis.
*               Step in the z-axis: step_z[i] = (i * step_x)^2, i = 1, 2, ... - the number of points.
* @return      The function returns the maximum number of grid rows in the z-axis which can
*               be rendered for the defined radius and step_x. It means that if you add one more
*               grid row to the z-axis, then the vertexes of this row do not belong to the input
*               camera frame. It is going to be outside of the camera FOV.
* @remarks     The output is calculated for the input values of the radius and step.
*********************************************************************************/
int getBowlHeight(double radius, double step_x);
```

- Get the base radius.

```
/*********************************************************************************
* @brief        Get the base radius.
* @param        —
* @return       double—the function returns the value of a private class attribute radius, which is
*               defined as the radius of the circle which is circumscribed around the calibration
*               poster in pixels. The value is calculated from the poster size as a half of its
*               diagonal.
*********************************************************************************/
double getBaseRadius();
```

- Set the ROI for contour searching.

```
/*********************************************************************************
* @brief        Define the region of interest for contours' searching.
```

```
* @param  in      uint val—the region of interest in which the pattern contours are going to be
*                 searched. It is defined in the percentage of the input frame height from the bottom
*                 to the top.
* @return         The private attribute ROI is set.
*******************************************************************************/
void setRoi(uint val);
```

- Set the minimum contours' size.

```
/*******************************************************************************
* @brief          Set the empiric bound for the minimum allowed perimeter for contour squares.
* @param  in      uint val—the empiric bound for the minimum allowed perimeter for contour
*                 squares.
* @return         The private attribute cntr_min_size is set.
*******************************************************************************/
void setContourMinSize(uint val);
```

- Remove the fisheye distortion from the input frame.

```
/*******************************************************************************
* @brief          Remove the fisheye distortion from the image.
* @param  in      Mat &img—input fisheye image.
*         out     Mat &out—output undistorted image.
* @return         —
*******************************************************************************/
void defisheye(Mat &img, Mat &out);
```

Example 1. Camera object creation

```
#define CAMERA_NUMBER 4    // Cameras number
#define CORNER_HOR 7             // Number of chessboard corners in horizontal direction
#define CORNER_VER 7             // Number of chessboard corners in vertical directions.

vector<Camera*> camera(CAMERA_NUMBER);         // Camera objects
extern vector<Mat*> src_img(CAMERA_NUMBER);    // Calibration frames from cameras
float sf = 8;       // Scale factor
int poster_width = 1200;   // Poster width
int poster_height = 800;   // Poster height

int camera_class_example() {
    for (uint i = 0; i < CAMERA_NUMBER; i++) {
        string camera_model = "calib_results_" + lexical_cast<string>(i + 1) + ".txt";
       // Camera model
        string template_points = "template_" + lexical_cast<string>(i + 1) + ".txt";
       // Template points
        string chessboard = "chessboard_" + lexical_cast<string>(i + 1) + "/";         //
Chessboard image

        Creator creator;
        camera[i] = creator.create(camera_model.c_str(),sf, i);    // Create Camera object
```

```
        if(camera[i] == NULL)                           // Check if camera object was created
            return(-1);

