On Calibrating a Projector-Camera Rig to Enable Spatial Augmented Reality Use Cases

Michael Hornacek
TU Vienna (IMW/CPS)

May 2021

1 Motivation

Spatial augmented reality (SAR) is a branch of augmented reality whereby the augmentation of the scene is carried out using a projector. While pointing a projector at a scene and projecting an image is on its own not a challenge, doing so in a manner such that the projected content **appear to be naturally part of the scene** is more tricky. The intuition for the general problem we face can be drawn from using a projector to project an image to a flat wall: unless the projector faces the wall frontally, the bounds of a projected rectangular image will not appear rectangular, but more generally as a trapezoid (i.e., the image will appear distorted). The cause for this effect is that the geometry of the scene as it relates to the geometry of the projector plays a role in *how the pixels of the projector's image plane map to points in the scene*.

The central **objective** we set for ourselves is to **enable correcting for distortion** caused by projecting to arbitrary scene geometry in an **automatic** fashion. This is to be done in a manner that lends itself to practical application in a real-world setting where the **scene** to be augmented—e.g., an **object** being gradually assembled and its **placement** relative to the projector—**does not remain static**. Moreover, the object in question may but may not in its entirety fall within the projector's **field of view (FOV)**. Finally, the solution is ideally to rely on the existing **hardware setup**—i.e., projector with steerable mirror system and stereo camera—mounted in the Pilotfabrik[1] of TU Vienna.

2 Approach

The problem of correcting for distortions as outlined in Section 1 requires knowledge of the manner in which the respective rays through the pixels of the projector's image plane fan out into the scene, and the geometry of the scene itself within at least the projector's field

[1] https://www.pilotfabrik.at/?lang=en

of view (determined in part by the steerable mirror),[2] as illustrated in Figure 1. This is because the scene point 'illuminated' by a pixel in the projector's image plane is given by intersecting its corresponding ray with the geometry of the scene surface. To model this interaction calls for two components in particular: (i) one-time **projector calibration** and (ii) **recovery of scene geometry** whenever the scene undergoes change relative to the field of view of the projector.

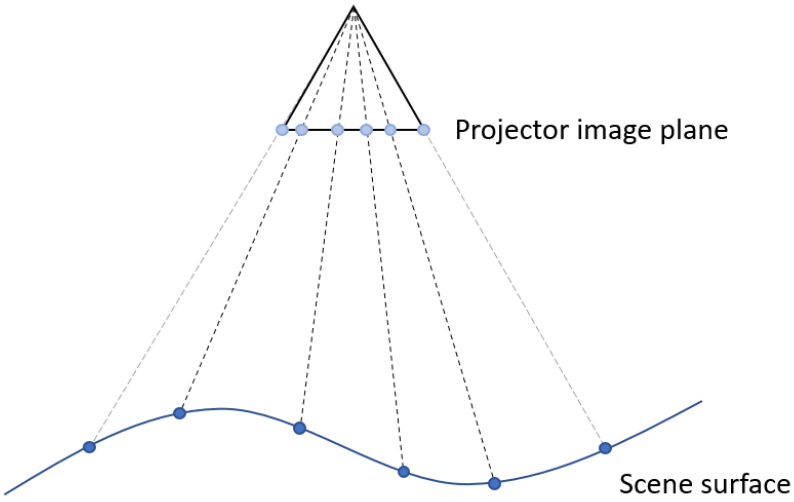


Figure 1: Projector-scene geometry. The way rays 'fan out' through the pixels of the projector's image plane is given by calibrating the projector (note that the projector's field of view is the portion of the scene in between the gray rays). The scene point 'illuminated' by a pixel in the projector's image plane is given by intersecting its corresponding ray with the geometry of the scene surface. Projector calibration and recovery of scene geometry are accordingly prerequisites of the image warping that it is our ambition to carry out.

The task of finally *warping* imagery is left outside the scope of this document, as a subject to be addressed only after the topics of projector calibration and recovery of scene geometry have been nailed, since these two topics are prerequisites of such image warping. The remainder of this section is organized in accordance with these two components: one-time projector calibration, and recovery of scene geometry.

