MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Live-view visualization tool for STEM detectors

Master's Thesis

MILAN HRABOVSKÝ

Brno, Spring 2026

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Live-view visualization tool for STEM detectors

Master's Thesis

MILAN HRABOVSKÝ

Advisor: RNDr. Pavol Ulbrich, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2026

FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

During the preparation of this thesis, I used the following AI tools:

- Grammarly for grammar check.
- ChatGPT codex for faster code writing / refactoring.
- ChatGPT to improve my writing style.

I declare that I used these tools in accordance with the principles of academic integrity. I checked the content and took full responsibility for it.

Milan Hrabovský

Advisor: RNDr. Pavol Ulbrich, Ph.D.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, RNDr. Pavol Ulbrich, Ph.D., for his guidance, support, valuable advice, and continuous direction throughout the development of this thesis. I would also like to thank my manager, Manuel Bornhoefft, Ph.D., for his support and assistance with fact-checking and technical consultations related to this work. Special thanks go to our tester, Bc. Radek Juráček, for testing the application and providing valuable feedback during the internal study and evaluation process. Finally, I would like to thank the entire Scallion team for their willingness to help, answer my questions, and provide support throughout the preparation of this thesis.

Abstract

The aim of this thesis is to implement a STEM Live Feed Widget running on scanning transmission electron microscopes (STEM). The work was conducted in collaboration with Thermo Fisher Scientific in Brno. The feature is to provide users in the factory responsible for microscope calibrations with a unified environment combining live STEM visualization together with calibration procedures and instructions. The implementation also creates opportunities for developing additional STEM-based plugins and automated procedures in the future.

The widget was implemented using the PyQt6 framework. As part of the implementation, a streaming acquisition task was developed to provide continuous live image acquisition and visualization. The application supports all required STEM controls necessary for operation and interaction with the microscope system. The final solution was tested on a real transmission electron microscope equipped with STEM support. The functionality, generated data, and overall behavior of the application were validated through practical testing and evaluation by an experienced tester familiar with the microscope systems and STEM workflows.

Keywords

Widget, STEM, Rendering, Threading, Wrapper, Data Acquisition, Scanning Transmission Electron Microscopy, C++, Python, PyQt6, Thermo Fisher Scientific ...

Contents

Introduction 1

1 Theory 3
- 1.1 How the Electron Microscope Works . . . 3
- 1.2 Main Types of Electron Microscopes . . . 5
  - 1.2.1 Transmission Electron Microscope . . . 5
  - 1.2.2 Scanning Transmission Electron Microscope . . . 6
  - 1.2.3 Comparison of TEM and STEM . . . 7
- 1.3 Electron–Matter Interaction . . . 8
- 1.4 Applications and Benefits . . . 9
- 1.5 Components . . . 10
  - 1.5.1 Vacuum System . . . 10
  - 1.5.2 Electron Source . . . 11
  - 1.5.3 Scanning System (Deflectors) . . . 13
  - 1.5.4 Objective Lens . . . 14
  - 1.5.5 Detectors . . . 16
  - 1.5.6 Spatial Resolution and Resolution Limits . . . 18
  - 1.5.7 Other Components . . . 20
- 1.6 Beam–Specimen Interaction Effects . . . 22
  - 1.6.1 Beam Blanking . . . 23
- 1.7 Current Practices in TEM Software and Their Limitations 23

2 Solution Design 25
- 2.1 STEM Live Feed Application . . . 26
- 2.2 Controls . . . 26
- 2.3 UI Design . . . 28
- 2.4 Backend Design . . . 30
- 2.5 Data Flow . . . 31
- 2.6 Rendering Data Streaming . . . 33
- 2.7 Detector Data Streaming . . . 34

3 Implementation 36
- 3.1 Interfaces . . . 37
  - 3.1.1 IStemAcquisitionTask . . . 37
  - 3.1.2 IStemRasterStreamingSession . . . 39
- 3.2 Important Classes . . . 40

3.2.1 C++ . . . 41
3.2.2 Python . . . 43
3.3 UI Implementation . . . 45
3.3.1 Frontend . . . 45
3.3.2 Backend . . . 47
3.3.3 Flucam Integration . . . 49
3.4 STEM Acquisition . . . 50
3.4.1 Acquisition Control . . . 51
3.4.2 Detector Data Acquisition . . . 51
3.4.3 Data Stream . . . 52
3.4.4 Rendering . . . 52
3.4.5 Threading Model . . . 54
3.4.6 Python Wrapper and GIL Handling . . . 54
3.4.7 Use of QTimer . . . 54

4 Testing **56**
4.1 Validation Test Applications . . . 56
4.2 Demo Application . . . 56
4.3 Unit Tests and Mocks . . . 57
4.4 CLI Test Application . . . 58
4.5 Simulator . . . 59
4.6 System Testing . . . 60
4.7 Internal Usability Validation . . . 61

5 Conclusion **62**

A An appendix **64**
A.1 Demo STEM Showcase Application . . . 64
A.1.1 Requirements . . . 64
A.1.2 Build Process . . . 65
A.1.3 Execution . . . 65

Bibliography **66**

Introduction

Microscopy enables observation and investigation of structures beyond the resolving power of the human eye, such as cells and microorganisms. This capability is essential for understanding physical and biological principles and the structural organization of matter. Microscopy plays a crucial role in advancing scientific knowledge and supporting engineering applications across various domains. The most widely known microscope is the optical microscope, which uses lenses and transmitted light to produce magnified images of specimens. However, its resolution is limited by the wavelength of visible light. For higher resolution, electron microscopy provides a suitable alternative. Electron microscopes achieve resolutions 100-1000 times higher than optical microscopes. They enable observation of structures at the nanoscale and atomic level by using the high energy of electrons and their short wavelength to overcome light microscopy limitations.

Electron microscopes are more complex to operate and maintain. Their use introduces challenges, including precise calibration to ensure accurate and reliable data. Addressing these challenges often requires sophisticated software tools. At Thermo Fisher Scientific in Brno, automation teams are developing software solutions that incorporate various tools and plugins into their codebase. During validation, developers frequently interact directly with electron microscopes and use existing tools for data acquisition and analysis. Specialized plugins have also been developed to support data visualization.

While visualization plugin for Transmission Electron Microscopy (TEM) is available, but there is no equivalent plugin in the calibration team's Scanning Transmission Electron Microscopy (STEM) suite. This limits the efficiency of development and testing workflows for STEM functionality and forces users to rely on external applications. As a result, multiple applications must be used simultaneously during microscope preparation, complicating the workflow and reducing productivity.

This thesis aims to design and implement a widget that adds STEM visualization capabilities to the existing calibration codebase. Although several external applications can acquire STEM data, this solution lets developers visualize it directly within their development

environment. This integration improves workflow efficiency and reduces reliance on external tools.

The proposed plugin also establishes a foundation for future extensions by providing reusable user interface components, controls, and integration mechanisms. By supporting rapid development and testing of STEM-related tools, this work increases productivity and facilitates continued expansion of the software ecosystem.

1 Theory

1.1 How the Electron Microscope Works

The Transmission Electron Microscope (TEM) (Figure 1.1) is an advanced imaging and characterization instrument that investigates materials at nanometer to atomic resolution. It operates by interacting a high-energy electron beam with a thin, electron-transparent specimen, then manipulating transmitted electrons to form images, diffraction patterns, and spectroscopic data.

TEM operation begins with the generation of electrons from an electron source, typically a thermionic emitter or field emission gun (FEG). These electrons are accelerated by high voltage, usually 80–300 keV. This increases electron energy, reduces their wavelength, and enables much higher spatial resolution than optical microscopes.

After acceleration, the electron beam is shaped and controlled by electromagnetic condenser lenses. These lenses regulate beam size, intensity, and convergence. Depending on the operating mode, the beam is either nearly parallel for conventional imaging and diffraction or focused into a fine probe for scanning transmission electron microscopy (STEM) and analytical applications [1].

The electron beam then transmitted through a very thin specimen, typically less than 100 nm thick. As electrons pass through the material, they interact with the atomic structure via elastic and inelastic scattering. Elastic scattering changes the direction of electrons without significant energy loss and mainly causes diffraction contrast and crystallographic information. Inelastic scattering involves partial energy loss from specimen interactions, generating signals like characteristic X-rays and energy-loss electrons used for chemical and electronic analysis. It also adds to background signal, reducing image contrast and sharpness.

Diffraction is a wave phenomenon that occurs when transmitted electron waves interact with the periodic atomic structure of the specimen. Scattered electron waves interfere, producing diffraction patterns that reveal the crystal structure, lattice spacing, and orientation of the material. Diffraction analysis is commonly used to investigate crystallographic properties.

After interacting with the specimen, transmitted electrons are described by a complex electron wavefunction carrying information in amplitude and phase. The interaction with the sample modifies this wavefunction through scattering processes. The objective lens, one of the most critical components of the TEM, collects and focuses the transmitted electron wave to form a real-space image or diffraction pattern. The imaging mode depends on the configuration of intermediate and projector lenses, which magnify and project the electron distribution onto the detector.

Although the information is physically encoded in the complex electron wavefunction, conventional detectors measure only the resulting intensity, which is proportional to the squared magnitude of the wavefunction, $|\psi|^2$. The phase information is not measured directly but inferred from interference effects that give rise to phase contrast imaging and diffraction phenomena.

Since electrons cannot be observed directly, the final signal is converted into a visible form. This uses fluorescent screens or digital detectors like charge-coupled devices (CCD) or complementary metal–oxide–semiconductor (CMOS) cameras. Measured electron intensity translates into brightness variations, producing interpretable images or diffraction patterns. [1].

Figure 1.1: Transmission Electron Microscope Titan Themis-Z with S/TEM.[1]

1.2 Main Types of Electron Microscopes

Electron microscopes can be broadly classified based on how the electron beam interacts with the specimen and how the resulting signals are detected. The most important types relevant to this work are the Transmission Electron Microscope) and the Scanning Transmission Electron Microscope (STEM).

1.2.1 Transmission Electron Microscope

In a conventional transmission electron microscope, a broad, nearly parallel electron beam illuminates a relatively large area of the specimen (what can be seen on the left part of the Figure 1.2) up to several hundred nanometers. The transmitted and scattered electrons are used

1. Image source: Weizmann Institute of Science

to form images or diffraction patterns. TEM is particularly well-suited for studying crystal structure, defects, and morphology at high spatial resolution.

Imaging in TEM is typically performed in different modes, such as bright-field and dark-field imaging, which rely on selecting specific scattered or unscattered electrons. For very high resolution in the TEM, the phase contrast, which is generated by the interference of the scattered and transmitted beams, is utilized. Additionally, TEM can provide diffraction patterns that reveal crystallographic information, including lattice spacing and symmetry. Because of its versatility, TEM is widely used in materials science, nanotechnology, and physics[1].

1.2.2 Scanning Transmission Electron Microscope

In scanning transmission electron microscopy, the electron beam is focused by the condenser and objective lens system into a highly localized probe, as shown on the right in Figure 1.2, which is scanned across the specimen in a raster pattern using scanning coils. As the focused probe interacts with the sample, transmitted and scattered electrons are collected by detectors positioned below the specimen. Unlike conventional TEM, where the image is formed directly by the imaging lens system, STEM images are reconstructed electronically from the detector signal acquired at each probe position. The image contrast, therefore, depends on both the probe–sample interaction and the specific detector geometry used to collect the scattered or unscattered electrons.

STEM offers several advantages over conventional TEM. The use of a focused probe enables highly localized analysis with nanometer- or even atomic-scale resolution. Different detectors can be used to collect various signals, such as bright-field, dark-field, or high-angle annular dark-field signals [2], providing complementary structural and compositional information. Furthermore, STEM is particularly well suited for analytical localized techniques such as energy-dispersive X-ray spectroscopy (EDS) and electron energy-loss spectroscopy (EELS), making it a powerful tool for chemical characterization at the nanoscale[1].

1.2.3 Comparison of TEM and STEM

Although TEM and STEM are based on similar physical principles and are often implemented within the same instrument, they differ in their typical applications and imaging characteristics. TEM is generally more suitable for rapid imaging and diffraction analysis over larger specimen areas, whereas STEM is preferred for localized structural and analytical investigations with high spatial precision.

An important advantage of STEM is the flexibility of signal detection. In particular, high-angle annular dark-field STEM imaging provides approximately atomic-number-dependent (Z-contrast) intensity, where brighter regions generally correspond to heavier atomic columns. Compared to high-resolution TEM imaging, STEM images are therefore often more directly interpretable.

Another important distinction is related to the distribution of the electron dose. In STEM, the focused probe can produce a high local instantaneous dose, which may increase the risk of beam-induced damage. However, the extent of the damage strongly depends on experimental conditions such as beam current, dwell time, scan strategy, and total accumulated dose. In conventional TEM, the electron dose is typically distributed more uniformly over a larger illuminated area.

In practice, TEM and STEM are complementary techniques. TEM is commonly used for general imaging and diffraction analysis, whereas STEM is preferred for localized structural characterization and analytical measurements at the nanoscale.