        // Set roi in which contours will be searched (in % of image height)
        camera[i]->setRoi(60);
        // Set empiric bound for minimal allowed perimeter for contours
        camera[i]->setContourMinSize(200);
        // Set template size and reference points
        if(camera[i]->setTemplate(template_points.c_str(), Size2d(poster _width, poster
_height)) != 0)
            return(-1);
        // Calculate intrinsic camera parameters using chessboard image
        if(camera[i]->setIntrinsic(chessboard.c_str(), (char*)"MIPI", 1, Size(CORNER_HOR,
CORNER_VER)) != 0)
            return(-1);
        //Estimate extrinsic camera parameters using calibrating template
        if(camera[i]->setExtrinsic(src_img[i]) != 0)
            return(-1);
    }
    return(0);
}
```

6.3. Grid class

The grid class generates a 3D grid of vertices/texels which are used to map the image from the camera into a 3D bowl using OpenGL. The grid has a bowl shape with a flat circle base and a parabolic bowl edge. The center of the flat circle base is in the center of the template.

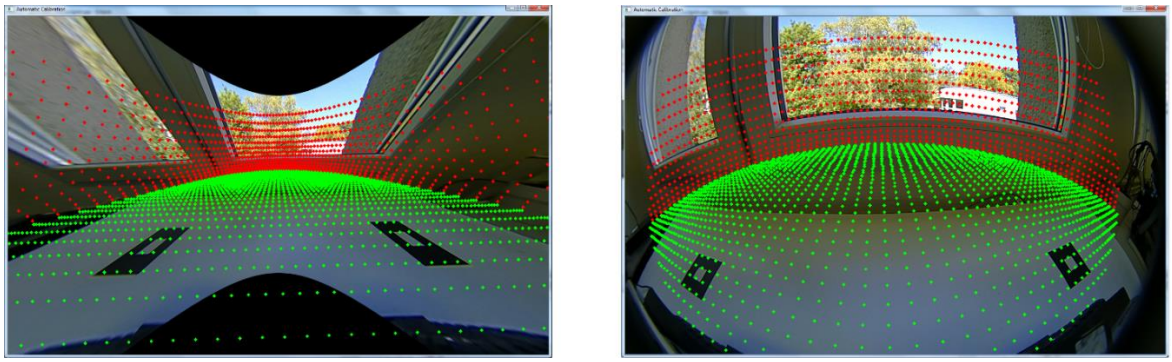

Figure 19. 3D grid definition

The procedure of a 3D grid definition is as follows:

1. Set the grid parameters.
2. Define the 3D grid points relatively to the center of the calibration poster.
3. Project the 3D points to an image plane.
4. Generate triangles from the 3D grid and save them to a file. The file is used for texture mapping.

There are two variants of grid definition for the flat bowl bottom with the same API:

1. The CurvilinearGrid class—the grid is more dense in the center of the bowl bottom and more sparse at the bowl bottom edge.
2. The RectilinearGrid class—the grid is sparse in the middle of the bowl bottom and more dense at the bowl bottom edge.

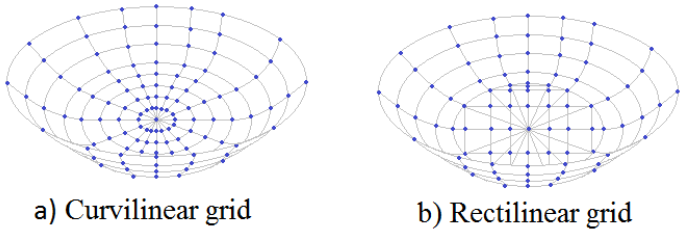

Figure 20. Two variants of grid definition

Private attributes (inaccessible from outside the class):

- GridParam parameters—the parameters of the grid:
  - uint angles—every quadrant of the circle is divided into this number of arcs (see Section 4.1, “System calibration settings”).
  - **uint** start_angle—the parameter sets the circle segment for which the grid is generated (see Section 4.1, “System calibration settings”).
  - **uint** nop_z—the number of points in the z-axis (see Section 4.1, “System calibration settings”).
  - **double** step_x—the step in the x-axis which is used to define the grid points in the z-axis (see Section 4.1, “System calibration settings”).

- vector<Point3f> seam—the points’ array for the seam definition. Eight edge points are defined for each grid.

  Points 1-4 describe the left edge of the grid (they are located in the second quadrant).
  - Point 1 is located on the flat circle base (z = 0). It is the leftmost point with the minimum value of the y coordinate.
  - Point 2 is located on the flat circle base (z = 0). It is the leftmost point of the grid and lies on the base circle edge;
  - Point 3 is located on the bowl edge. It is the last point in the first grid column with (z! = 0).
  - Point 4 is located on the bowl edge. It is the leftmost point with the maximum value of the z-coordinate.

  Points 5-8 describe the right edge of the grid (they are located in the first quadrant).
  - Point 5 is located on the bowl edge. It is the rightmost point with the maximum value of the z-coordinate.
  - Point 6 is located on the bowl edge. It is the last point in the last grid column with (z! = 0).
  - Point 7 is located on the flat circle base (z = 0). It is the rightmost point of the grid and lies on the base circle edge.

– Point 8 is located on the flat circle base (z = 0). It is the rightmost point with the minimum value of the y-coordinate.

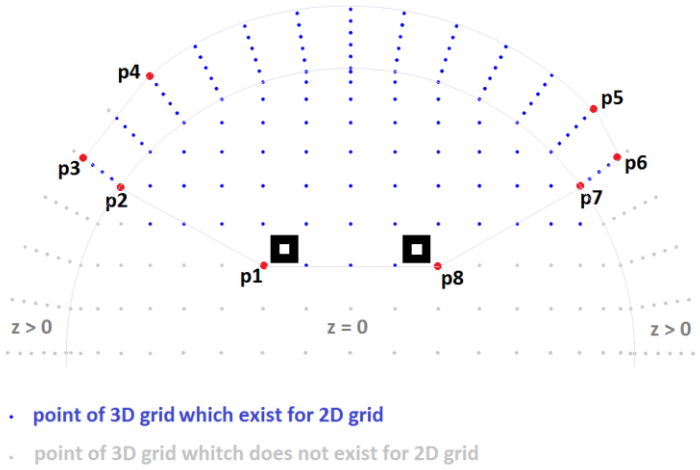

Figure 21. Seams' definition

Public methods (accessible from outside the class):

- Create the Grid object.