2.1 One-time Projector Calibration

The 'output' of projector calibration[3] is a mathematical model that determines, for each pixel $\mathbf{x}_{proj}$ in the projector's image plane π_{proj}, the ray from the projector's center of projection $\mathbf{C}_{proj}$ through $\mathbf{x}_{proj}$, along which the pixel is projected *to the scene*. Inversely, the model also determines the projection $\mathbf{x}'_{proj}$ of any point $\mathbf{X}'_{proj}$ in the scene *to the projector's*

[2] Cf. Section 4.1.2 of the Spatial Augmented Reality textbook of Bimber and Raskar, available at http://pages.cs.wisc.edu/~dyer/cs534/papers/SAR.pdf. Note that the authors also mention **user location**; this is relevant for some—but not all—use cases, and we shall leave it too outside the scope of this document.
[3] Cf. again Section 4.1.2 of Bimber and Raskar.

image plane π_{proj}, obtained by intersecting π_{proj} with the ray from $\mathbf{C}_{proj}$ through $\mathbf{X'}_{proj}$. Such a model is said to function in accordance with the principle of **central projection**.

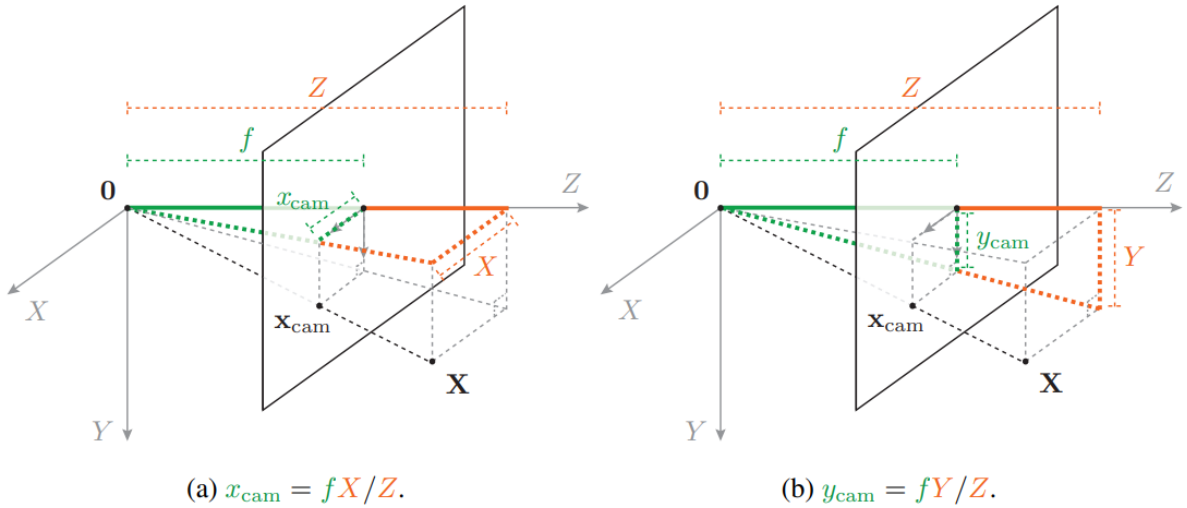


Figure 2: Pinhole camera model. The image plane π is perpendicular to the Z axis of the camera's coordinate frame; its location is given by the focal length f, the distance of π along the Z axis. The camera's center of projection **C** is located at the origin **0** of the camera coordinate frame. The projection **x** = (x_{cam}, y_{cam}) of a point **X** = (X, Y, Z) to π, by central projection, follows from the principle of similar triangles. Note that the origin of the coordinate system of the image plane—called the principal point—is situated at the intersection of the Z axis with π. Note that the Z component of a point expressed in the camera's coordinate frame is referred to as the point's depth relative to the camera.