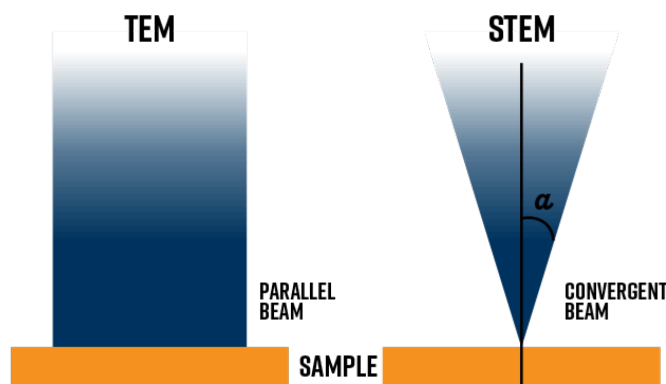
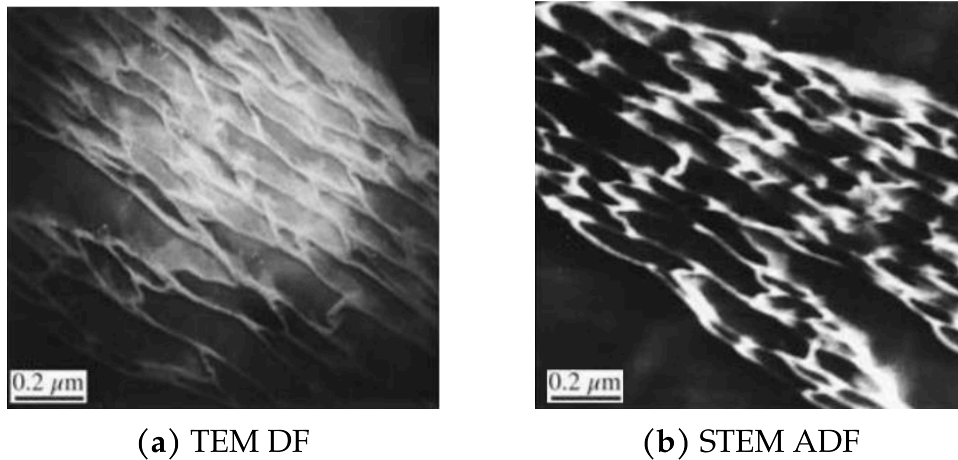


Figure 1.2: This figure shows the difference between TEM and STEM, how the electrons are emitted to the specimen.[2]



(a) TEM DF (b) STEM ADF

Figure 1.3: Comparison of dark field image from TEM and STEM [3]

1.3 Electron–Matter Interaction

When an electron beam passes through a specimen, it interacts with the atoms in the material. These interactions can be broadly divided into elastic and inelastic scattering.

In elastic scattering, electrons change direction without losing significant energy. This process provides information about the crystal structure and contributes to image contrast and diffraction patterns. In inelastic scattering, electrons lose energy as they interact with the specimen, producing signals such as characteristic X-rays and energy-loss electrons.

These interactions form the basis of all imaging and analytical techniques in electron microscopy, as different detectors collect different types of transmitted electrons, scattered electrons, or resulting signals like x-rays to extract structural and compositional information[3].

2. Image source: Nanoscience Instruments
3. Image source: STEM vs TEM

1.4 Applications and Benefits

Materials Science In Materials Science, it enables observation of detailed characterization of microstructures, including grain boundaries, phase distributions, and crystallographic defects. Such analysis is essential for understanding and predicting the mechanical, electrical, and thermal properties of materials [4].

In Chemistry and Surface Science, electron microscopy provides both structural and compositional information. When combined with analytical techniques such as energy-dispersive X-ray spectroscopy (EDS), often integrated with scanning electron microscopy (SEM) or transmission electron microscopy, it enables precise elemental analysis of materials[4].

Biology and Medicine In biology and medicine, electron microscopy enables the observation of cellular ultrastructure, viruses, and macromolecular complexes with exceptional detail. These insights contribute significantly to diagnostics, drug development, and fundamental biological research[5].

Semiconductor Industry In the semiconductor industry, electron microscopy is widely used for failure analysis, quality control, and the inspection of integrated circuits at nanoscale dimensions, where conventional optical methods are insufficient[6].

Nanotechnology In the field of Nanotechnology, electron microscopy enables the visualization and manipulation of nanostructures, nanoparticles, and thin films. This capability is fundamental to the development and optimization of advanced materials and nanoscale devices[7].

1.5 Components

1.5.1 Vacuum System

A high-vacuum environment is essential for the operation of a transmission electron microscope, as it allows electrons to travel through the column without significant scattering by gas molecules. The presence of air would lead to electron loss, reduced beam intensity, degraded image quality, and destruction of the electron source [1].

The vacuum system in a TEM typically consists of multiple stages, as you can see in Figure 1.4, including roughing pumps and high-vacuum pumps such as turbomolecular or ion pumps. These pumps work together to achieve and maintain pressures on the order of 10^{-5} to 10^{-7} Pa in the microscope column [8].

In addition to enabling efficient electron transmission, the vacuum also protects sensitive components, particularly the electron source. Field emission guns, in particular, require ultra-high vacuum conditions to ensure stable operation and to prevent contamination of the emitter tip[9].

The microscope is usually divided into separate vacuum regions, such as the gun chamber, column, and specimen chamber, which can be independently controlled. This design allows sample exchange through airlock systems without compromising the overall vacuum conditions.

Overall, maintaining a stable and clean vacuum environment is critical for achieving high-resolution imaging, ensuring instrument stability, and prolonging the lifetime of microscope components.

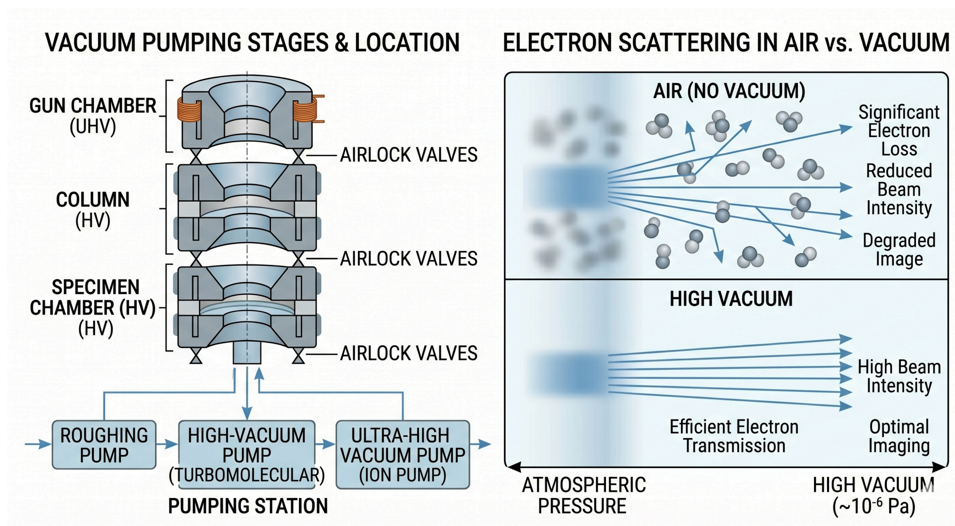


Figure 1.4: This image shows how the vacuum affects transmitted electrons.

1.5.2 Electron Source

The electron source (A in Figure 1.8) is a critical component of a transmission electron microscope because it determines the properties of the electron beam used for imaging and analysis. The source quality directly influences key beam characteristics such as brightness, coherence, and energy spread, which affect the resolution and analytical performance of the microscope [3].

In TEM systems, electrons are generated in an electron gun and accelerated by a high potential difference. Two main types of electron sources are commonly used: thermionic emitters and field emission guns (FEGs), which can be seen in Figure 1.5.

4. Image source: Thermo Fisher Scientific

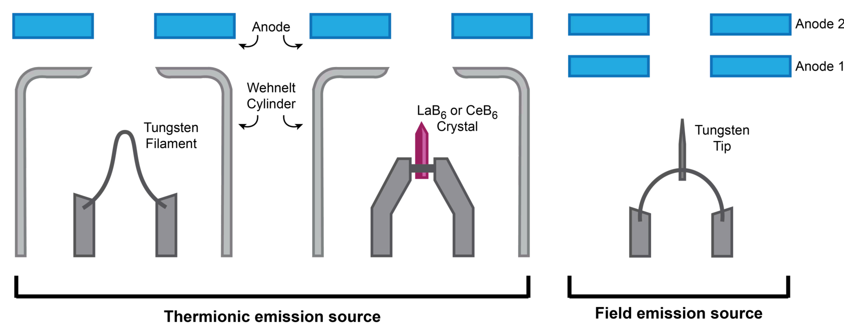


Figure 1.5: Different types of electron sources.[4]

Thermionic Emission Source

Thermionic emission is based on heating a material until electrons gain sufficient energy to overcome the material's work function and escape into the vacuum. Common thermionic emitters include tungsten filaments and lanthanum hexaboride (LaB_6).

Tungsten sources are simple and robust but require high operating temperatures. In contrast, LaB_6 sources have a lower work function, so they operate at lower temperatures and offer higher brightness and longer lifetime [10].

Thermionic sources are generally stable and easy to operate, making them suitable for routine imaging. However, they typically exhibit lower brightness and a broader energy spread than field-emission sources.

Field Emission Source

Field emission sources generate electrons by applying a strong electric field to a sharp metallic tip, typically made of tungsten. The localized electric field reduces the surface potential barrier at the tip, enabling electron emission. In pure cold field emission guns (cold FEGs), electrons are emitted primarily through quantum mechanical tunneling without thermal assistance[11].

However, most modern electron microscopes marketed as field-emission systems actually use Schottky emitters, also called thermal field emitters, which can be seen in Figure 1.5. In these sources, the

tungsten tip is moderately heated and coated with zirconium oxide, which lowers its work function. Electron emission is therefore supported by both thermal excitation and the applied electric field, resulting in improved beam stability and a higher emission current than in cold-field-emission sources [12].

Field emission guns provide higher brightness and a smaller energy spread than thermionic sources, resulting in improved beam coherence and spatial resolution. This makes them particularly suitable for high-resolution imaging and analytical techniques such as STEM and EELS [3].

1.5.3 Scanning System (Deflectors)

The scanning system in an electron microscope controls the position of the electron beam on the specimen. This is achieved using electromagnetic deflectors, which steer the beam in a controlled way [3]. By varying the deflection signals, the beam moves across the sample to form an image or perform localized analysis.

In scanning transmission electron microscopy, the beam is rastered over the specimen in a point-by-point manner. The deflection system ensures precise positioning of the beam, which is essential for accurate imaging and analytical measurements [3].

Scanning Deflectors

Scanning deflectors (E in Figure 1.8) are used to move the electron beam across the specimen in a defined pattern, typically a raster scan. This is accomplished by applying time-dependent currents to the deflector coils, generating magnetic fields that steer the beam in the horizontal and vertical directions [8].

The scanning process is synchronized with the detector system, allowing the signal intensity at each beam position to be recorded and reconstructed into an image [3]. The resolution and scan speed depend on parameters such as dwell time and scan size.

Beam Alignment Deflectors

In addition to scanning, deflectors are also used for beam alignment. These deflectors ensure that the electron beam remains properly aligned with the optical axis of the microscope. Misalignment can lead to image distortion, reduced resolution, and inaccurate analytical results [8].

Beam alignment is typically performed during microscope setup and calibration, and may also be adjusted dynamically during operation.

Stigmators

Stigmators are specialized deflectors used to correct astigmatism in the electron beam. Astigmatism occurs when the beam is not symmetrically focused, leading to distorted or blurred images.

By applying controlled electromagnetic fields, stigmators adjust the beam shape to restore circular symmetry, improving image sharpness and resolution [1].

Scan Rotation and Shift

Additional deflection systems allow for scan rotation and beam shift. Scan rotation changes the orientation of the scanned image without physically rotating the specimen, while beam shift enables fine positioning of the beam on the specimen [3].

These controls improve flexibility during imaging and calibration procedures.

1.5.4 Objective Lens

The objective lens is one of the most important components of a transmission electron microscope, as it strongly influences image formation and achievable spatial resolution. It is located directly below the specimen (H in Figure 1.8) and interacts with the transmitted and scattered electrons emerging from the sample.

In conventional TEM, the objective lens forms the first intermediate image of the specimen and simultaneously produces a diffraction pattern in its back focal plane. By selecting either the image plane or

diffraction plane using the microscope lens system, the microscope can operate in imaging or diffraction mode [8].

In STEM, the objective lens plays a different role. Together with the condenser lens system, it focuses the electron beam into a highly localized probe that is scanned across the specimen. The probe size and shape strongly affect the achievable spatial resolution and analytical performance.

The quality of the objective lens is limited by lens aberrations, particularly spherical and chromatic aberrations. Spherical aberration causes electrons passing through different regions of the lens to be focused at different positions, while chromatic aberration arises from variations in electron energy [3]. These aberrations reduce image sharpness and limit the achievable resolution in both TEM and STEM.

To improve image quality, apertures are commonly used together with the objective lens system to control which electrons contribute to the final image or detector signal, thereby enhancing contrast and reducing unwanted scattering contributions.

1.5.5 Detectors

Detectors in electron microscopy are used to collect signals generated by the interaction of the electron beam with the specimen. In scanning transmission electron microscopy, STEM detectors are positioned below the sample (L in Figure 1.8) to capture transmitted and scattered electrons at different angles, which can also be seen in Figure 1.6. The detector geometry determines the contrast mechanism and the type of information obtained from the image [2].

Figure 1.6: This figure shows, which detector collect which electrons.[5]

5. Image source: Microscopy Australia

Bright-Field (BF) Detector

The bright-field detector collects electrons that are transmitted through the specimen with little or no scattering, and is typically positioned along the optical axis.

BF imaging produces contrast mainly due to mass-thickness and diffraction effects. Regions that scatter electrons strongly appear darker, while regions that transmit electrons more efficiently appear brighter what can be seen on Figure 1.7 Image (a) [2].

Dark-Field (DF) Detector

The dark-field detector collects electrons scattered to intermediate angles while excluding the direct beam.