```
/*************************************************************************
* @brief        Grid class constructor.
* @param  in    uint angles—every quadrant of the circle is divided into this number of arcs.
*               uint start_angle—it is not necessary to define the grid through a whole
*               semi-circle.
*                    The start_angle sets the circular sector for which the grid is generated.
*               uint nop_z—the number of points in the z-axis.
*               double step_x—the grid step in the polar coordinate system.
*                    The value is also used to define the grid points in the z-axis.
*                    The step in the z-axis: step_z[i] = (i * step_x)^2, i = 1, 2, ... - the number
*                    of points.
* @return       The function creates the Grid object.
* @remarks      The function sets the main properties of a new Grid object.
*************************************************************************/
Grid(uint angles, uint start_angle, uint nop_z, double step_x);
```

- Get the seam points.

```
/*************************************************************************
* @brief        Copy the points from the seam property into the S vector.
* @param  out   vector<Point3f> &S—the pointer to the output vector of the seam points.
* @return       —
* @remarks      The function copies the seam points into the input vector S. The vector size
*               must be 8. If the vector size is not 8, it is better to redefine the camera object
```

```
*                   because some problems occurred.
*******************************************************************************/
void getSeamPoints(vector<Point3f> &S);
```

- Create the 3D grid and save it to a file for frame mapping.

```
/*******************************************************************************
* @brief          Generate a 3D grid of texels/vertices for the input Camera object.
* @param  in      Mat &img—the image captured by the camera (it is needed only to draw a grid).
*         in      Camera* camera—the pointer to the Camera* object.
*         in      double radius—the radius of the base circle.
*                 The radius must be defined in pixels and must be bigger than a half
*                 of the template diagonal.
* @return         —
* @remarks        The function defines the 3D grid, generates triangles from it, and saves the
*                 triangles into a file.
*******************************************************************************/
void createGrid(Mat &img, Camera *camera, double radius);
```

Example 2. Grid object creation

```
#define CAMERA_NUMBER      4      // Number of cameras
#define ANGLES             40     // Number of grid angles
#define START_ANGLE               5      // Number of the first grid angle
#define STEP_X                    0.1    // Step in x axis
#define RADIUS             2.3    // Radius scale

extern vector<Camera*> camera(CAMERA_NUMBER);  // Camera objects
extern vector<Mat*> src_img(CAMERA_NUMBER);    // Calibration frames from cameras

