Mouse and Gaze Tracking Application: Architecture and Technical Documentation

ESTIA Gaze Project

2025-05-08

# Mouse and Gaze Tracking Application: Architecture and Technical Documentation

## Table of Contents

1. [Introduction](#introduction)
2. [Project Structure](#project-structure)
3. [Core Components](#core-components)
4. [Features and Functionality](#features-and-functionality)
5. [Design Decisions](#design-decisions)
6. [Technical Implementation](#technical-implementation)
7. [Cross-Platform Compatibility](#cross-platform-compatibility)
8. [Libraries and Dependencies](#libraries-and-dependencies)
9. [Best Practices](#best-practices)
10. [Future Improvements](#future-improvements)

## Introduction

The Mouse and Gaze Tracking Application is a comprehensive software solution designed to track, analyze, and visualize mouse movements and eye gaze patterns on a computer screen. This system provides researchers, UX professionals, and accessibility specialists with powerful tools to understand how users interact with digital interfaces through both physical (mouse) and visual (gaze) inputs.

This document explains the technical architecture, design decisions, and implementation details of the application, providing a complete reference for developers who want to understand, modify, or extend the system.

## Project Structure

The application follows a modular architecture with clearly separated concerns. The project’s folder structure is organized as follows:

MouseTracking/  
├── mouse\_tracker.py # Core mouse tracking functionality  
├── gaze\_tracker.py # Core gaze tracking functionality  
├── sync\_tracker.py # Synchronized tracking controller  
├── mouse\_analytics.py # Data analysis and visualization  
├── mouse\_tracker\_gui.py # Basic GUI for mouse tracking  
├── sync\_tracker\_gui.py # Advanced GUI for synchronized tracking  
├── run\_sync\_tracker.py # Easy-to-use launcher script  
├── requirements.txt # Project dependencies  
├── MouseTracker\_Explained.md # User-friendly explanation  
├── GazeTrackingIntegration.md # Gaze tracking documentation  
├── README.md # Project overview and setup instructions  
├── LICENSE # MIT License  
├── setup\_env.sh # Environment setup script for Unix/macOS  
├── setup\_env.bat # Environment setup script for Windows  
├── models/ # Directory for ML models  
│ └── shape\_predictor\_68\_face\_landmarks.dat # Face landmarks model  
└── mouse\_data/ # Directory for tracking data  
 ├── mouse\_events\_\*.csv # Mouse event data (CSV format)  
 ├── mouse\_events\_\*.json # Mouse event data (JSON format)  
 ├── gaze\_events\_\*.csv # Gaze event data (CSV format)  
 ├── gaze\_events\_\*.json # Gaze event data (JSON format)  
 ├── sync\_events\_\*.csv # Synchronized events data (CSV format)  
 ├── sync\_events\_\*.json # Synchronized events data (JSON format)  
 └── \*\_report\_\*.html # Generated HTML reports

### Key Files and Their Roles

| File | Purpose |
| --- | --- |
| mouse\_tracker.py | Core module for tracking mouse movements, clicks, and scrolls |
| gaze\_tracker.py | Core module for tracking eye gaze using webcam or Tobii devices |
| sync\_tracker.py | Coordinating module that combines mouse and gaze data |
| mouse\_analytics.py | Data processing and visualization |
| mouse\_tracker\_gui.py | Basic graphical interface for mouse tracking |
| sync\_tracker\_gui.py | Advanced interface for synchronized tracking |
| run\_sync\_tracker.py | User-friendly launcher with setup assistance |

## Core Components

The application consists of five core components that work together to provide the complete tracking and analysis functionality:

1. **Mouse Tracker**: Captures and records mouse movements, clicks, and scroll events.
2. **Gaze Tracker**: Tracks eye movements using webcam, Tobii hardware, or simulated data.
3. **Synchronization System**: Coordinates data from both tracking systems and calculates relationships.
4. **Analytics Engine**: Processes recorded data and generates insights through visualizations.
5. **User Interface**: Provides easy-to-use controls and real-time visualizations.

### Component Interaction Diagram

┌─────────────────┐ ┌─────────────────┐  
│ Mouse Tracker │ │ Gaze Tracker │  
└────────┬────────┘ └────────┬────────┘  
 │ │  
 ▼ ▼  
 ┌──────────────────────────────┐  
 │ Synchronization Layer │  
 └──────────────┬───────────────┘  
 │  
 ▼  
 ┌──────────────────────────────┐  
 │ Analytics Engine │  
 └──────────────┬───────────────┘  
 │  
 ▼  
 ┌──────────────────────────────┐  
 │ User Interface │  
 └──────────────────────────────┘

## Features and Functionality

The application provides a comprehensive set of features for tracking and analyzing user interactions:

### Mouse Tracking Features

* Real-time tracking of mouse coordinates (x, y positions)
* Detection and recording of mouse clicks (left, right, middle buttons)
* Tracking of mouse scroll events
* Calculation of movement metrics (distance, speed, acceleration)
* Identification of click patterns and dwell times