In DF imaging, only scattered electrons contribute to the image, causing strongly scattering regions to appear bright against a dark background Figure 1.7 Image (b). This technique is particularly useful for highlighting crystal defects, grain orientations, and phase variations within the material [2].

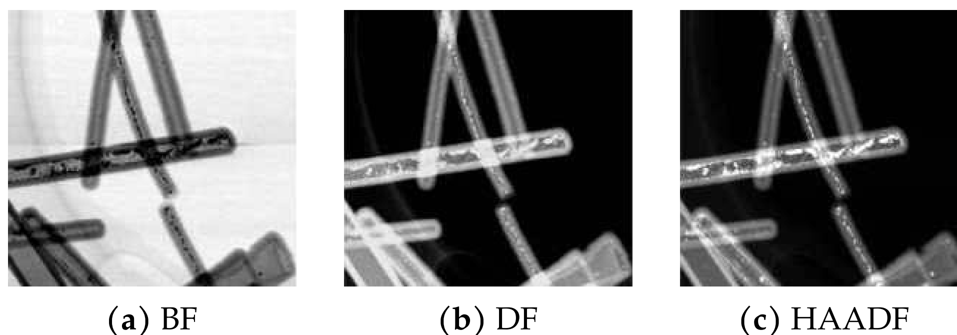
High-Angle Annular Dark-Field (HAADF) Detector

The high-angle annular dark-field detector collects electrons scattered to large angles using an annular detector geometry. [2]

HAADF imaging is often referred to as *Z-contrast* imaging, as the image intensity is approximately proportional to the atomic number of the elements present. Heavier elements scatter electrons more strongly and therefore appear brighter in the image, Figure 1.7 Image (c).

Compared to BF and DF imaging, HAADF provides reduced diffraction contrast and is less sensitive to sample orientation, resulting in images that are easier to interpret [3].

6. Image source: ETH Zürich



(a) BF (b) DF (c) HAADF

Figure 1.7: Results from different detectors [6]

1.5.6 Spatial Resolution and Resolution Limits

One of the main advantages of electron microscopy is its high spatial resolution, which is enabled by the short wavelength of high-energy electrons. This allows TEM and STEM to resolve features at the nanometer and even atomic scale.

However, the achievable resolution is limited by several factors. Lens aberrations, especially spherical and chromatic aberrations, distort the electron beam and reduce image sharpness. Spherical aberration causes electrons passing through different regions of the lens to be focused at different positions, while chromatic aberration occurs because electrons with different energies are focused differently by the lens system. Additionally, mechanical vibrations, sample drift, and instabilities of the sample stage or electron beam can further degrade resolution, particularly during long acquisitions and atomic-resolution imaging.

Spectroscopic Detectors (EDS and EELS)

In addition to imaging detectors, TEM systems are often equipped with spectroscopic detectors such as energy-dispersive X-ray spectroscopy (EDS) and electron energy-loss spectroscopy (EELS) (P in Figure 1.8). These detectors provide chemical and compositional information about the specimen.

EDS detects characteristic X-rays emitted from the specimen, allowing elemental identification and mapping. EELS measures the energy loss of transmitted electrons, providing information about elemen-

tal composition, bonding, and electronic structure. These techniques complement imaging by enabling detailed analytical characterization at the nanoscale [3].

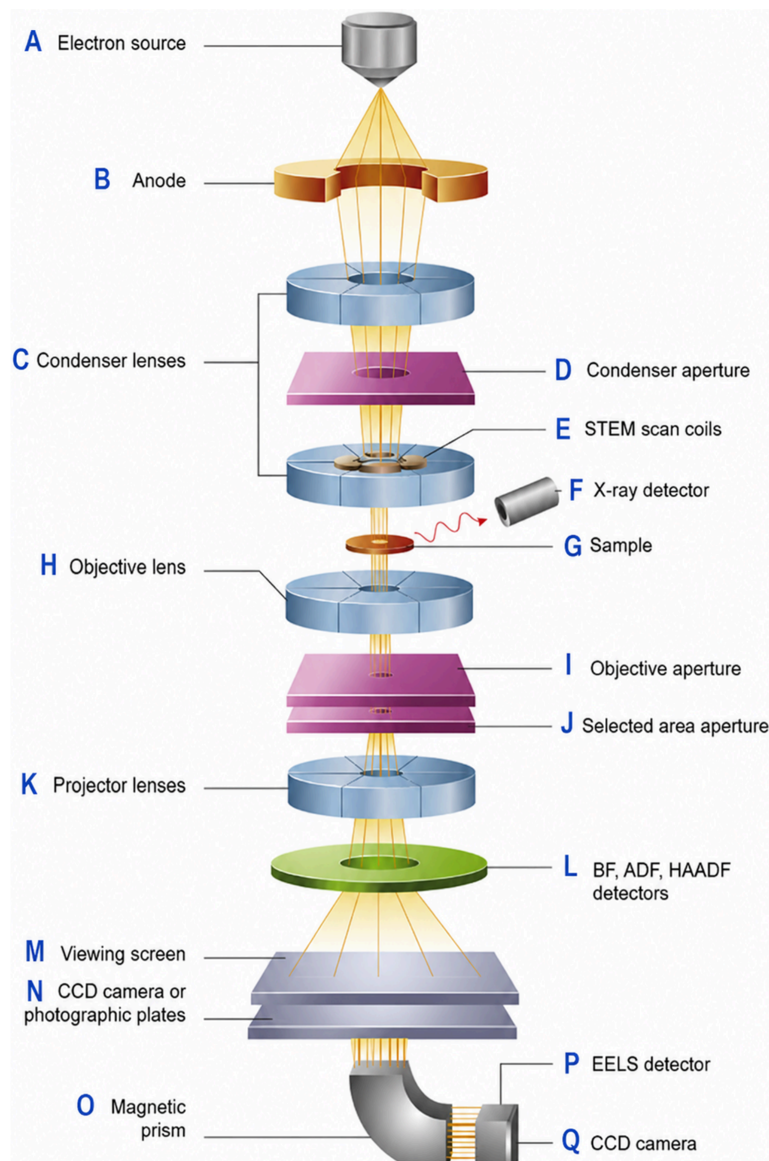


Figure 1.8: A schematic diagram of the transmission electron microscopy.[7]

1.5.7 Other Components

In addition to the main optical system, detectors, and vacuum system, a transmission electron microscope (TEM) includes several auxiliary components that are essential for its operation and usability.

Sample Holder and Stage

The specimen is mounted on a sample holder, which is inserted into the microscope through an airlock system. The stage allows precise positioning of the sample in multiple directions, including translation and tilt. This enables accurate alignment of the specimen and selection of specific regions for imaging and analysis [8].

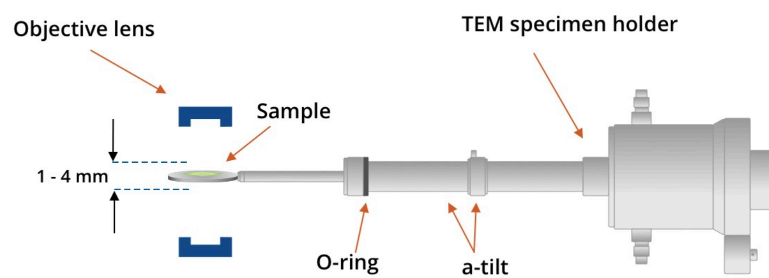


Figure 1.9: This figure shows, specimen holder.[8]

7. Image source: Transmission Electron Microscopy
8. Image source: Microscopy Australia

Apertures

Apertures are small openings placed at specific positions within the electron optical system, such as the objective aperture and selected area aperture. They limit the range of electron angles contributing to the image or diffraction pattern, thereby improving contrast and enabling control over the imaging conditions [1].

Control System

Modern TEM instruments use computer-based control systems and dedicated hardware interfaces, such as the hand panels commonly used in Thermo Fisher Scientific microscopes shown in Figure 1.10. These interfaces allow adjustment of parameters such as lens currents, beam alignment, stage movement, and detector settings. The control system enables precise microscope operation and is typically integrated with data acquisition and analysis software.

Figure 1.10: Hand control panels for the electron microscope.[9]

9. Image source: Microscopy Australia

FluCam

FluCam is a live-view camera system used in Thermo Fisher Scientific transmission electron microscopes. It observes a phosphorescent viewing screen inside the microscope column and provides real-time visualization for sample navigation, beam alignment, and microscope setup.

Stability and Environmental Control

To achieve high-resolution imaging, TEM systems require high mechanical and thermal stability. This is ensured through vibration isolation, temperature control, and shielding from external electromagnetic interference. These measures minimize disturbances that could otherwise degrade image quality [3].

1.6 Beam–Specimen Interaction Effects

The interaction between the electron beam and the specimen can lead to changes in the material being observed. Prolonged exposure to the electron beam may cause beam-induced damage, such as structural defects, heating, contamination, or atom displacement.

These effects are particularly important for beam-sensitive materials, where high-energy electrons can alter or destroy the sample during observation. In many cases, beam damage becomes the practical limiting factor of the experiment rather than the intrinsic resolution of the microscope itself. For this reason, techniques such as reducing beam intensity, minimizing exposure time, or parking the beam away from the region of interest are used to limit damage during imaging and analysis.

Understanding beam–specimen interaction effects is essential for obtaining reliable results and preserving the integrity of the specimen [1].

1.6.1 Beam Blanking

Beam blanking is a method in electron microscopy that temporarily prevents the electron beam from reaching the specimen. Instead of turning off the electron source, the beam is quickly moved away from the main path using electrostatic or electromagnetic deflectors.

The main purpose of beam blanking is to minimize unnecessary exposure of the specimen to the electron beam. If the beam stays on too long, it can damage the sample, cause contamination, or change its structure. By blanking the beam when data is not being collected, the total electron dose can be significantly reduced.

Beam blanking is also commonly used during scanning operations. For example, during the flyback period, when the beam returns from the end of one scan line to the beginning of the next, the beam is blanked to prevent unwanted signal acquisition and image artifacts.

1.7 Current Practices in TEM Software and Their Limitations

Modern software platforms for transmission electron microscopy provide integrated environments for data acquisition, imaging, and analysis. These systems streamline experimental workflows by combining imaging with spectroscopy techniques (such as EDS and EELS) and incorporating automation tools. Common features include drift compensation, real-time data processing, and intuitive user interfaces, enabling both experienced users and beginners to obtain reliable and reproducible results in nanoscale investigations.

Velox Application Velox is a representative example of such software, offering a comprehensive solution within a single platform [13]. It supports precise data acquisition and high-quality imaging while integrating analytical capabilities into a unified environment.

Calibration Oriented Workflow Despite these capabilities, Velox is primarily designed as an end-user application for standard microscopy workflows and must maintain a high level of stability and

reliability. Although it can be extended, integrating calibration-specific functionality into such a complex system is not optimal.

Calibration workflows in factory and service environments use a focused and streamlined interface that is closely integrated with procedural instructions. These workflows benefit from having all necessary tools available within a single application, rather than relying on multiple systems in parallel.

For this reason, developing a separate, lightweight solution is more appropriate. It allows the calibration team at Thermo Fisher Scientific in Brno to create targeted functionality while providing a simpler and more efficient user experience tailored to calibration tasks.

Sherpa is a standalone application that functions as a collection of plugins, used in factory and service environments. It integrates plugins developed by different teams, each providing specific functionality used during microscope preparation and maintenance.

These plugins support various tasks, such as calibration procedures, system checks, and configuration of microscope components. As a result, Sherpa serves as a unified environment for executing workflows related to microscope setup and calibration.

2 Solution Design

This chapter presents the design of a system for real-time STEM data acquisition and visualization. While many of the required components are already implemented, like acquiring a single image, blanking, setting rotation, parking beam, and detector insertion, their integration into a unified architecture with a focus on calibration tasks requires careful consideration of data flow, performance, and usability.

The user interface (UI) must provide all necessary controls for operating the STEM while remaining intuitive and easy to use. It should allow users to adjust acquisition parameters, monitor the system, and visualize data in real time without unnecessary complexity.

On the backend, the system needs to communicate with lower-level components responsible for data acquisition and streaming. These components handle microscope control and data transfer, so the communication must be efficient and reliable.

The system follows a layered architecture. The UI is implemented using the Qt framework (via PyQt) [14] and serves as the main point of user interaction. The UI communicates with Python modules that serve as wrappers for functionality implemented in C++. This approach simplifies interaction with the lower-level code while preserving performance.

The call hierarchy is structured as follows:

PyQt UI $\rightarrow$ *Python wrapper layer* $\rightarrow$ C++ *acquisition control* $\rightarrow$ *low-level communication components*

At the lowest level, the communication components directly interact with the acquisition hardware and execute operations related to data acquisition. This separation into layers improves modularity and makes the system easier to maintain and extend.

2.1 STEM Live Feed Application

The live feed application provides real-time visualization of data acquired from the microscope. At Thermo Fisher Scientific in Brno, it will be used by developers and users involved in microscope calibration, allowing them to monitor the specimen and immediately observe the effects of adjustments during operation.

In addition to displaying the live image, the application also computes and presents the Fast Fourier Transform (FFT)[15] of the acquired data. This enables users to analyze spatial frequency information, which is particularly useful for assessing image quality, focus, and alignment during calibration procedures.

The application further supports real-time interaction with the microscope, allowing users to adjust parameters for STEM acquisition and observe their effects without delay. An important feature of the application is beam parking, which allows the electron beam to be positioned away from the region of interest. This is essential because the continuous exposure of the specimen to the electron beam can cause damage, such as structural changes, defects, or contamination, especially during prolonged observation. By parking the beam in a non-critical area, users can minimize beam-induced damage to the region being studied.