int grid_class_example() {
    int nopz = 10;
    for (uint i = 0; i < src_img.size(); i++) {       // Get number of points in z axis
        int tmp = camera[i]->getBowlHeight(RADIUS * camera[i]->getBaseRadius(), STEP_X);
        nopz = MIN(nopz, tmp);
    }
    vector< vector<Point3f> > seam;
    for (uint i = 0; i < src_img.size(); i++) {
        // Create Grid object, set grid parameters
        CurvilinearGrid grid(ANGLES, START_ANGLE, nopz, STEP_X );
        // Calculate grid points and save grid to the file
        grid.createGrid(src_img[i], camera[i], RADIUS * camera[i]->getBaseRadius());
        vector<Point3f> seam_points;
        grid.getSeamPoints(seam_points);       // Get grid seams
        seam.push_back(seam_points);
    }
    return(0);
}
```

6.4. Masks class

The Masks object prepares the masks' images for the Camera objects. These masks are used to stitch the frames from the cameras with the overlaps being as invisible as possible. They must be defined for the original captured image from the camera (with fisheye distortion), because the same transformation is going to be applied to the camera frames and masks in the real-time rendering application.

The procedure of the mask calculation is as follows:

- Get the edges of the grids. Eight edge points must be defined for each grid (see Section 6.3, "Grid class").
- Calculate the seams for every two adjacent grids.

The seam of two adjacent grids is a line $y = a * x + b$, where the a and b coefficients are found from the grid intersection:

- Find the intersection point of the grids' bottom borders (line between points p1 and p8)—s11.
- Find the center of the line between point p2 from one grid and point p7 from another grid—s12.
- Find the seam line coefficients a and b.

The seam points are defined for the 3D template and then projected to the image plane using the "projectPoints" function from the OpenCV library. To use the "projectPoints" function, you must know the extrinsic and intrinsic camera parameters (camera matrix, distortion coefficients, rotation, and translation vectors).

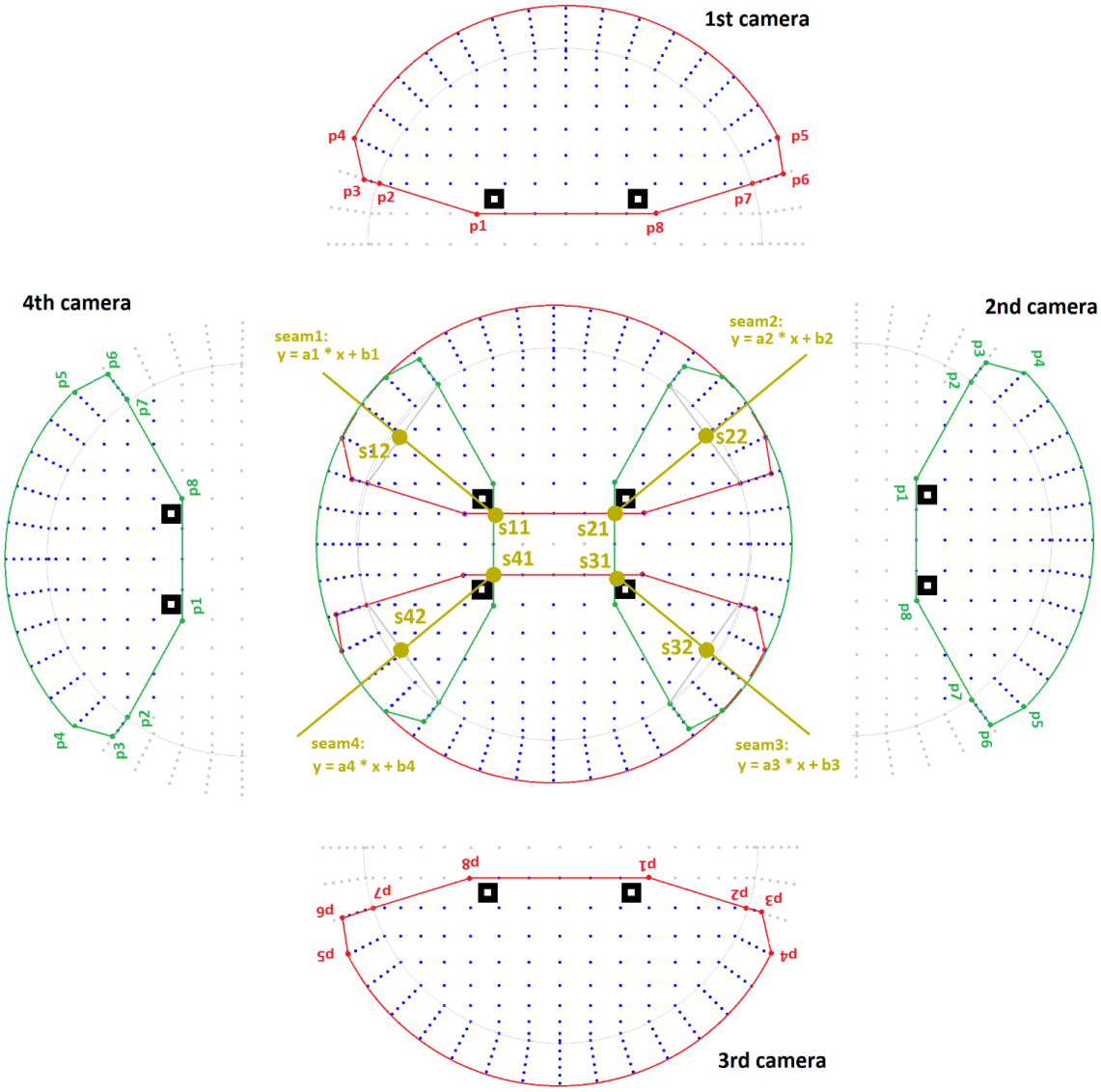

Figure 22. Masks' definition

- Create the masks which are limited to the seam points.

  The mask for each camera is defined for the 3D template. It is limited to the grid seams. The seam points are projected from a 3D space into a 2D image. All 2D seam points describe the convex polygon which defines the mask in a 2D image. The polygon is filled with white color and the background is filled with black color.

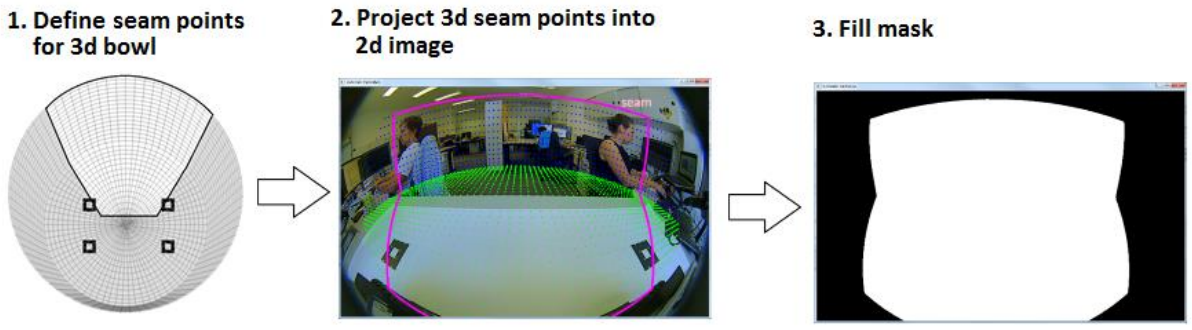

Figure 23. Mask creating flow

- Smooth the mask edges.

  To obtain a seamless blending of frames, the left and right edges of the masks are smoothed. The angle of smoothing defines the area in which the mask edge is going to be smooth. The original seam divides the smoothing angle into two angles with equal measures (angle bisector). The angle-based smoothing is applied only for a flat base. The seam at the bowl side is smoothed with a constant width of smoothing. See also Figure 12 for the smoothing area definition.

Private attributes (inaccessible from outside the class):

- vector<Mat> masks—the vector of masks (one mask for each camera).
- vector<Seam> seaml—the description of the left seam. It contains the coordinates of two points lying on the seam and the a and b coefficients of the seam (y = a*x + b).