Notably, central projection is the same principle that underlies the modelling of conventional cameras. A camera modeled in accordance with the principle of central projection is called a **pinhole camera**. A pinhole camera—as illustrated in Figure 2—can be expressed in terms of a 3 × 3 'camera calibration' matrix **K**,[4] derived from the camera's focal length and principal point. In practice, a pinhole camera model works well in modelling how the respective rays through the pixels of the image plane fan out into the scene, on the condition that images to which the model is applied have been corrected for **lens distortions** caused by the camera's lens system (cf. Figure 3 for an example). The 'output' of camera calibration is accordingly typically not only the matrix **K**, but the associated **lens distortions model coefficients** as well. The lens distortion model coefficients enable applying a lens distortion model to **undistort** any image acquired by the camera.

In practice, calibrating a *camera* relies on (i) establishing **2D-3D correspondences** between pixels in the camera's image plane and corresponding points in the scene, and on (ii) using those correspondences as input to an **optimization procedure** that outputs the calibration matrix **K** and the associated distortion model coefficients. Calibration of a *projector* can be carried out in precisely the same manner insofar as step (ii) is concerned; *the major*

[4] If you'll forgive a brazen bit of self-indulgence, have a look at Chapter 2 of my dissertation at https://publik.tuwien.ac.at/files/PubDat_243668.pdf for a more detailed attempt at a self-contained but accessible introduction to the geometry pinhole cameras, the derivation of the camera calibration matrix **K**, and other topics of relevance to this document (including epipolar geometry).

difference in projector calibration relative to camera calibration concerns the manner in which 2D-3D correspondences are identified, i.e., between pixels in the image plane of the projector and the corresponding points in the scene.

What remains of this section first, for context, reviews how 2D-3D correspondences are conventionally obtained between camera and scene. Next, we address how 2D-3D correspondences are conventionally obtained between projector and scene. Finally, we outline the manner in which 2D-3D correspondences are used to compute the calibration matrix **K** and associated lens distortion model coefficients. Note that these steps constitute fundamental tasks in computer vision and can be carried out largely using functionality available in OpenCV.

2.1.1 Obtaining Camera-Scene 2D-3D Correspondences

The recovery of 2D-3D correspondences in support of camera calibration is carried out, conventionally, by relying on a planar **calibration surface** of known **scale** to automatically identify correspondences between the 3D points on the calibration surface and their 2D correspondences in the image plane. A widely used example of such a calibration object is a **chessboard pattern**.[5] The 3D **corner points** of the chessboard are obtained analytically in a coordinate system defined in the plane of the chessboard, requiring knowledge only of the length of a side of a chessboard square.[6] Next, using the `findChessboardCorners()` function in OpenCV[7] the corresponding 2D pixels of the corner points in any single image of the chessboard are obtained as a list of lists (one list per input image); an example is given in Figure 3.

In practice, multiple images of the chessboard are acquired, from a set of **widely varying viewpoints**.[8] The rationale behind doing so is to capture the varying effects of lens distortion across the full extent of the camera's field of view; notably, images of the chessboard acquired only at approximately the middle of the image plane would fail to provide information about radial distortions present at the image plane's boundaries.

[5] Cf. https://markhedleyjones.com/projects/calibration-checkerboard-collection for a variety of chessboard calibration patterns ready for use.
[6] E.g., (0,0,0), (1.5,0,0), (3,0,0), …, (9,7.5,0) for a chessboard with 7 × 6 corners, with each square of length and width of 1.5 unit, respectively.
[7] See https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html for a comprehensive tutorial.
[8] It is good practice to acquire images of the calibration surface across all corners and edges of the image plane alongside an image or two from around the middle of the image plane. Beware: for any given viewpoint, the *entirety* of the chessboard must be visible if relying on OpenCV's `findChessboardCorners()`. Otherwise, the function will fail to find any corners in the image acquired from that viewpoint.