2.2 Controls

Several parameters control how the scanning process is performed. The most important parameter is the *dwell time*, which defines how long the electron beam remains on a single pixel. Another key parameter is the image resolution, which determines the number of pixels in the scan, typically defined by the image width and height.

The scanning time can be estimated from the dwell time and image dimensions. The time required to scan a single line is given by

$$t_{\text{line}} = t_{\text{dwell}} \cdot N_{\text{width}} \tag{2.1}$$

where t_{dwell} is the dwell time per pixel and N_{width} is the number of pixels in one line.

The total time required to scan a full frame can then be calculated as

$$t_{\text{frame}} = t_{\text{line}} \cdot N_{\text{height}} \tag{2.2}$$

where N_{height} is the number of lines in the image. This first estimation ignores further complications taken into account for the physics of the scanning, such as the fly back time of the beam after every line scanned.

Additional controls include scan rotation, which adjusts the orientation of the scanned image, and camera length, which affects the magnification (or effective zoom) of the scan.

Fourier Transform The Fourier transform is an important tool in electron microscopy, as it allows images to be analyzed in the frequency domain. In this representation, periodic structures in the specimen appear as distinct spots or patterns, which can be used to determine structural information such as lattice spacing.

In practice, the Fast Fourier Transform (FFT) is commonly applied to acquired images. This is particularly useful for evaluating image quality, detecting periodic features, and assisting with alignment and focus.

The relationship between real-space images and diffraction patterns reflects the wave nature of electrons in electron microscopy. Diffraction patterns are related to the Fourier transform of the complex electron wavefunction emerging from the specimen, while recorded detector images represent intensity distributions derived from this wavefunction. Similarly, the FFT of an acquired image provides a frequency-domain representation of the recorded image intensity and is therefore closely related, although not identical, to the experimentally observed diffraction pattern.

Camera Length Camera length is an effective imaging parameter used in electron diffraction and STEM imaging. It determines how the diffraction pattern is projected onto the detector and can be understood as the effective distance between the specimen and the recording plane. In practice, it is not a direct physical distance, but rather a parameter controlled by the microscope lens system.

Increasing the camera length causes the diffraction pattern to spread over a larger area of the detector, while decreasing it results in a more compact pattern.

In STEM, the effective camera length influences which scattering angles are captured by the detector. As a result, changing the camera length affects the angular range of detected electrons and therefore also the resulting image contrast.

2.3 UI Design

The main goal of the user interface (UI) design was to maximize the available space for image visualization while maintaining an intuitive layout and ensuring that essential controls remain easily accessible. For this reason, the STEM Live Feed UI differs from the existing TEM Live Feed interface. The TEM interface was developed incrementally and lacks a clear structure. It attempts to prioritize horizontal space; however, this approach is limited in practice, as the overall layout is constrained by vertical space.

To increase the rendering area, primary controls were moved to the left side of the interface, and less frequently used controls were partially hidden. This approach allows for a larger visualization area without reducing usability. The rendering area itself can be divided into two sections. While the user can display only the main image, additional views such as FFT or Flucam (a camera used during system calibration) can also be enabled.

Users also required a fast and simple way to switch between different views, particularly when calibrating the system. This functionality is provided through tabs located at the top of the widget, allowing easy switching between rendering modes and camera views.

The UI provides all necessary controls for acquisition. These include detector selection (top of control panel in Figure 2.1), a camera length drop-down menu (as camera length is selected from predefined values rather than entered manually), and dwell time control (in the middle of the control panel in Figure 2.1). Image adjustments are supported through sliders for brightness and contrast, along with a dedicated auto-contrast button. A rotation control is available both

as a slider and as a manual input field. Additionally, users can select the maximum frame size to control image resolution.

Several buttons are provided to control acquisition-related actions, including *Search*, *Acquire*, *Blank*, *Focus*, *Show Flucam / Show FFT*, and *Beam Position Marker*. The *Search* function initiates continuous line-by-line acquisition, while *Acquire* captures a single full frame. The *Blank* function is used to control detector blanking.

The UI also includes a focus control feature for scanning just in the selected area, which is implemented as a collapsible panel (visible below the main widget in Figure 2.1), as it is not required during standard operation.

When focus mode is activated (red rectangle in Figure 3.4), an interactive rectangle is displayed within the image view. This rectangle can be repositioned and resized by the user to define a specific region of interest. The selected region is used during acquisition. When the *Search* function is initiated in focus mode, only the defined area is scanned and updated. Reducing the scanned area decreases the number of acquired pixels, which allows faster image updates and more responsive focusing and alignment. In addition, limiting the exposed region reduces unnecessary electron irradiation of the surrounding specimen, helping to minimize contamination and beam-induced damage.

Additional controls affect the visualization area. The Show Flucam and Show FFT options toggle auxiliary views displayed alongside the main rendering.

Finally, the Beam Position Marker (red reticle in Figure 3.4)) The function allows the user to adjust the beam position. This enables the beam to be moved away from the area of interest, effectively allowing the user to "park" the beam when needed.

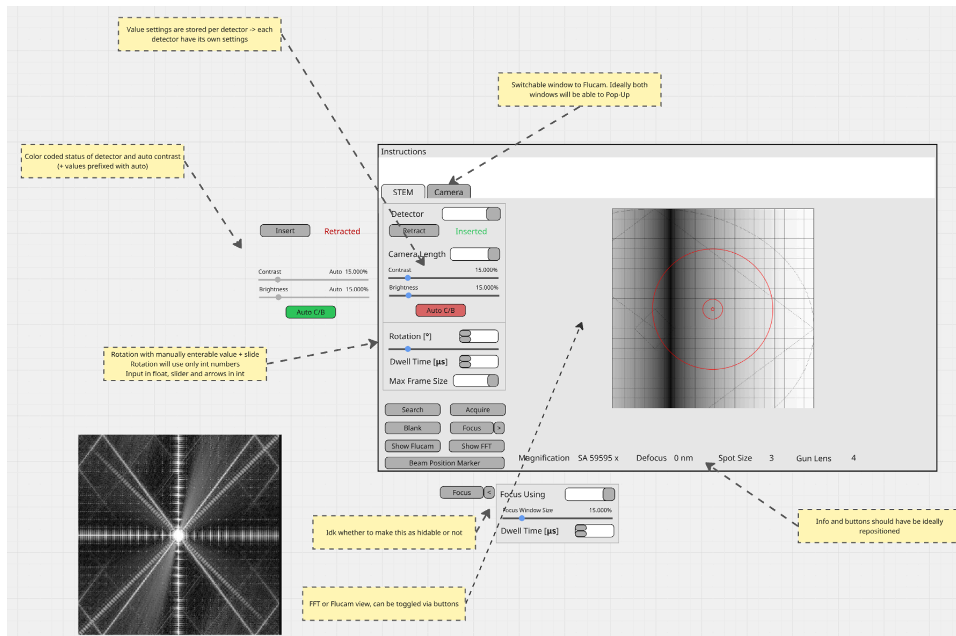


Figure 2.1: Design of the user interface for the widget.

2.4 Backend Design

The backend architecture separates user interface responsibilities from acquisition and device control logic. The UI, implemented in PyQt, is responsible for rendering and user interaction, while acquisition-related operations are handled through Python wrappers and lower-level components.

The UI is structured around a base class from which the main widget inherits. This widget is integrated into a tab wrapper, which provides functionality for switching between different views. The widget does not communicate directly with the hardware; instead, it invokes functions exposed by Python wrappers responsible for STEM tasks and device control.

Single-frame acquisition functionality is already available. However, continuous line-by-line acquisition must be implemented. This requires a streaming mechanism between the detector and the Python layer, enabling real-time updates of the rendered image.

To prevent blocking the UI, acquisition and streaming operations are executed in separate threads. This ensures that long-running acquisition tasks do not interfere with user interaction or rendering.

The system operates with two main data flows. The first flow transfers data from the detector to the raw buffer, where incoming data is stored in its original form. The second flow processes data from the raw buffer to the rendering (drawing) buffer, which represents the image displayed to the user.

During this processing step, image adjustments such as brightness and contrast are applied. These operations are implemented in Python using NumPy [16] to ensure efficient computation with minimal performance impact. The design goal is to ensure that system performance is limited by the acquisition process rather than by data processing or rendering.

FFT computation is performed directly on the raw buffer data. This ensures that frequency-domain analysis is based on unmodified data and is not affected by display adjustments.

All hardware interactions are handled indirectly through Python wrappers, which provide an abstraction layer over lower-level functionality.

Beneath the Python wrapper layer is the *StemAcquisition* layer. This layer manages acquisition processes, including starting, stopping, and restarting acquisition, configuring parameters, validating input settings, and monitoring the state and availability of detectors. It also handles buffer management and streaming control.

The lowest layer consists of communication components that directly interface with the microscope hardware. These components execute operations through predefined interfaces and are responsible for low-level control of the detectors and data acquisition hardware.

2.5 Data Flow

The data flow is designed to separate acquisition, processing, and visualization. The detector continuously sends data to the raw buffer, which stores unprocessed acquisition data. This acquisition stream runs in a dedicated thread to ensure continuous data transfer without blocking the UI.

From the raw buffer, data follows two paths. The first path leads to the rendering buffer, where image processing operations such as brightness and contrast adjustments are applied. The processed data is then converted into a *QImage* object and directly rendered in the UI. This processing and rendering pipeline operates independently of the acquisition thread.

The second path is used for FFT computation, which operates directly on the raw data to ensure accurate frequency-domain analysis. The FFT result is generated as a separate image, independent of the rendered image, and can be displayed alongside the main view.

This separation of acquisition, processing, and rendering into independent threads ensures that the UI remains responsive during continuous streaming.

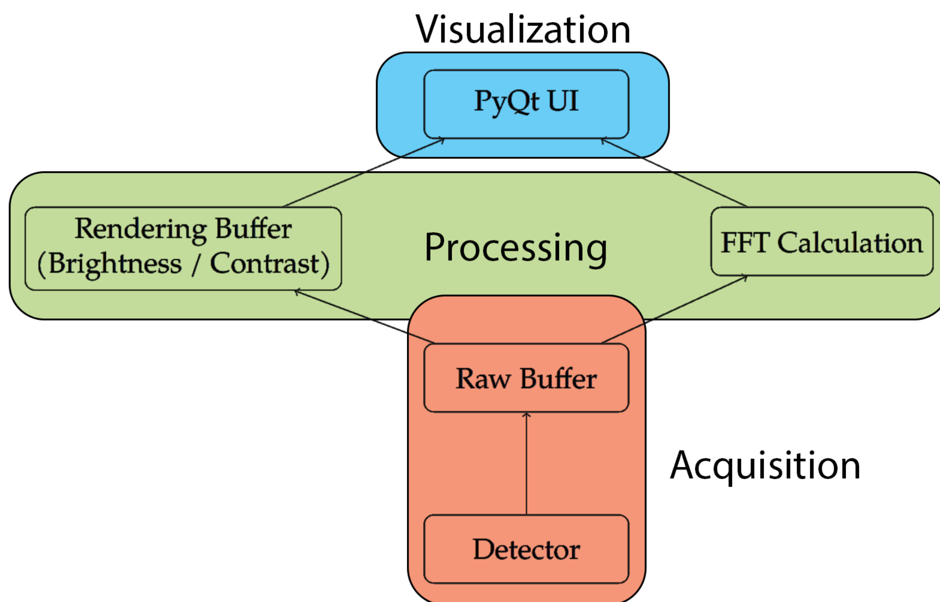


Figure 2.2: Backend data flow showing separation between acquisition, processing, and visualization.

This layered design ensures a clear separation of concerns, improves maintainability, and allows individual components to be extended or modified independently.

2.6 Rendering Data Streaming

The rendering data streaming mechanism provides a direct connection between acquired data and its visualization in the UI. The core component of this mechanism is the *raw buffer*, which stores image data received from the detector.

The raw buffer has the same dimensions as the final image and is directly exposed to Python as a memory view. This allows Python to operate on the data without copying or serialization, which significantly improves performance. The detector continuously updates this buffer as new data are acquired.

To ensure correct rendering, updates are performed incrementally on a per-line basis. After each acquired line, a notification is sent to the UI layer, indicating which line of the image has been updated. Thanks to PyQt's *QImage* update capabilities, only the modified lines are redrawn, avoiding the need to re-render the entire image.

The raw buffer also serves as the source of original, unmodified data. This is important for image processing operations such as brightness and contrast adjustment. Since these operations are applied during rendering, the original data remains unchanged, preventing cumulative loss of information caused by repeated modifications.

Because Python maintains a direct reference to the buffer, its lifetime is tied to the lifetime of the widget. This design avoids unnecessary data transfers and ensures efficient memory usage.

However, a potential issue arises when the acquisition rate exceeds the rendering speed. In such cases, incoming data can outrun rendering. To address this, the system uses a notification-based mechanism with frame identifiers and line indices. Each notification specifies the corresponding frame and line of the updated data.

Notifications are stored in a circular (ring) buffer. If a new notification arrives for the same line but with a higher frame identifier, it overwrites the older one. This ensures that only the most recent data are rendered and outdated updates are effectively discarded.

This approach prioritizes responsiveness over perfect visual continuity. Minor inconsistencies like visible line updates or differences between lines are expected. For example, when the user changes the rotation in the middle of the acquisition, it is expected that the image will be rotated, but not continuously, since updates will come in dif-

ferent phases of rotation, until rotation and scan are stable, but these are acceptable, as the image stabilizes once the user stops interacting with the system. The design assumption is that precise observation occurs after acquisition or interaction has paused.