```
struct Line {        // Line description y = alpha * x + beta
       double alpha;
       double beta;
};
struct Seam {        // Seam definition
       Line line;    // Seam coefficients y = alpha * x + beta
       Point2f p1;   // Seam first point
       Point2f p2;   // Seam second point
};
```

- vector<Seam> seamr—the description of the right seam. It contains the coordinates of two points lying on the seam and the a and b coefficients of the seam (y = a*x + b).

Public methods (accessible from outside the class):

- Get the seam.

```
/*************************************************************************************
* @brief          Get the seam.
* @param  out     Seam* left—the left seam.
*                 Seam* right—the right seam.
*          in     uint i—the number of the seam elements.
* @return         The function returns 1 if the vector<Seam> seaml and
*                 vector<Seam> seamr sizes are larger than i. Otherwise, it returns 0.
* @remarks        The function copies the i elements from the seaml and seamr vectors (left and
*                 right seam) to the left and right pointers.
*************************************************************************************/
```

int getSeam(Seam* left, Seam* right, uint i);

- Calculate the masks.

```
/*************************************************************************************
* @brief        Calculate the masks for SV3D.
* @param in     vector<Camera*> cameras—the vector of the Camera objects.
*               vector< vector<Point3f> > &seam_points—the pointer to the vector containing
*                    the seam points for all grids.
*                    The grid edge consists of eight points. Points 1-4 describe the left edge of
*                    the grid (they are in the second quadrant). Points 5-8 describe the right
*                    edge of the grid (they are in the first quadrant).
*                    - The first point is located on the flat circle base (z = 0). It is the leftmost
*                      point with the minimum value of the y-coordinate.
*                    - The second point is located on the flat circle base (z = 0). It is the
*                      leftmost point of the grid which lies on the base circle edge.
*                    - The third point is located on the bowl edge. It is the last point in the first
*                      grid column with a value of (z != 0).
*                    - The fourth point is located on the bowl edge. It is the leftmost point with
*                      the maximum value of the z-coordinate.
*                    - The fifth point is located on the bowl edge. It is the rightmost point with
*                      the maximum value of the z-coordinate.
*                    - The sixth point is located on the bowl edge. It is the last point in the last
*                      grid column with a value of (z != 0).
*                    - The seventh point is located on the flat circle base (z = 0). It is the
*                      rightmost point of the grid which lies on the base circle edge.
*                    - The eighth point is located on the flat circle base (z = 0). It is the
*                      rightmost point with the minimum value of the y-coordinate.
*               vector<Mat> &img—the seam is drawn on the image.
*               float smothing—the smothing angle value.
* @return       —
* @remarks      The function calculates the masks for SV3D. The masks are used for texture
*                mapping. They must be defined for the original captured image from the camera
*               (with fisheye distortion) because the same transformation is applied on
*               the camera frames and masks.
*               The procedure of mask calculation is as follows:
*               —    Calculate the seams for every two adjacent grids.
*                    The seam of two adjacent grids is a line y = a * x + b.
*                    The a and b coefficients are found from the grid intersection.
*                    The seam points are defined for a 3D template and then
*                    projected to an image plane using the projecPoints function from
*                    the OpenCV library. To use the projectPoints function, it is necessary to
*                    know the extrinsic and intrinsic camera parameters: camera matrix,
*                    distortion coefficients, and rotation and translation vectors.
```

```
*                   —    Create masks which are limited to the seams. All 2D seam points describe
*                        a convex polygon which defines the mask in a 2D image. The polygon is
*                        filled with white color and the background is filled with black color.
*                   —    Smooth mask edges.
*********************************************************************************/
void createMasks(vector<Camera*> &cameras, vector< vector<Point3f> > &seam_points,
vector<Mat> &img, float smothing);
```

- Split vertices/texels grid.