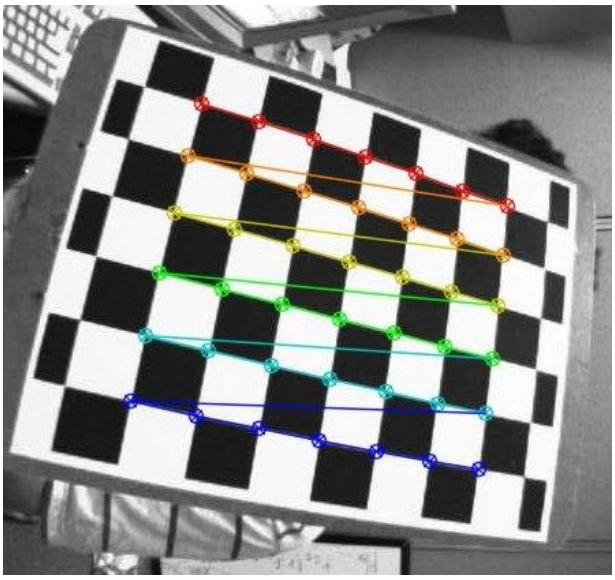


Figure 3: 2D chessboard corners recovered automatically using OpenCV's `findChessboardCorners()` function, color coded using the `drawChessboardCorners()` function. The corresponding 3D positions can be computed analytically, given prior knowledge of the length of a square of the chessboard. Note the distortions caused by the camera's lens system, clearly visible in the image: the edges of the chessboard should be straight lines, but instead appear to bend outwards.

2.1.2 Obtaining Projector-Scene 2D-3D Correspondences

Todo: describe how we are now obtaining 3D points via intersection with ground plane of back-projections of detected circles… Whether we want to project to a ground plane or to arbitrary scene geometry, we will calibrate our projector by projecting circles to the floor. Besides recovering the intrinsics of the projector, for each of the locations on the floor to which we project and thereby for precisely the respective projector look directions, we will obtain the extrinsics of the projector. These will be the projector look directions for which we will be able to carry out spatial AR.

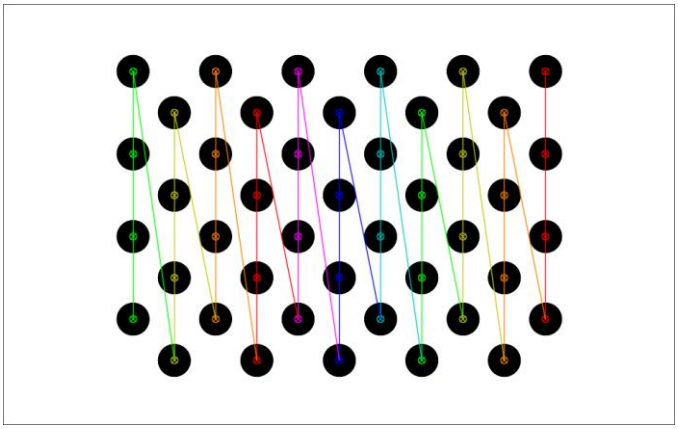


Figure 4: todo... Projected circle pattern, with 2D circle centers obtained using again the OpenCV function `findChessboardCorners()` function, color coded using the `drawChessboardCorners()` function. The corresponding 3D locations on the ground for any image are obtained by computing the intersection of the ground plane with the back-projections of the detected circle centers …

2.1.3 Computing the Calibration

Given a set of 2D-3D correspondences in the form of a list of lists[9] of 2D pixel positions ('image points') and a corresponding list of 3D points ('object points'), the OpenCV[10] function `calibrateCamera()` can be used to carry out the actual calibration. The function returns the camera calibration matrix **K** and the lens distortion model coefficients.[11] Additionally, the function returns a list of 3D rigid body transformations whose length equals the number of calibration images used as input; the n^{th} such rigid body transformation gives the pose (**R**, **t**) of the camera or projector relative to the corresponding 3D points[12]. The projection of such a 3D point **X** to the image plane of the camera or projector is given by **K**(**RX** + **t**), and is obtained in a manner such that, collectively over all 2D-3D correspondences, the sum of squared distances in the image plane between the 2D projections of these 3D points and their respective 2D correspondence are minimized.

2.2 Recovery of Scene Geometry

Todo: the discussion of stereo here is relevant to the case where we want to project to arbitrary surface geometry. We will pursue this as out next step (and then texture map the resulting mesh). For our first step (projecting to a flat surface), we get the ground plane using a checkerboard pattern…

Although useful for obtaining 2D-3D correspondences in support of projector calibration (cf. Section 2.1.2), we are in principle not restricted to structured light stereo for obtaining scene geometry in general. Structured light stereo, however, *is* the most attractive of the options that present themselves given the hardware setup in the Pilotfabrik, and is a **gold standard** approach for reconstruction of **indoor poorly textured surfaces** given its favorable quality-to-price tradeoff.