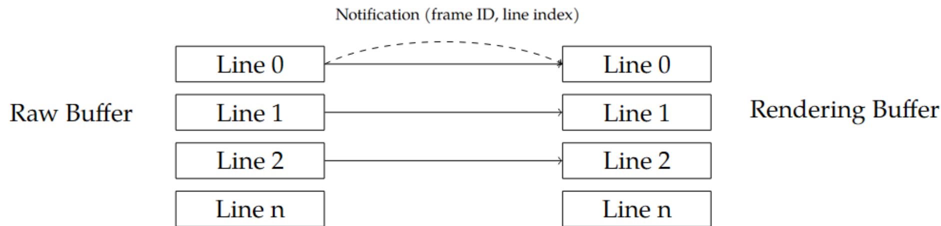


Figure 2.3: This figure shows how the buffers updates.

2.7 Detector Data Streaming

Before starting acquisition, the detector must be verified (on code level) to ensure it is operational and available. Once validated, acquisition parameters are configured. These include image resolution, dwell time, and acquisition mode. The acquisition mode defines whether the system performs full-image raster scanning or operates on a restricted region of interest (ROI). The ROI-based mode is particularly useful for focusing, as it limits scanning to a selected area while maintaining the defined resolution.

The detector interface emits data-access events, which are received by the callback methods:

- *OnImageDataChanged*
- *OnPatternDataChanged*
- *OnImageBufferChanged*
- *OnPatternDataBufferChanged*

These callbacks can be categorized based on data type and level of detail.

Image vs. Pattern callbacks

- Image callbacks correspond to raster-style STEM acquisition, where data are acquired line-by-line to form an image.
- Pattern callbacks correspond to non-raster acquisition modes, such as spot, line, or custom scanning patterns.

DataChanged vs. BufferChanged

- *DataChanged* callbacks provide precise metadata about modified regions. For image data, this includes a dirty rectangle specifying the updated area. For pattern data, this corresponds to a range of updated samples.
- *BufferChanged* callbacks indicate that the underlying data buffer has changed, but do not provide detailed information about the modified region.

Although acquisition may be shared with other components, it is beneficial to observe and handle all available callbacks to ensure compatibility and extensibility of the acquisition system for future implementations. However, for real-time STEM streaming, the *OnImageDataChanged* callback is the most suitable. This is because it provides both frame-related information and the exact region of the image that has been updated, enabling efficient incremental rendering.

The system also supports synchronization mechanisms for acquisition. While multiple synchronization modes are available, such as software or hardware. Software synchronization is ideal in this implementation. In this mode, synchronization occurs at the end of each scanned line (there is also an option for at the end of the frame). At this point, acquisition pauses briefly to maintain alignment with the processing and rendering pipeline. This approach helps prevent acquisition from significantly outrunning rendering and contributes to more stable real-time visualization. The detector generates notifications at approximately 60 Hz, each containing information about the updated image region.

3 Implementation

The final implementation differs slightly from the original design, primarily in the approach to streaming. The initial design proposed the use of a circular buffer for storing per-line notifications, where older entries would be overwritten by newer ones. These notifications were intended to carry both line identifiers and frame identifiers.

In practice, this approach proved unnecessary. Due to sufficient rendering performance, the system is capable of processing incoming notifications without the need to discard or overwrite them aggressively. As a result, the circular buffer was replaced with a bounded queue with a predefined maximum size. This simplifies the design while still preventing unbounded memory growth.

Another important change concerns synchronization. The original design included software synchronization at the end of each scanned line to prevent acquisition from outrunning rendering. However, experimental evaluation showed that rendering performance is sufficient to keep up with acquisition, making this synchronization unnecessary. Moreover, synchronization introduced additional delays, as scanning had to pause at the end of each line before continuing.

Since lower resolutions, update regions correspond to the entire frame, enforcing line-by-line synchronization significantly degraded performance, as the acquisition process was repeatedly paused. Therefore, synchronization was removed to allow continuous streaming and improve overall system responsiveness.

These changes resulted in a simpler and more efficient streaming pipeline without negatively impacting rendering accuracy.

3.1 Interfaces

During development, several interfaces were introduced across different layers of the system.

3.1.1 IStemAcquisitionTask

The main interface responsible for controlling and exposing acquisition functionality to the Python layer is *IStemAcquisitionTask*. This interface defines all required settings, data structures, and operations for managing the STEM acquisition stream (the interface functionality can be seen in Figure 3.2).

In addition to acquisition control, the interface provides direct access to the underlying raw data buffer, which stores the unprocessed pixel data. The raw buffer data can be accessed via the following methods.

- GetRawBuffer() - returns a direct reference to the raw buffer
- GetRawBufferSnapshot() - returns a copy of the raw buffer
- CopyRawBufferRows() - copy rows in the area to the destination array

The interface is wrapped using *pybind11* and exposed to Python. This allows the Python layer to hold a direct reference to the raw buffer, eliminating the need for serialization and deserialization of image data. As a result, the data transfer between C++ and Python is highly efficient and suitable for real-time streaming.

The interface also provides factory methods for creating acquisition task instances:

```
std::shared_ptr<IStemAcquisitionTask>
  CreateStemAcquisitionTask(
    std::shared_ptr<IStemDetector> stemDetector)
        const override;

std::shared_ptr<IStemAcquisitionTask>
  CreateStemAcquisitionTaskUsingName(
    std::string stemDetectorIdentification)
        const override;
```

These factory methods allow flexible initialization of acquisition tasks, either by directly providing a detector instance or by specifying a detector identifier.

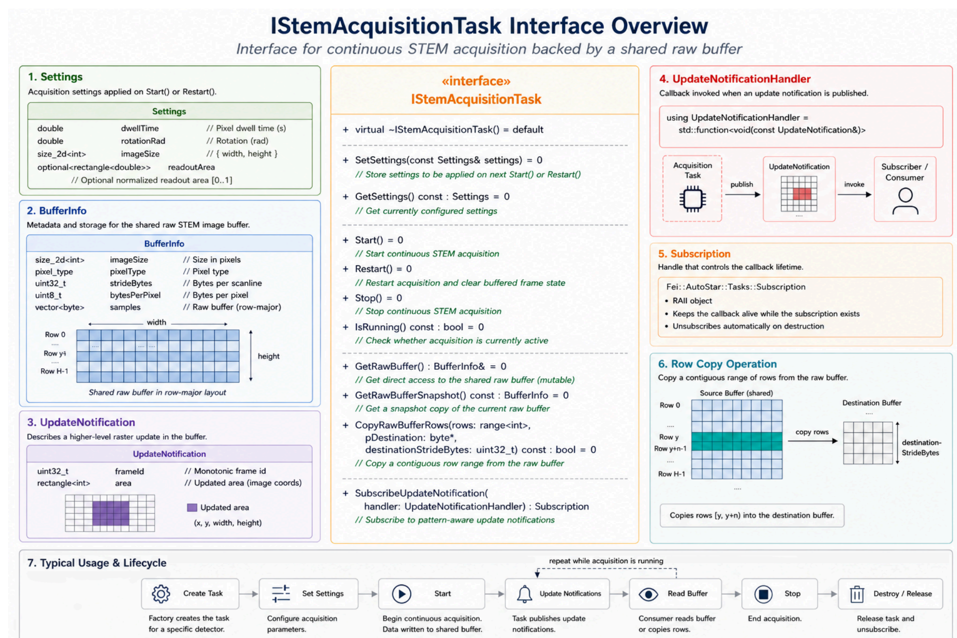


Figure 3.1: Visualization of the IStemAcquisitionTask interface and typical usage.

3.1.2 IStemRasterStreamingSession

Due to the layered architecture, the *IStemAcquisitionTask* interface does not communicate directly with microscope components. Instead, this responsibility is delegated to a lower-level interface, *IStemRasterStreamingSession*, which handles communication with the microscope and the underlying acquisition pipeline.

This interface provides functionality for receiving and handling callbacks from events emitted by the scan task (the interface functionality can be seen in Figure 3.2). For streaming purposes, it exposes a subscription mechanism that allows higher layers to register callback handlers for acquisition events. These callbacks are invoked whenever new data is available, enabling efficient propagation of updates to higher-level components.

In addition, the interface provides methods for copying acquired data (e.g., selected rows) from the scan task into the *raw_buffer*, which is later used for rendering and processing.

The *IStemRasterStreamingSession* interface is not directly exposed to Python. Instead, it is used internally within the C++ layer and serves as an intermediary between low-level acquisition components and higher-level abstractions.

The interface also includes a factory method for creating instances of the streaming session:

```
TEM_IOM4_EXPORT std::shared_ptr<
    IStemRasterStreamingSession>
    CreateStemRasterStreamingSession(
      CComPtr<Fei::Imaging::Pipeline::Com::
        IPipelineSource> pDetectorOutput);
```

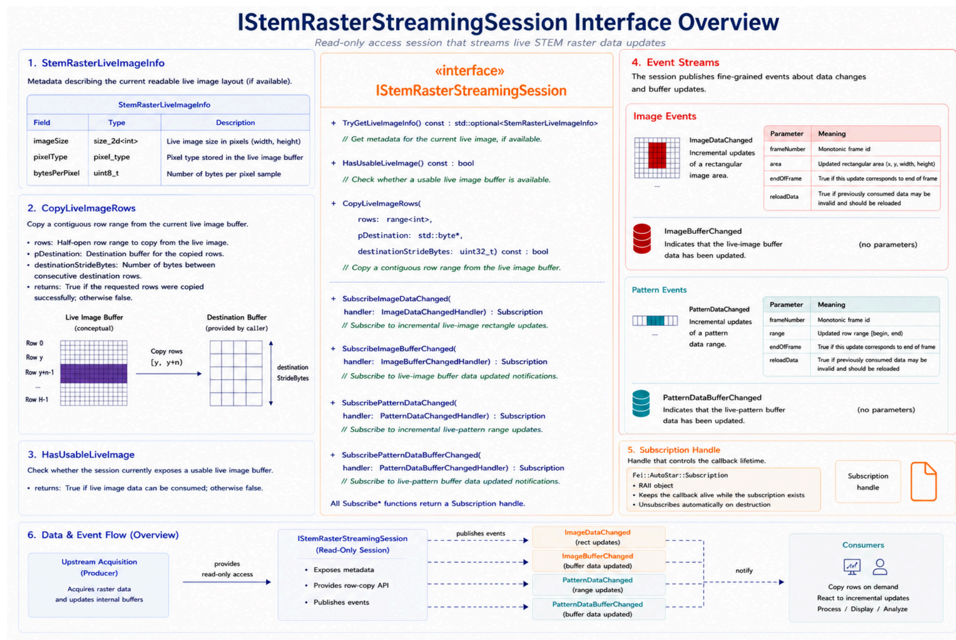


Figure 3.2: IStemRasterStreamingSesssion visualization.

3.2 Important Classes

Due to the application's complexity and the requirement for a modular architecture, the STEM Live Feed application is structured into multiple classes distributed across the C++ and Python layers. Each class is responsible for a specific aspect of the application, such as data acquisition, state management, user interface, or communication between layers.

This section describes the main classes that form the core of the application architecture. Although the implementation contains additional helper classes, the following components are the most important for understanding the system design and functionality.

3.2.1 C++

DataAccessEventsSink is a C++ adapter class responsible for receiving data update events from the microscope imaging pipeline via the COM interface *IDataAccessFilterEvents*. The class is registered as an event listener using the COM connection-point mechanism, through which it receives low-level callbacks.

Upon receiving these callbacks, the class converts COM-specific arguments into native C++ types and republishes them as signal. This allows higher-level components to subscribe to strongly typed events without directly interacting with COM interfaces.

The primary purpose of this class is to isolate COM-specific event handling from the rest of the streaming logic. Instead of processing raw COM callbacks, higher-level components such as *StemRasterStreamingSession* subscribe to the signals exposed by *DataAccessEventsSink*.

IStemAcquisitionTaskWrapper is a binding class implemented in the *AcquisitionTasksPythonWrapper* module using *pybind11*. Its purpose is to expose the native C++ interface *IStemAcquisitionTask* to the Python layer, including task control methods, configuration structures, access to raw image buffers, and update notifications.

This wrapper enables the Python application layer to interact with STEM acquisition tasks while keeping the underlying acquisition logic implemented in C++. As a result, the design preserves performance-critical functionality in the native layer while providing a flexible and convenient interface for higher-level application logic.

FakeIStemAcquisitionTask is simulation implementation of the *IStemAcquisitionTask* interface. Instead of communicating with a real STEM detector, it generates synthetic image data and emits update notifications in the same way as a real acquisition task.

Its primary purpose is to support development, testing, and demonstration of the acquisition pipeline and user interface without requiring access to microscope hardware.

The image is created procedurally by combining multiple mathematical intensity patterns, including sinusoidal stripes, radial ring structures, local checkerboard contrast, moving bright features, and pseudo-random noise. How the synthetic data are produced can be seen on Figure 3.3.