```
/*********************************************************************************
* @brief          Split the vertices/texels grid into a grid which is rendered with blending and
*                 a grid which is rendered without blending for all cameras.
* @param          —
* @return         The function returns 0 if all grid files in the “arrayX” file (X = 1,2,3,4 is camera
*                 number) exist. If one of the files is not found, the function returns -1.
* @remarks        The function reads the grid of texels/vertices from the “arrayX” file (X = 1,2,3,4
*                 is the camera number) and produces two output grids: first one for the
*                 overlapping regions (“arrayX1”) and the second one for the non-overlapping
*                 regions (“arrayX2”). A camera blending mask is used to split the grid into two
*                 parts. The mask value is checked for each texel of a rendered triangle.
*                 If at least one texel is masked with a value lower than 255, the triangle is written
*                 to the overlap grid. Otherwise, the triangle is written to the non-overlap grid.
*********************************************************************************/
int splitGrids();
```

Example 3. Grid object creation

```
#define SMOOTH_ANGLE       0.15   // Defines the area in which mask edge will be smooth
#define CAMERA_NUMBER      4      // Cameras number

extern vector<Camera*> camera(CAMERA_NUMBER);  // Camera objects
extern vector<Mat*> src_img(CAMERA_NUMBER);    // Calibration frames from cameras
extern vector<Point3f> seam_points;                  // Vector of seam points

int masks_class_example() {
    Masks masks;
    masks.createMasks(camera, seam, src_img, SMOOTH_ANGLE);
    if(masks.splitGrids() != 0) // Split vertices/texels grid
        return(-1);
    return(0);
}
```

6.5. Compensator class

The exposure correction process is done in the texture-mapping application and includes two parts (these functions are not part of the Compensator class):

- Computation of the exposure correction coefficients.
- Applying exposure correction to the input frames.

The exposure correction coefficients for the current camera are computed as the sum of the pixel values in the overlapping area between the current camera frame and the previous camera frame in the linear RGB color space. Each camera has two overlap regions: one with the previous camera and the other with the next camera.

The Compensator class generates the grids only for the overlapping regions that lay on the flat bowl bottom for each camera and saves the grids into the *compensator* folder (one grid per camera). Each grid contains a description of two overlapping regions (left and right).

The application also saves the file with the coordinates of the circumscribed rectangle for each overlapping region. This information is used to copy only the overlapping regions from the frame buffer when the exposure correction coefficients are calculated.

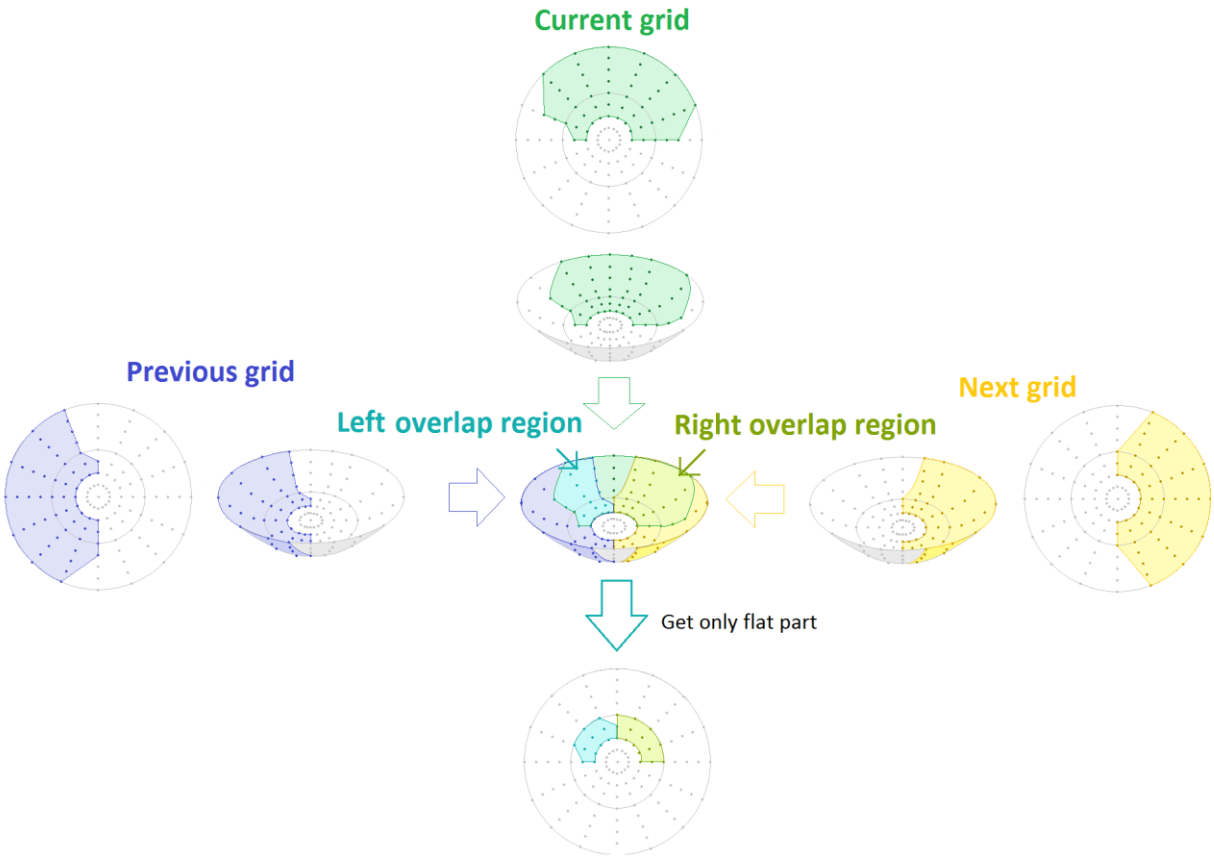

Figure 24. Overlapping regions' definition

Private attributes (inaccessible from outside the class):

- CompensatorInfo cinf—a structure with the compensator parameters. All parameters are calculated when the compensator is created and fed.