[9] Each list contains a list corresponding to each respective image.
[10] See again https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html for a comprehensive tutorial.
[11] See https://www.learnopencv.com/understanding-lens-distortion/ for more details on how these particular coefficients are interpreted mathematically.
[12] Note that if these points are of a calibration surface placed on the floor, then such a rigid body transformation gives the pose of the projector or camera relative to the '**ground plane**'; the inverse of this rigid body transformation gives the pose of the ground plane relative to the projector or camera.

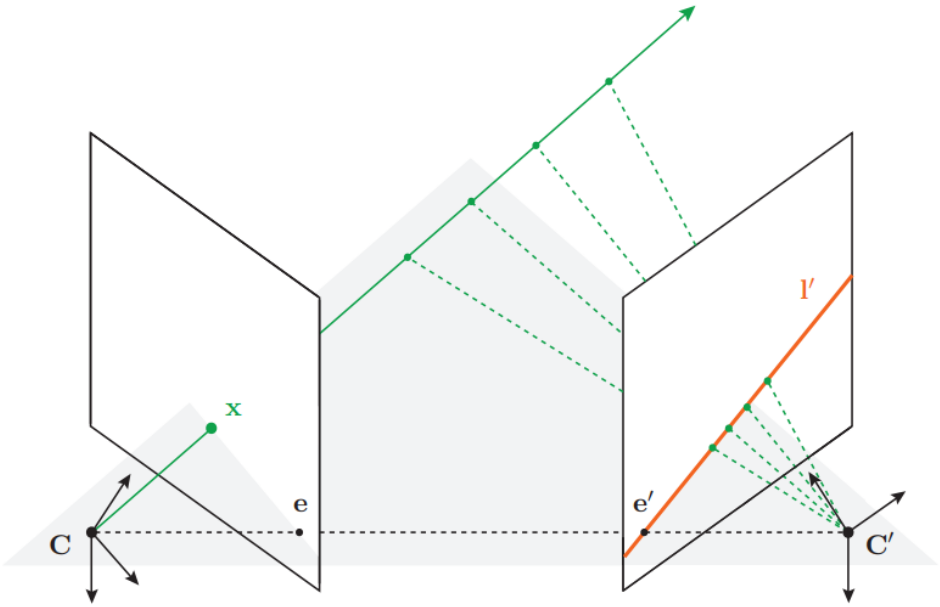


Figure 5: Epipolar geometry (i.e., the geometry of stereo). The 3D point **X** that projects to the pixel **x** in the left camera's image plane must lie along the ray passing through **x** from the left camera's center of projection **C**. Accordingly, the projection **x**' of **X** to the image plane of the *right* camera must lie along the epipolar line **l**' spanned by **x**, obtained by projecting that ray to the right camera's image plane. This restricts the search for **x**' to only the pixels along that line, a constraint common to all stereo approaches. Once **x**' has been identified, the point **X** can be recovered by triangulation.

Fundamental to all variants of stereo is that, given a pixel **x** in the left image, the search for the corresponding pixel **x'** in the right image is restricted to only pixels situated along the **epipolar line** spanned by **x**, as illustrated in Figure 5. *Where stereo approaches differ is primarily in how the 'best' match along an epipolar line is determined.* Classical **correlation-based** stereo approaches rely on minimizing a similarity score over a window about **x** in the left image and the corresponding windows about respective candidate matching pixel in the right image. However, a correlation-based approach can be expected to work well only in the presence of adequately **discriminative texture**.[13] In structured light stereo, matching is carried out with respect to binary codes derived from projecting successive strip patterns, thereby in effect *introducing* discriminative texture into the scene and enabling the reconstruction of poorly textured surfaces. As it relies on illumination introduced to the scene, structured light stereo is termed an **active** sensing approach; in contrast, correlation-based stereo is considered a **passive** sensing approach.