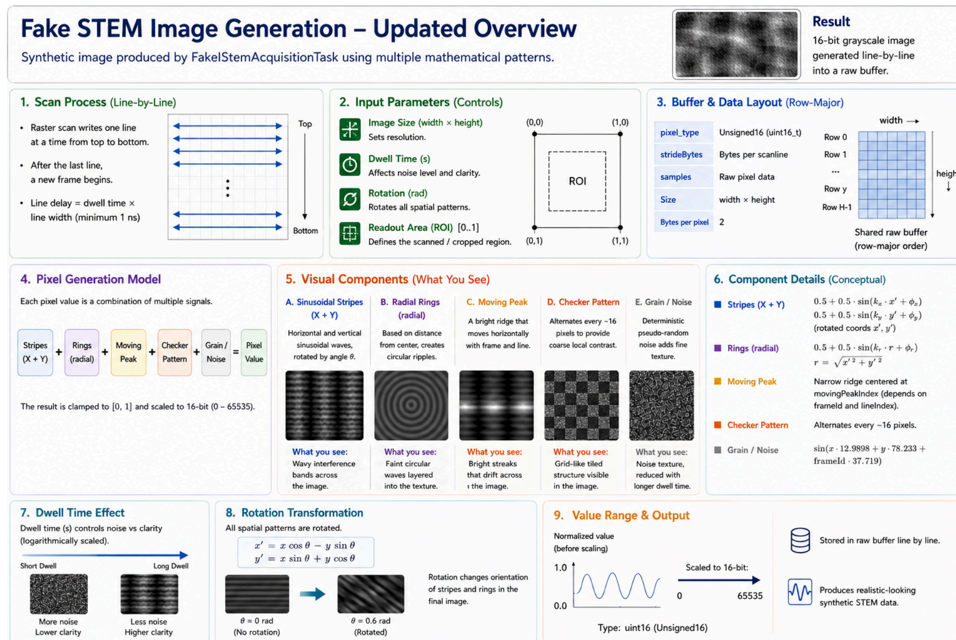


Figure 3.3: FakeIStemAcquisitionTask visualization.

3.2.2 Python

AcquisitionTasksFactory is a Python factory class responsible for creating acquisition task objects used by the application. It wraps the underlying C++ acquisition task factory and provides Python-side instances, such as *StemAcquisitionTask*, which are then consumed by higher-level control logic.

Its primary role is to encapsulate object creation and hide the complexity of the native wrapper layer. This design provides a clean and consistent entry point for obtaining acquisition tasks.

StemLiveFeedWidget is the main user interface component of the STEM live feed module. It integrates acquisition controls and preview areas into a single widget and serves as the primary interaction point for the user. Its role is to orchestrate UI behavior while delegating acquisition control, rendering, and state updates to dedicated components.

StemLiveFeedAcquisitionController represents the main control layer of the STEM live feed. It translates user actions into acquisition operations and manages the acquisition lifecycle, including start, stop, and restart. It also processes incoming updates and coordinates display refreshes, acting as a bridge between the UI and acquisition logic.

StemPreviewController manages the visual presentation of STEM preview data. It controls the main preview and optional secondary views, such as FFT or Flucam images. Its role is to handle layout, visibility, and interactive overlays, separating visualization concerns from acquisition and UI logic.

StemTaskRuntime encapsulates the threaded execution of the acquisition task. It runs acquisition logic in a dedicated worker thread and communicates with the UI using Qt signals. This ensures that long-running operations do not block the GUI and allows safe interaction between threads.

StemAcquisitionState is a lightweight container representing the current state of the acquisition process. It tracks lifecycle transitions and maintains identifiers used to filter outdated asynchronous events. Its purpose is to ensure consistent behavior during concurrent operations and user interaction.

StemFramePipeline is a helper class responsible for processing and preparing STEM image data for display. It maintains the raw frame data, display buffer, and corresponding Qt image representation. Its role is to support incremental updates by applying only modified regions of incoming data, enabling efficient real-time visualization.

BufferedImage is a reusable image display widget that supports efficient partial updates. It allows updating only modified regions of an image instead of redrawing the entire frame. This behavior is essential for incremental rendering of streaming STEM data.

_StemPreviewImage is a specialized preview widget used for displaying interactive STEM-related images. It inherits from *BufferedImage*, reusing its image rendering and partial repaint capabilities, and extends it with support for interactive overlays.

Its role is to host and manage overlay elements, such as *CrosshairShape* and *FocusAreaShape*. It forwards mouse events (press, move, and release) to these shapes, updates the cursor based on the current interaction, and renders the overlays on top of the displayed image. In practice, it serves as the interactive image surface used by *StemPreviewController* for STEM, FFT, and Flucam previews.

PreviewInteractiveShape is an abstract base class for interactive overlays displayed on preview images. It defines common functionality for rendering, hit testing, and user interaction. This abstraction enables extensible graphical annotations without coupling interaction logic to the image widget.

TabWidgetWrapper is a wrapper around *QTabWidget* that simplifies tab management and access to contained widgets. It provides utility methods for adding, removing, and querying tabs. In this project, it is used to integrate the STEM and camera live feed views into a unified interface.

StatusReportWidget is a UI component that displays relevant microscope state alongside the STEM preview. It retrieves values such as magnification, defocus, and camera length from the TEM polling mechanism. Its role is to provide contextual information during operation without directly affecting acquisition logic.

3.3 UI Implementation

The user interface is implemented in Python using the PyQt framework. While PyQt is responsible for rendering the interface, it also handles many application-level functions, including handling user input, managing timing, and coordinating communication between threads.

The main part of the interface is implemented in the *StemLiveFeedWidget* class, which encapsulates the core UI functionality of the live feed module. This widget is designed to be integrated into the plugin suite (Sherpa), where it provides a unified interface for controlling acquisition and visualizing STEM data.

From an architectural perspective, the user interface can be divided into two parts. The frontend layer is responsible for user interaction and presentation, including widget layout, controls, and event handling. The backend layer is connected to the underlying application logic and manages data flow, acquisition control, and communication with lower-level components.

3.3.1 Frontend

For simplicity, easier future layout modifications, and improved code maintainability, most of the user interface elements (such as buttons and the main layout) were designed using *Qt Designer 6*, from which the corresponding GUI class was generated. This class defines UI

components, including their types, default values, and valid ranges. The same approach is used for the *StatusReportWidget*, which displays current microscope parameters.

The *StemLiveFeedWidget* class inherits from the generated GUI class, providing direct access to all UI elements that can be seen marked as a blue area in Figure 3.4. The layout follows the original design with minor adjustments.

The user interface is divided into three main areas. The left side contains control elements defined in the generated GUI class. The bottom area displays the *StatusReportWidget*, which can be seen marked as a green area in Figure 3.4. The remaining space is used for image previews, which are defined programmatically. The entire widget is wrapped in a floatable container, allowing it to be detached from the main application window. Additionally, the widget is integrated into a *TabWidgetWrapper*, which enables switching between the STEM live feed and the Flucam view what can be seen in Figure 3.4 marked by the purple area.

Certain interactive features cannot be defined using Qt Designer and are therefore implemented programmatically. One example is the use of a *QSplitter*, which allows the user to dynamically adjust the size of preview areas when displaying multiple images. The behavior of the splitter and preview layout is managed by the *StemPreviewController*.

Several UI components were also adjusted to improve usability. For example, standard spin boxes were replaced with incremental controls that modify values multiplicatively, allowing faster adjustment over larger ranges. The focus control panel was also redesigned as a collapsible element, enabling it to be hidden when not in use.

Image rendering is implemented using the *BufferedImage* class, which uses *QImage* and *QPixmap* for display. The update mechanism is divided between two components: *StemPreviewController* determines which parts of the image need to be updated, while *BufferedImage* performs the actual repainting of the affected regions.

Interactive overlays are implemented using a hierarchy based on the *PreviewInteractiveShape* base class. Two shapes are defined in the application. The first is a reticle that indicates the current beam position and can be repositioned when enabled. The second is a focus area rectangle, which defines a region of interest for acquisition. This rectangle can be moved and resized by the user.

The shapes are rendered by the preview widget (*_StemPreviewImage*, red area in Figure 3.4) and handle their own interaction logic, including mouse events and cursor behavior. High-level control, such as visibility and position updates, is managed by the *StemPreviewController*.

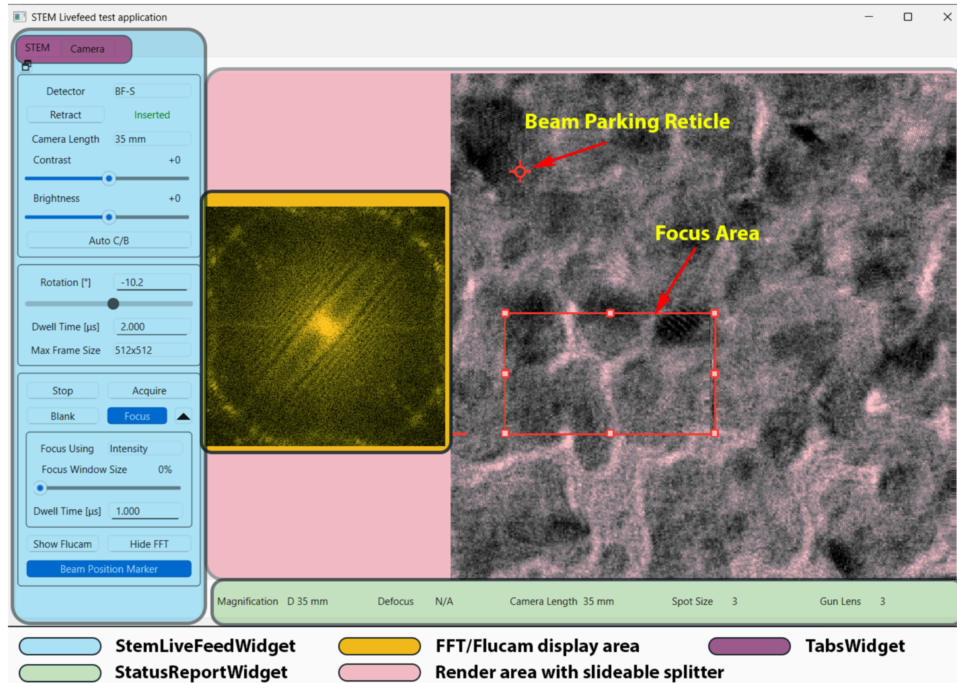


Figure 3.4: This figure shows, how is the STEM Live Feed Widget separated into smaller widgets.

3.3.2 Backend

The backend of the STEM live feed is responsible for controlling the acquisition lifecycle, retrieving data from the microscope, transforming incoming data into an application-level buffer, and delivering incremental updates to the frontend. The implementation follows a layered architecture that separates high-level application logic from low-level microscope communication.

The backend is divided into three main layers:

- **Python coordination layer** – manages acquisition control, threading, and interaction with the user interface.
- **C++ acquisition-task layer** – provides a stable abstraction for configuring and executing acquisition tasks, including shared buffer management and update notifications.
- **TEM_OMP streaming layer** – handles direct communication with the microscope imaging pipeline and provides access to live raster data.

This separation ensures that the user interface is not directly coupled to low-level hardware communication, improving maintainability and enabling independent development of individual layers.

A central component of the backend is the *StemAcquisitionTask* abstraction. It provides a consistent interface for configuring acquisition parameters, controlling task execution, accessing the raw image buffer, and subscribing to incremental updates. Internally, it connects to the TEM_OMP streaming layer and translates low-level detector events into application-level notifications.

The backend implementation builds upon existing functionality provided by the underlying STEM acquisition framework. Operations such as detector selection, rotation configuration, and camera-length adjustment are handled through already available STEM task classes. These capabilities are reused directly by the *StemAcquisitionTask*, ensuring consistency with existing microscope control mechanisms.

In addition to continuous acquisition, the backend also supports single-frame acquisition. The *Acquire* operation invokes the acquisition task in a single-shot mode, reusing the same stem task class pipeline as other controls. This approach utilizes already implemented functionality, without creating duplication in acquisition tasks.

On the Python side, the acquisition task is accessed through a wrapper and executed within a dedicated worker thread. This design prevents blocking of the graphical user interface and enables smooth real-time interaction.

3.3.3 Flucam Integration

The Flucam preview is integrated into the STEM live feed as an auxiliary visualization and is not acquired through the newly implemented STEM backend. Instead, it reuses the existing camera live-feed subsystem and mirrors its output into the STEM user interface.

The integration is performed at the UI level. The combined live-feed widget creates both the STEM live-feed widget and the standard camera live-feed widget. The STEM widget then receives a reference to the camera live-feed component and subscribes to image updates emitted by its viewport. These updates are provided in the form of QImage objects, which are forwarded to the *StemPreviewController* and displayed as the auxiliary preview.

The data flow for Flucam can be summarized as follows:

$$\textit{LivefeedWidget} \rightarrow \textit{ImageFeedWidget} \rightarrow \textit{image_updated}(\textit{QImage}) \rightarrow \\ \textit{StemLiveFeedWidget} \rightarrow \textit{StemPreviewController}$$

This design ensures that the STEM widget does not duplicate acquisition functionality and instead relies on already processed image data from the camera subsystem.

The Flucam acquisition itself is implemented in the existing camera live-feed module. The *ImageFeedWidget* component is responsible for image acquisition. It creates a dedicated QThread and moves an AcquisitionWorker object into this thread. The worker connects to the acquisition service, configures the selected camera, and continuously retrieves image frames. Each acquired frame is processed and emitted as a signal, which is received in the GUI thread, converted into a QImage, and propagated through the image_updated signal.

As a result, the acquisition pipeline operates asynchronously, ensuring that image acquisition does not block the user interface. The STEM widget passively listens to these updates and displays them without participating in the acquisition process itself.

In addition to the continuous acquisition thread, the camera live-feed module uses separate Python threads for blocking hardware operations, such as insertion or retraction of the camera or screen. These operations are executed independently so that they do not delay the main application flow.

QTimer is a class in the Qt framework used for scheduling and handling timed events in applications. It allows the application to execute a function (slot) either once after a specified delay or repeatedly at fixed time intervals, without blocking the main program flow. QTimer operates within the Qt event loop and is commonly used for tasks such as periodic updates, delayed execution, or coordinating asynchronous operations.