```
struct CompensatorInfo {
vector<Rect2f> roi;  // Vector of rectangles which circumscribe overlap regions
 double radius;             // Radius of bowl flat base
 Mat mask;           // Mask of all overlap regions
};
```

Public methods (accessible from outside the class):

- Create the Compensator object.

```
/*************************************************************************************
* @brief          The Compensator class constructor.
* @param  in      Size mask_size—the size of the compensator mask (screen size).
* @return         The function creates the Compensator object.
* @remarks        The function mask property of a new Compensator object.
*************************************************************************************/
```

Compensator(Size mask_size);

- Get the coordinates of the overlapping region rectangle reflection across the x-axis.

```
/*************************************************************************************
* @brief          Get the coordinates of the overlap region rectangle reflection across the x-axis.
* @param  in      uint index—the rectangle index in the vector<Rect2f> ROI.
* @return         Rect—the rectangle reflection across the x-axis.
* @remarks        The function reflects the rectangle across the x-axis. This action is needed
*                 due to the inverse window coordinates' definition in OpenGL.
*************************************************************************************/
```

Rect **getFlipROI**(uint index);

- Calculate the compensator.

```
/*************************************************************************************
* @brief          Fill the CompensatorInfo private property of the Compensator object.
* @param  in      vector<Camera*> cameras—the vector of the Camera objects.
*                 vector< vector<Point3f> > &seam_points—the pointer to the vector containing
*                        the seam points for all grids (cameras).
*                        The grid edge consists of eight points. Points 1-4 describe the left edge of
*                        the grid (they are in the second quadrant). Points 5-8 describe the
*                        right edge of the grid (they are in the first quadrant).
*                        - The first point is located on the flat circle base (z = 0). It is the leftmost
*                          point with the minimum value of the y-coordinate;
*                        - The point is located on the flat circle base (z = 0). It is the leftmost point
*                          of the grid which lies on the base circle edge.
*                        - The third point is located on the bowl edge. It is the last point in the first
*                          grid column (z != 0).
*                        - The fourth point is located on the bowl edge. It is the leftmost point with
```

```
*                       the maximum value of the z-coordinate.
*                     - The fifth point is located on the bowl edge. It is the rightmost point with
*                       the maximum value of the z-coordinate.
*                     - The sixth point is located on the bowl edge. It is the last point in the last
*                       grid column with (z != 0).
*                     - The seventh point is located on the flat circle base (z = 0). It is the
*                       rightmost point of the grid which lies on the base circle edge.
*                     - The eighth point is located on the flat circle base (z = 0). It is the
*                       rightmost point with the minimum value of the y-coordinate.
* @return       —
* @remarks      The function fills the CompensatorInfo property of the Compensator.
*               It calculates the mask which defines four overlap regions and circumscribed
*               rectangles of each region.
*********************************************************************************/
void feed(vector<Camera*> &cameras, vector< vector<Point3f> > &seam_points);
```

- Save the compensator info.

```
/*********************************************************************************
* @brief        Save the compensator info.
* @param  in    char* path—path name.
* @return       Returns 0 if the output file is created successfully. Otherwise, it returns -1.
* @remarks      The function generates the grids only for the overlap regions which lay on the flat
*               bowl bottom for each camera and saves the grids into the compensator folder (one
*               grid per camera). Each grid contains a description of two overlap regions: left and
*               right. The application also saves a file with the texel coordinates of the
*               circumscribed rectangle for each overlap region. The texture mapping
*               application uses this information to copy only the overlap regions from the frame
*               buffer when it calculates the exposure correction coefficients.
*********************************************************************************/
int save(char* path);
```

- Load the compensator info.