[13] Objects like the white walls of a room *or typical manmade objects* are too poorly textured to enable reliable matching via correlation-based stereo.

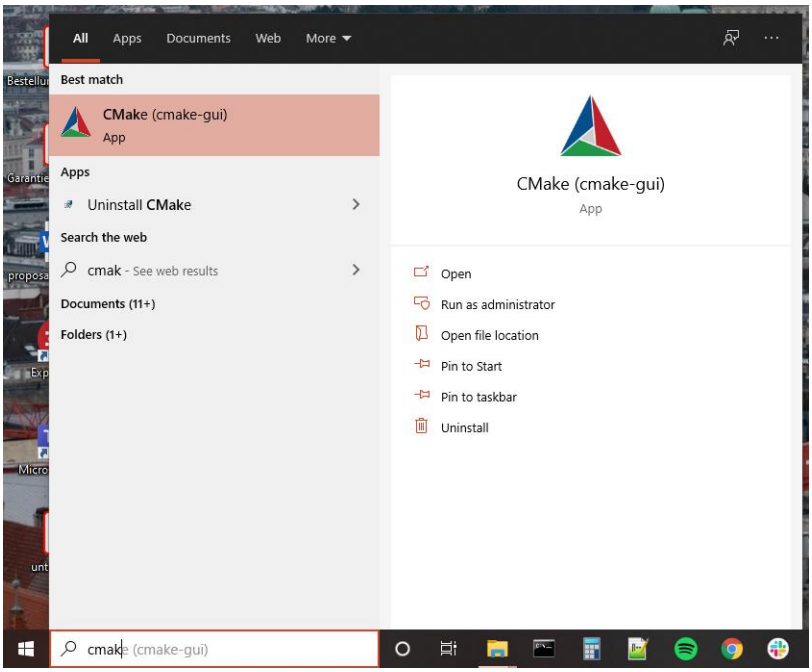
3 Building the Code

Let CPSSPATIALAR_BASEDIR refer to the base directory of your local copy the **cpsspatialar** Git repository[14] (such that `src` and `docs` are subdirectories of CPSSPATIALAR_BASEDIR):

1. Install **Visual Studio Professional 2019**[15]
2. Install **CMake**[16]

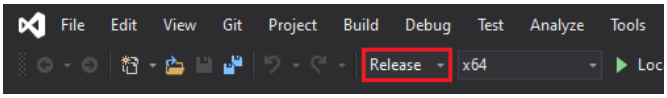
3. Let `EIGEN_BASEDIR` refer to `CPSSPATIALAR_BASEDIR\src\external\eigen`; download version 3.4 of the source code of **Eigen**[17] and extract to `EIGEN_BASEDIR` (such that the `CMakeLists.txt` file is located in `EIGEN_BASEDIR`)

4. Let `OPENCV_BASEDIR` refer to `CPSSPATIALAR_BASEDIR\src\external\opencv`; download version 4.5.1 of the source code of **OpenCV**[18] and extract to `OPENCV_BASEDIR` (such that the `CMakeLists.txt` file is located in `OPENCV_BASEDIR`)
5. Run CMake (cmake-gui):



[14] https://github.com/m-hornacek/cpsspatialar
[15] Available via TU Wien Campus Software at https://www.it.tuwien.ac.at/en/services/software/software-provision/campus-software/.
[16] Via Windows win64-x64 Installer at https://cmake.org/download/.
[17] https://gitlab.com/libeigen/eigen/-/archive/3.4/eigen-3.4.zip
[18] https://github.com/opencv/opencv/archive/4.5.1.zip

6. Enter OPENCV_BASEDIR in the field to the right of 'Where is the source code' and OPENCV_BASEDIR\build in the field to the right of 'Where to build the binaries' (note that backslashes are converted automatically to forward slashes)
7. Click 'Configure' and click 'Yes' if prompted to create the directory OPENCV_BASEDIR
8. If prompted to specify the generator for this project and to optionally indicate the platform for the generator, provide 'Visual Studio 16 2019' and 'x64', respectively, and click 'Finish'
9. Click 'Open Project' or, equivalently, open OPENCV_BASEDIR\build\OpenCV.sln; this opens the Visual Studio solution OpenCV.sln in Visual Studio Professional 2019
10. Select 'Release' from the Solution Configuration pull-down menu and select 'Build:Build Solution' from the menu bar to finally build (i.e., compile) OpenCV in release mode (note that it will take some minutes for building to complete):