In this project, a QTimer is used within the *camera live-feed widget* to trigger periodic repainting of the viewport. The timer does not control the acquisition itself, as image acquisition runs in a separate worker thread. Instead, it ensures that the displayed image is refreshed at a consistent rate.

3.4 STEM Acquisition

This section describes the processing chain used to obtain, transport, and display live STEM detector data. The implementation is based on continuous raster acquisition, where the detector produces image data progressively during scanning.

Instead of transferring only complete frames, the system propagates partial updates corresponding to modified image regions.

3.4.1 Acquisition Control

Acquisition control is coordinated by the Python control layer and executed by the backend task. User actions, such as starting or stopping acquisition, changing dwell time, adjusting frame size, or enabling a focus area, are translated into acquisition requests by the control logic.

These requests are forwarded to a runtime component that executes acquisition operations in a background thread. The acquisition task applies the current configuration when starting or restarting the scan.

When acquisition parameters are modified during active streaming, the system does not immediately restart the acquisition for each individual change. Instead, a delayed restart mechanism is used to group rapid updates. This is implemented using single-shot *QTimer* instances, which postpone the restart operation for a short period. This approach prevents excessive task restarts during continuous user interaction.

3.4.2 Detector Data Acquisition

Detector data acquisition is implemented in the C++ *StemAcquisitionTask*. This component is responsible for creating the underlying scan task, preparing the STEM detector context, and configuring the acquisition for raster scanning.

After the acquisition starts, the system waits until valid live-image metadata becomes available, in particular, the image dimensions and pixel type. These parameters are provided by the underlying TEM_OMP streaming session and reflect the actual configuration of the detector. Once the metadata is received, the internal raw buffer is rebuilt to match the layout of the incoming data. This ensures that the buffer representation is always consistent with the detector output.

The raw buffer stores the current state of the live STEM image and is shared with higher layers of the application. It can be accessed either as a full snapshot or through partial row-based copying, depending on the needs of the processing pipeline (more details about the buffer can be seen in Figure 3.1, and how it can be accessed is explained in 3.1.1).

The buffer is updated incrementally as new data arrives from the detector. Instead of repeatedly copying complete frames, only the mod-

ified rows or rectangular regions are transferred and written into the buffer. These updates are triggered by streaming callbacks originating from the *TEM_OMP* layer and are processed asynchronously within the acquisition task.

This incremental update strategy significantly reduces memory bandwidth usage and avoids unnecessary data duplication. As a result, it enables efficient real-time streaming of large STEM images while maintaining the responsiveness of the application.

3.4.3 Data Stream

The data stream originates in the *TEM_OMP* layer through the raster streaming session abstraction. This session connects to the detector output and subscribes to update events generated by the imaging pipeline.

When new data becomes available, update callbacks provide information about the modified image region. These low-level events are received by the acquisition task, which copies the updated data into its internal raw buffer.

After processing, the acquisition task emits higher-level notifications containing the updated image region and the frame identifier. These notifications are propagated through the Python wrapper to the runtime layer and subsequently to the frontend by means of queued Qt signals.

The overall data flow is therefore incremental and event-driven. The system reacts to detector-generated updates and transfers only the data that has changed, thereby avoiding unnecessary full-frame processing.

3.4.4 Rendering

Rendering is based on partial frame updates rather than full-frame redrawing. After receiving an update notification, the newly delivered data are inserted into the current frame representation, and the corresponding image region is marked as modified.

A frame-processing component tracks these changes and schedules a deferred display update. During this update, only the affected image

regions are converted and rendered. The rendering itself is performed using buffered image widgets that support efficient partial repainting.

In addition to incremental rendering, contrast and brightness adjustments are applied during the visualization stage. These adjustments are implemented as a display mapping and do not modify the raw acquisition data.

The raw STEM data stored in the internal buffer remain unchanged. Instead, the intensity values are transformed during conversion to the display buffer. This transformation is implemented using NumPy, which enables efficient vectorized processing. The processed values are written into a separate rendering buffer, thereby preserving the original data.

The mapping operates on normalized intensity values in the range $[0, 1]$. Brightness shifts the intensity relative to the midpoint, while contrast scales the deviation from that midpoint. The transformation is defined as

$$\text{mapped} = \big((\text{normalized} - 0.5 - \text{brightness}_{\text{shift}}) \cdot \text{contrast}_{\text{factor}}\big) + 0.5$$

The result is then clipped to the valid range and converted to the display format.

Automatic Contrast and Brightness In automatic mode, the displayed intensity range is derived directly from the image data. A subsampled version of the raw frame is used to estimate the minimum and maximum intensity values. The image is then normalized so that the sampled minimum maps to black and the sampled maximum maps to white:

$$\text{normalized} = \frac{\text{value} - \text{low}}{\text{high} - \text{low}}$$

After normalization, the values are clipped to the valid display range and converted to the output format. This approach improves the visibility of image details without modifying the underlying acquisition data. When automatic mode is enabled, the manual contrast and brightness controls are disabled.

3.4.5 Threading Model

Threading is essential for maintaining application responsiveness. Acquisition is executed in a dedicated QThread managed by the runtime component. A worker object is moved into this thread and communicates with the user interface through queued Qt signals.

This design ensures that acquisition-related operations, including task creation, start, stop, and update handling, do not block the GUI thread.

On the C++ side, shared resources such as the raw buffer and acquisition state are protected by synchronization primitives. This is necessary because update callbacks may arrive asynchronously from the *TEM_OMP* layer.

3.4.6 Python Wrapper and GIL Handling

The Python wrapper ensures safe interaction between Python and C++. Blocking operations such as starting, restarting, or stopping acquisition are executed with the Python Global Interpreter Lock (GIL) released, allowing other Python code to continue execution while the underlying C++ operation is in progress.

When update callbacks are delivered from C++ to Python, the wrapper ensures that the Python execution context is properly acquired before invoking Python code. This prevents deadlocks and supports safe communication across the Python-C++ boundary.

3.4.7 Use of QTimer

Timers are used to coordinate asynchronous operations and improve responsiveness. Their main roles include

- delaying acquisition restart after parameter changes,
- batching display updates,
- deferring layout recalculation,
- handling bounded waiting for worker shutdown.

Short-delay timers group rapid successive changes into a single operation, while zero-delay timers schedule work for the next event-loop iteration, allowing efficient processing of multiple updates without unnecessary immediate recomputation.

4 Testing

Testing played a crucial role during the development of the Live Feed STEM widget. Multiple approaches were used, including validating control flow, developing proof-of-concept applications, and conducting extensive manual testing in a simulated environment. The testing process was designed to verify both the correctness of the implementation and the performance of the system under realistic conditions.

4.1 Validation Test Applications

During development, two smaller validation applications were created to verify specific parts of the system. These applications were primarily used for proof-of-concept testing of acquisition behavior, event handling, and inter-thread communication.

A first validation application was used to analyze how acquisition events are emitted and what information they provide. This was essential for the correct implementation of the acquisition task. The second test application focused on streaming and notification handling across threads, which helped identify issues such as missing notifications or invalid data.

These validation applications were removed after development, once their purpose was fulfilled.

4.2 Demo Application

A demo application was developed and used during the early stages of development, before real acquisition functionality was available. Its purpose was to validate the interface design and overall data flow.

Instead of real detector data, the application generated synthetic STEM data, which can be seen in Figure 4.1, and was used to simulate acquisition. This allowed verification of the streaming pipeline, notification delivery, and rendering performance. The scan results from acquisition confirmed that the rendering pipeline was sufficiently efficient in streaming and rendering generated data, making the originally

proposed circular buffer unnecessary. Instead, a bounded queue was sufficient for handling update notifications.

The demo application was also connected to STEM control elements, providing early feedback on user interaction, detector insertion, and status updates.

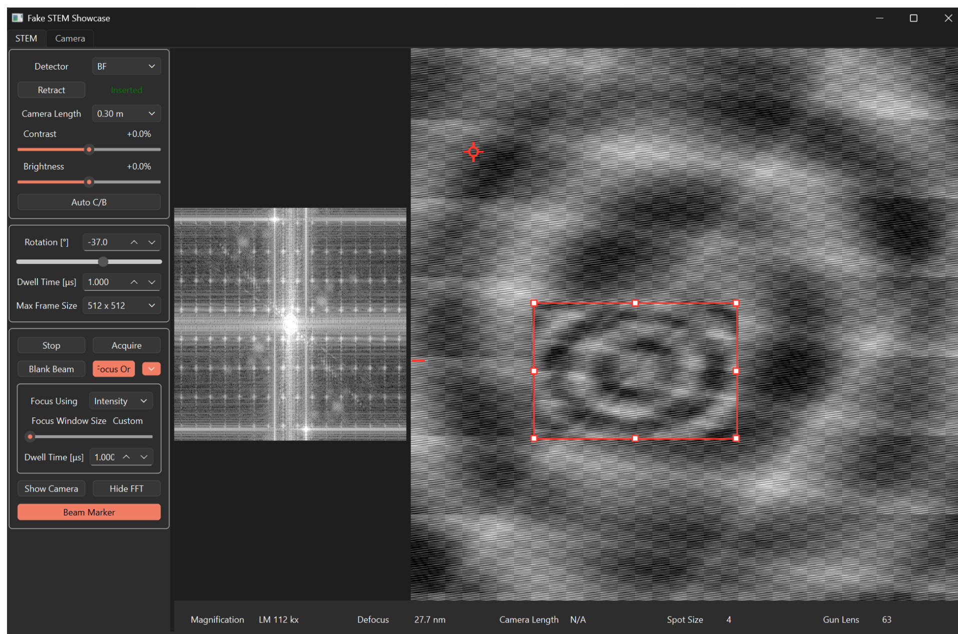


Figure 4.1: This image shows the demo application widget.

4.3 Unit Tests and Mocks

Unit tests and mock implementations were used to verify individual components in isolation, without requiring access to real hardware or the full acquisition pipeline. In the unit tests, mocked versions of the interfaces have been used, so real tasks don't have to be created.

This approach enabled testing of acquisition logic, Python wrappers, and data flow under controlled conditions. It improved reliability, supported regression testing, and allowed validation of edge cases that would be difficult to reproduce on a real system.

4.4 CLI Test Application

After the core functionality was implemented, a dedicated test application was developed for STEM acquisition. This application provides access to all acquisition controls, but is operated through a command-line interface rather than a graphical user interface.

The application allows users to select a detector, configure dwell time and resolution, and choose between full-frame and region-based acquisition. It also processes update events and provides diagnostic information, such as buffer contents, updated regions, frame identifiers, and buffer checksums. This functionality enables verification of correct data propagation and buffer updates, as you can see in Figure 4.2.

All parameters can be modified during runtime without restarting the application, closely reflecting the behavior of the final system. Unlike the main application, this test tool remains available for future testing and development.

```
Choose acquisition mode: camera / stem / quit [camera/stem/quit]: stem
Available STEM detectors:
0: BF-S, available=1, enabled=1
1: DF-S, available=1, enabled=0
2: HAADF, available=1, enabled=1
Select detector index [0]: 0
Use selected area [0=no, 1=yes] [0]: 0
Dwell time [s] [0]: 0.000001
Image width [0]: 512
Image height [0]: 512
Settings applied: size=512x512, dwellTime=1e-06, rotationRad=0 (ignored in v1), readoutArea=<full-frame>
Command [start|stop|restart|buffer|analyze|status|save|detector|settings|quit]: start
Starting acquisition...
Status: running=1, observedUpdates=0, observedLatestFrame=0
Buffer: 512x512, stride=1024, bytesPerPixel=2, sampleBytes=524288, checksum=1147117824754123651
PixelStats: type=Unsigned16, min=0, max=0, nonZero=0/262144
RowTail16[row=0]=0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
RowTail16[row=128]=0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
RowTail16[row=256]=0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
RowTail16[row=384]=0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
RowTail16[row=511]=0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
BytePreview[lastRowLast64,row=511]=0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
BufferLayout: width=512, height=512, strideBytes=1024, bytesPerPixel=2, expectedRowBytes=1024
Command [start|stop|restart|buffer|analyze|status|save|detector|settings|quit]: buffer
Buffer: 512x512, stride=1024, bytesPerPixel=2, sampleBytes=524288, checksum=4316053928802520298
PixelStats: type=Unsigned16, min=0, max=65535, nonZero=260733/262144
RowTail16[row=0]=45430,47298,40290,39937,45102,40587,41609,41729,41894,40452,41371,41260,41386,41272,41741,30648
RowTail16[row=128]=30803,33498,27945,29552,30453,33362,34042,34655,34082,33285,34686,33408,32750,33705,32922,30220
RowTail16[row=256]=20497,21501,21050,20604,22840,26837,25951,26325,26668,26492,26839,26209,26640,27152,27533,17182
RowTail16[row=384]=32245,27953,26961,29501,37388,33404,32930,31389,33129,32435,33238,31509,33276,33726,30939,30471
RowTail16[row=511]=23636,20773,24168,15574,16799,13744,14036,12142,12441,13902,12835,13483,13422,13740,14308,26602
BytePreview[lastRowLast64,row=511]=0x0c,0x70,0xc7,0x65,0x07,0x58,0x93,0x5f,0xfe,0x5c,0x45,0x52,0x75,0x54,0xf6,0x6b,0x9e,
0x5a,0x96,0x49,0x44,0x6c,0x6d,0x56,0x5b,0x4d,0x6e,0x53,0x79,0x43,0x84,0x4d,0x54,0x5c,0x25,0x51,0x68,0x5e,0xd6,0x3c,0x9f,
0x41,0xb0,0x35,0xd4,0x36,0x6e,0x2f,0x99,0x30,0x4e,0x36,0x23,0x32,0xab,0x34,0x6e,0x34,0xac,0x35,0xe4,0x37,0xea,0x67
BufferLayout: width=512, height=512, strideBytes=1024, bytesPerPixel=2, expectedRowBytes=1024
Command [start|stop|restart|buffer|analyze|status|save|detector|settings|quit]: stop
Stopping acquisition...
Status: running=0, observedUpdates=48, observedLatestFrame=3
```

Figure 4.2: This image shows the test app running in the command line.