```
/*********************************************************************************
* @brief        Load the compensator info.
* @param  in    char* path—the path name.
* @return       Returns 0 if the compensator loads successfully. Otherwise, it returns -1.
* @remarks      The function loads the texel coordinates of the circumscribed rectangle for each
*               overlap region. The texture mapping application uses this information to copy
*               only the overlap regions from the frame buffer when it calculates the exposure
*               correction coefficients.
*********************************************************************************/
int load(char* path);
```

Example 4. Compensator object creation

```
#define DISPLAY_HEIGHT     1080
#define DISPLAY_WIDTH      1920
#define CAMERA_NUMBER      4

extern vector<Camera*> camera(CAMERA_NUMBER);  // Camera objects
extern vector<Point3f> seam_points;                  // Vector of seam points

int compensator_feed_example() {
    Compensator compensator(Size(DISPLAY_WIDTH, DISPLAY_HEIGHT));
    compensator.feed(camera, seam);
    if(compensator.save((char*)"./compensator") !=0)
        return(-1);
    return(0);
}

#define NEXT(x, max)  ((x < max) ? (x + 1) : (0)) // Next index
Mat overlap_roi[CAMERA_NUMBER][2]; // Left and right rois for each camera

int compensator_load_example() {
    Compensator compensator = new Compensator(Size(DISPLAY_WIDTH, DISPLAY_HEIGHT));
    if(compensator->load((char*)"./compensator")!=0)
        return(-1);
    for (int j = 0; j < CAMERAS_NUM; j++)
    {
        overlap_roi[j][0] = Mat(compensator->getFlipROI(j).height, compensator-
>getFlipROI(j).width, CV_8UC(4));
        int next = NEXT(j, CAMERAS_NUM - 1);
        overlap_roi[j][1] = Mat(compensator->getFlipROI(next).height, compensator-
>getFlipROI(next).width, CV_8UC(4));
    }
}

int compensator_using_example() {
    …
    glReadPixels(   compensator->getFlipROI(index).x, compensator->getFlipROI(index).y,
              compensator->getFlipROI(index).width, compensator->getFlipROI(index).height,
              GL_RGBA,
              GL_UNSIGNED_BYTE,
              overlap_roi[index][0].data);
    int next = NEXT(index, camera_num - 1);
    glReadPixels(   compensator->getFlipROI(next).x, compensator->getFlipROI(next).y,
              compensator->getFlipROI(next).width, compensator->getFlipROI(next).height,
              GL_RGBA,
              GL_UNSIGNED_BYTE,
              overlap_roi[index][1].data);
    …
}
```

7. Real-time rendering application

The real-time rendering application uses the OpenCV and Assimp libraries to load the rendering data and the OpenGL API to render the surround view. It is the final part of the whole Surround View system.

The application renders the camera frames (default real-time mode) on a prepared 3D mesh and blends them. Alternatively, it can also render static images or video files (demo mode). The rendering application stitches four input images into a single output and displays it using the GPU. Finally, the simple car model is stitched into the center of the scene. For the application controlling, see the *Surround View User's Guide* [1].

Table 6. Real-time rendering application dependency

Application inputs	/App/Build/arrayN	
	/App/Build/maskN.jpg	
	/App/Build/compensator/	
	/App/Content/models/ferrari.dae	
	/App/Content/font.png	
	Real-time camera frames 1)	/App/Content/camera_inputs/src_X.* 2)
Application settings	/App/Content/settings.xml	
Binary file name	/App/Build/SV3D-1.1	
Target platform	i.MX8 family device, i.MX6QP	
Target OS	Yocto Linux OS	

1. Default real-time mode
2. Optional static demo mode (jpg, avi, or raw video files)

NOTE

Only the real-time rendering application runs continuously. The other Surround View system applications (capturing, lens and system calibrations) are one-shot applications.

8. References

1. *Surround View Application User's Guide* (document SVAUG)
2. "i.MX6 / i.MX7 / i.MX 8 Series Software and Development Tool Resources" available at www.nxp.com
3. "Surround View Application", Reference Design Project page available at www.nxp.com
4. Davide Scaramuzza, "OCamCalib: Omnidirectional Camera Calibration Toolbox for Matlab" available at https://sites.google.com/site/scarabotix/ocamcalib-toolbox
5. Matlab Runtime R2017a, available at www.mathworks.com
6. OCamCalib", Omnidirectional Camera Calibration Toolbox for Matlab Runtime" available at https://source.codeaurora.org/external/imx

9. Revision history

Table 7 summarizes the changes done to this document since the initial release.

Table 7. Revision history

Revision number	Date	Substantive changes
0	03/2018	Initial release.
1	07/2018	Added Matlab Runtime for OCamCalib.

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number: SVARM
Rev. 1
07/2018