11. Let FREEGLUT_BASEDIR refer to CPSSPATIALAR_BASEDIR\src\external\ freeglut; download version 2.8.1 of **freeglut**[19] pre-built binaries to FREEGLUT_BASEDIR (such that the CMakeLists.txt file is located in FREEGLUT_BASEDIR)

12. Let GLEW_BASEDIR refer to CPSSPATIALAR_BASEDIR\src\external\glew; download version 2.1.0 of the source code of **glew**[20] and extract to GLEW_BASEDIR (such that the Makefile file is located in GLEW_BASEDIR)
13. Run CMake (cmake-gui)
14. Enter GLEW_BASEDIR\build\cmake in the field to the right of 'Where is the source code' and GLEW_BASEDIR in the field to the right of 'Where to build the binaries' (note that backslashes are converted automatically to forward slashes)
15. Click 'Configure'
16. If prompted to specify the generator for this project and to optionally indicate the platform for the generator, provide 'Visual Studio 16 2019' and 'x64', respectively, and click 'Finish'
17. Click 'Open Project' or, equivalently, open GLEW_BASEDIR\glew.sln; this opens the Visual Studio solution glew.sln in Visual Studio Professional 2019
18. Select 'Release' from the Solution Configuration pull-down menu and select 'Build:Build Solution' from the menu bar to build glew in release mode

19. Let GLFW_BASEDIR refer to CPSSPATIALAR_BASEDIR\src\external\glfw; download version 3.3.4 of the source code of **glfw**[21] and extract to GLFW_BASEDIR (such that the Makefile file is located in GLFW_BASEDIR)
20. Run CMake (cmake-gui)
21. Enter GLFW_BASEDIR in the field to the right of 'Where is the source code' and GLFW_BASEDIR\build in the field to the right of 'Where to build the binaries' (note that backslashes are converted automatically to forward slashes)
22. Click 'Configure'

[19] https://www.transmissionzero.co.uk/files/software/development/GLUT/older/freeglut-MSVC-2.8.1-1.mp.zip
[20] https://sourceforge.net/projects/glew/files/glew/2.1.0/glew-2.1.0.zip/download
[21] https://github.com/glfw/glfw/releases/download/3.3.4/glfw-3.3.4.zip

23. If prompted to specify the generator for this project and to optionally indicate the platform for the generator, provide ‘Visual Studio 16 2019’ and ‘x64’, respectively, and click ‘Finish’
24. Click ‘Open Project’ or, equivalently, open `GLEW_BASEDIR\glew.sln`; this opens the Visual Studio solution `glew.sln` in Visual Studio Professional 2019
25. Select ‘Release’ from the Solution Configuration pull-down menu and select ‘Build:Build Solution’ from the menu bar to build glew in release mode

26. Run CMake (cmake-gui)
27. Enter `CPSSPATIALAR_BASEDIR\src` in the field to the right of ‘Where is the source code’ and `CPSSPATIALAR_BASEDIR\build` in the field to the right of ‘Where to build the binaries’ (note that backslashes are converted automatically to forward slashes)
28. Click ‘Configure’ and click ‘Yes’ if prompted to create the directory `CPSSPATIALAR_BASEDIR`
29. If prompted to specify the generator for this project and to optionally indicate the platform for the generator, provide ‘Visual Studio 16 2019’ and ‘x64’, respectively, and click ‘Finish’
30. Click ‘Open Project’ or, equivalently, open `CPSSPATIALAR_BASEDIR\build\cpspatialar.sln`; this opens the Visual Studio solution `cpspatialar.sln` in Visual Studio Professional 2019
31. Select ‘Release’ from the Solution Configuration pull-down menu and select ‘Build:Build Solution’ from the menu bar to build cpsspatialar in release mode

The **resulting executables** can be found at CPSSPATIALAR_BASEDIR`\build\bin\Release`