4.5 Simulator

A significant part of testing was performed using a simulator running in a virtual machine. The simulator provides a functional TEM/STEM environment, including the necessary server infrastructure for microscope control.

While the simulator supports most functionality, the generated image data are largely static. This limitation restricts testing of certain features, such as rotation or camera-length changes, which depend on dynamic image content. Despite this, the simulator proved sufficient for identifying the majority of implementation issues.

Development was performed on a local machine, with changes deployed to the simulator for validation, which can be seen in Figure 4.3. This approach allowed faster compilation of C++ code while still enabling testing in an environment close to the real system.

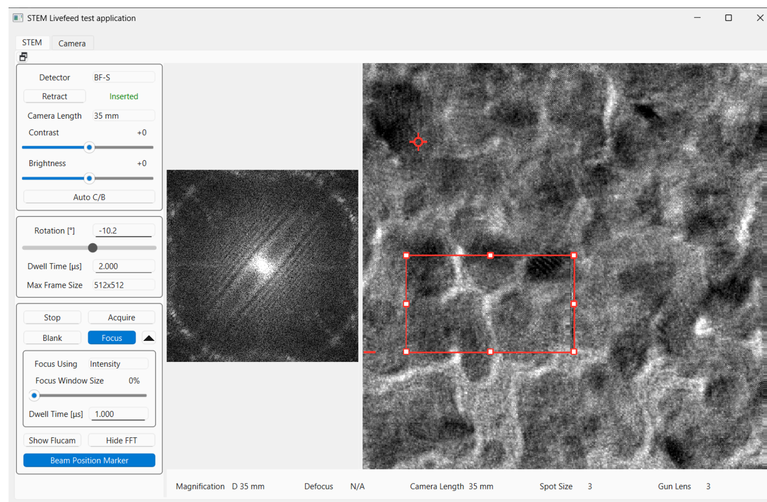


Figure 4.3: This image shows the widget in the simulator.

4.6 System Testing

The final implementation was tested on a real microscope system, specifically the Talos Alpha 6 equipped with STEM functionality. This environment provides real detector data and enables full validation of the application under operational conditions, as shown in Figure 4.4.

The testing was conducted in collaboration with a dedicated team of testers. All relevant parameters and their responsiveness were evaluated. The streaming pipeline was verified, and the acquired image results were compared with those obtained from the Velox application to ensure consistency and correctness.

System testing confirmed the correct functionality of key features, including rotation, camera-length adjustment, and beam positioning. It also verified proper handling of real detector data and overall system stability.

Only minor issues were identified during this phase. These included occasional inconsistencies in line update handling, where outdated notifications did not invalidate the update indicator on lines, and synchronization issues related to detector blanking not being fully integrated with the polling mechanism. Most of these issues had already been identified during simulator-based testing, and their impact on the final system was limited.

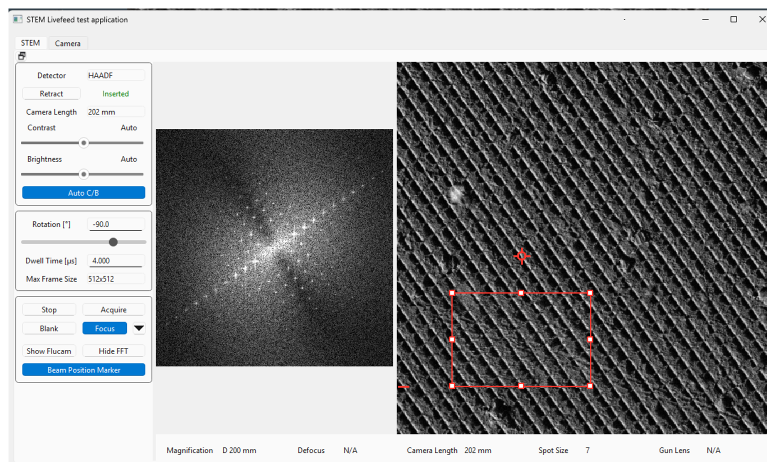


Figure 4.4: This image shows the widget in a real microscope system.

4.7 Internal Usability Validation

The usability of the STEM live-view tool was verified through an internal evaluation performed by a team member familiar with microscope workflows and the Sherpa environment. The evaluation focused on usability, performance, and workflow integration. Overall, the findings indicate a highly positive user experience, with the evaluator reporting that the widget behaved as expected throughout typical calibration workflows. The interaction with the tool was described as smooth and reliable, exceeding expectations when compared to the current workflow implementation.

From a functional perspective, the study confirmed that all essential acquisition parameters required for microscope operation were available and operating correctly. Parameters such as dwell time, resolution, camera length, and rotation were fully accessible and responsive during testing. In addition, the live-feed performance was considered comparable to Velox, with the evaluator noting smooth responsiveness and stable operation during real-time use. These observations suggest that the widget provides sufficient technical performance for practical microscopy tasks without introducing workflow limitations or usability concerns.

The qualitative feedback also emphasized the strengths of the user interface and the benefits of workflow integration. The interface was described as intuitive, clearly organized, and easy to navigate during routine operation. A particularly valuable aspect identified in the evaluation was the integration of STEM and Flucam into a single application, especially the support for continuous Flucam acquisition prior to scanning, which improved STEM alignment workflows. Based on the overall usability, responsiveness, and integration capabilities, the evaluator concluded that the widget would be suitable for daily use within the Sherpa environment.

5 Conclusion

The final product of this thesis is the STEM Live Feed widget, which has been successfully implemented, validated, and tested in both real and simulated microscope environments. The system provides reliable real-time streaming of acquisition data based on user-defined parameters and allows direct interaction with the microscope during operation.

The implemented solution features a modular, layered architecture that separates user interface logic, acquisition control, and low-level hardware communication. This design allows effective handling of high-throughput data streams while maintaining the responsiveness of the user interface. The use of incremental, event-driven data updates, combined with partial rendering, greatly improves performance and reduces unnecessary processing overhead.

The integration of the widget into the existing plugin suite enables developers and calibration engineers to perform STEM-related tasks within a unified environment. This eliminates the need for external applications and simplifies workflows during microscope preparation and calibration. In addition, the reuse of existing components, such as the Flucam live feed, demonstrates the flexibility of the proposed solution and reduces implementation complexity.

Several technical challenges were addressed during development, including thread-safe communication between Python and C++, efficient buffer management, and correct handling of asynchronous data streams. The use of a dedicated worker thread, proper synchronization mechanisms, and careful management of the Python Global Interpreter Lock ensured stable operation without blocking the user interface.

The solution was thoroughly tested using multiple approaches, including simulated environments, validation applications, and deployment on a real microscope system. These tests validated the correctness of the implementation and demonstrated that the system is suitable for practical use.

Future work may include extending the widget with additional analytical tools, improving visualization capabilities, and further optimizing performance for higher-resolution acquisitions. In addition,

the widget can be integrated with procedural instructions used during calibration workflows, allowing users to perform calibration steps and observe results within a single unified environment.

Overall, the developed STEM Live Feed widget provides a robust and extensible foundation for real-time acquisition and visualization, contributing to improved efficiency and usability of microscope calibration workflows.

A An appendix

A.1 Demo STEM Showcase Application

The demo STEM showcase application is a standalone PyQt6-based application designed to demonstrate the functionality of the STEM live-feed widget. The project is intentionally separated from AutoStar, TEM services, and internal company-specific Python and C++ interfaces in order to provide an isolated demonstration environment.

The application contains two primary components:

- A demo STEM tab providing:
  - live search,
  - single acquisition,
  - FFT preview,
  - focus ROI,
  - brightness and contrast adjustment,
  - demo detector insertion,
  - demo camera-length selection.
- A demo camera tab featuring:
  - an animated preview stream,
  - stream mirroring into the STEM widget.

The backend implementation includes a native C++ STEM acquisition module with callback-based subscription support.

A.1.1 Requirements

The application requires the following software components:

- Python 3.11 or newer,
- CMake 3.20 or newer,

- a C++17-compatible compiler,
- Python packages:
  - PyQt6,
  - numpy.

A.1.2 Build Process

The project can be built using the following commands:

```
python -m pip install -r requirements.txt
Remove-Item -Recurse -Force build -ErrorAction
   SilentlyContinue
cmake -S . -B build
cmake --build build --config Release
```

The compiled native module is generated in:

```
src/demo_stem_app/acquisition
```

After compilation, the application can be launched directly from the project source tree.

A.1.3 Execution

The application can be started using:

```
python main.py
```

Bibliography

1. WILLIAMS, David B. *Transmission Electron Microscopy: A Textbook for Materials Science*. Boston: Springer US, 2009. ISBN 978-0-387-76501-3.
2. LIU, Jingyue; COWLEY, JM. High-resolution scanning transmission electron microscopy. *Ultramicroscopy*. 1993, vol. 52, no. 3-4, pp. 335–346.
3. EGERTON, R.F. *Physical Principles of Electron Microscopy* [online]. Cham: Springer International Publishing, 2016 [visited on 2026-04-19]. ISBN 978-3-319-39876-1 978-3-319-39877-8. Available from DOI: 10.1007/978-3-319-39877-8.
4. SHINDO, Daisuke; OIKAWA, Tetsuo. *Analytical electron microscopy for materials science*. Springer Science & Business Media, 2013.
5. MIELAŃCZYK, Łukasz; MATYSIAK, Natalia; KLYMENKO, Olesya; WOJNICZ, Romuald. Transmission electron microscopy of biological samples. *Transm Electron Microscope-Theory Appl*. 2015, pp. 193–236.
6. RAI, Raghaw S; SUBRAMANIAN, Swaminathan. Role of transmission electron microscopy in the semiconductor industry for process development and failure analysis. *Progress in crystal growth and characterization of materials*. 2009, vol. 55, no. 3-4, pp. 63–97.
7. TAN, Susheng. Transmission electron microscopy: Applications in nanotechnology. *IEEE Nanotechnology Magazine*. 2020, vol. 15, no. 1, pp. 26–37.
8. LUDWIG REIMER, Helmut Kohl. *Transmission Electron Microscopy: Physics of Image Formation*. Vol. 36 [online]. New York, NY: Springer New York, 2008 [visited on 2026-04-19]. Springer Series in Optical Sciences. ISBN 978-0-387-40093-8 978-0-387-34758-5. Available from DOI: 10.1007/978-0-387-40093-8.

9. KONDO, Yukihito; OHI, Kimio; ISHIBASHI, Yu; HIRANO, Haruo; HARADA, Yoshiyasu; TAKAYANAGI, Kunio; TANISHIRO, Yasumasa; KOBAYASHI, Kunio; YAGI, Katsumichi. Design and development of an ultrahigh vacuum high-resolution transmission electron microscope. *Ultramicroscopy*. 1991, vol. 35, no. 2, pp. 111–118.

10. GOLDSTEIN, Joseph I.; NEWBURY, Dale E.; MICHAEL, Joseph R.; RITCHIE, Nicholas W.M.; SCOTT, John Henry J.; JOY, David C. *Scanning Electron Microscopy and X-Ray Microanalysis* [online]. New York, NY: Springer New York, 2018 [visited on 2026-04-19]. ISBN 978-1-4939-6674-5 978-1-4939-6676-9. Available from DOI: 10.1007/978-1-4939-6676-9.

11. CREWE, AV; EGGENBERGER, DN; WALL, J; WELTER, LM. Electron gun using a field emission source. *Review of Scientific Instruments*. 1968, vol. 39, no. 4, pp. 576–583.

12. BRONSGEEST, Merijntje. *Physics of Schottky electron sources: theory and optimum operation*. CRC Press, 2014.

13. SCIENTIFIC, Thermo Fisher. *Velox Software for Transmission Electron Microscopy*. Available also from: https://www.thermofisher.com/cz/en/home/electron-microscopy/products/software-em-3d-vis/velox-software.html.

14. COMPANY, The Qt. Qt for Python. [N.d.]. Available also from: https://doc.qt.io/qtforpython-6/.

15. NUSSBAUMER, Henri J. The fast Fourier transform. In: *Fast Fourier transform and convolution algorithms*. Springer, 1981, pp. 80–111.

16. HARRIS, Charles R.; MILLMAN, K. Jarrod; WALT, Stéfan J. van der; GOMMERS, Ralf; VIRTANEN, Pauli; COURNAPEAU, David; WIESER, Eric; TAYLOR, Julian; BERG, Sebastian; SMITH, Nathaniel J.; KERN, Robert; PICUS, Matti; HOYER, Stephan; KERKWIJK, Marten H. van; BRETT, Matthew; HALDANE, Allan; RÍO, Jaime Fernández del; WIEBE, Mark; PETERSON, Pearu; GÉRARD-MARCHANT, Pierre; SHEPPARD, Kevin; REDDY, Tyler; WECKESSER, Warren; ABBASI, Hameer; GOHLKE, Christoph; OLIPHANT, Travis E. Array programming with

NumPy. *Nature*. 2020, vol. 585, no. 7825, pp. 357–362. Available from DOI: 10.1038/s41586-020-2649-2.