### Gaze Tracking Features

* Support for three tracking modes:
  + **Webcam Mode**: Uses the computer’s webcam and computer vision techniques
  + **Tobii Mode**: Interfaces with professional Tobii eye tracking hardware
  + **Dummy Mode**: Generates simulated data for testing and development
* Detection of fixations (when gaze stays in one place)
* Detection of saccades (rapid eye movements between fixations)
* Detection of blinks
* Measurement of pupil size and dilation (hardware permitting)
* Confidence scoring for tracking accuracy

### Synchronized Tracking Features

* Temporal alignment of mouse and gaze events
* Calculation of distance between mouse cursor and gaze point
* Detection of attention patterns (whether eyes follow cursor)
* Percentage calculation of attention match

### Analytics Features

* Generation of heatmaps showing activity concentration
* Trajectory visualization of movement paths
* Distance plotting over time
* Statistical analysis of interaction patterns
* HTML report generation with interactive visualizations

### User Interface Features

* Start/stop controls for tracking sessions
* Real-time visualization of tracking data
* Mode switching between different tracking methods
* Clean Session feature for resetting data without restarting
* Report generation controls
* Settings configuration

## Design Decisions

Several key design decisions shaped the architecture and implementation of the application:

### Modular Architecture

The application uses a modular design with separate components for each major functionality area. This approach:

* Enhances maintainability by isolating concerns
* Allows independent development and testing of each component
* Facilitates extension with new features
* Enables selective use of components based on needs

### Multiple Gaze Tracking Options

The decision to support multiple gaze tracking methods (webcam, Tobii, and dummy) was driven by:

1. **Accessibility**: Webcam-based tracking works with standard hardware most users already have.
2. **Professional Use Cases**: Tobii integration provides high-precision tracking for research settings.
3. **Development and Testing**: Dummy mode allows testing without specialized hardware.

### Data Storage Strategy

The application saves data in both CSV and JSON formats to:

* Support easy import into spreadsheet applications (CSV)
* Enable programmatic processing with structured data (JSON)
* Maintain data integrity with minimal storage overhead
* Provide flexibility for various analysis workflows

### Real-time Processing with Multithreading

The application uses multithreaded processing to:

* Maintain UI responsiveness during tracking
* Capture events at high frequency without dropping data
* Process analytics computations without affecting tracking
* Support simultaneous webcam processing and mouse tracking

### User Interface Design

The GUI was designed with these priorities:

1. **Simplicity**: Clear, intuitive controls for core functions
2. **Real-time Feedback**: Immediate visualization of tracking data
3. **Flexibility**: Multiple visualization options for different analysis needs
4. **Discoverability**: Clearly labeled functions with sensible defaults

## Technical Implementation

### Event-Driven Architecture

The application uses an event-driven architecture for mouse and gaze tracking:

1. Hardware/OS events are captured through listeners
2. Events are processed, enriched, and timestamped
3. Processed events are stored in memory and periodically saved to disk
4. The UI updates based on event streams

This approach allows: - Low-latency processing of events - Minimal impact on system performance - Accurate temporal recording

### Data Models

The application uses dataclasses to represent events:

@dataclass  
class MouseEvent:  
 timestamp: float  
 event\_type: str # 'move', 'click', 'scroll'  
 x: float  
 y: float  
 button: Optional[str] = None  
 pressed: Optional[bool] = None  
 dx: Optional[float] = None  
 dy: Optional[float] = None  
  
@dataclass  
class GazeEvent:  
 timestamp: float  
 event\_type: str # 'fixation', 'saccade', 'blink'  
 x: float = 0.0 # Normalized x coordinate (0-1)  
 y: float = 0.0 # Normalized y coordinate (0-1)  
 duration: Optional[float] = None  
 pupil\_size: Optional[float] = None  
 confidence: float = 1.0  
 screen\_x: Optional[int] = None # Actual screen coordinate  
 screen\_y: Optional[int] = None # Actual screen coordinate  
  
@dataclass  
class SyncEvent:  
 timestamp: float  
 event\_type: str  
 mouse\_x: Optional[float] = None  
 mouse\_y: Optional[float] = None  
 gaze\_x: Optional[float] = None  
 gaze\_y: Optional[float] = None  
 distance: Optional[float] = None  
 normalized\_distance: Optional[float] = None  
 # Additional fields omitted for brevity

### Computer Vision Implementation

The webcam-based gaze tracking uses several computer vision techniques:

1. **Face Detection**: Using dlib’s frontal face detector
2. **Facial Landmark Detection**: Using a pre-trained model to locate 68 landmarks
3. **Eye Region Extraction**: Focusing on landmarks 36-47 (the eye regions)
4. **Pupil Detection**: Using thresholding and contour detection
5. **Gaze Estimation**: Mapping pupil position to screen coordinates
6. **Blink Detection**: Calculating eye aspect ratio to detect blinks

# Simplified example of pupil detection  
def detect\_pupils(eye\_region):  
 # Convert to grayscale  
 gray = cv2.cvtColor(eye\_region, cv2.COLOR\_BGR2GRAY)  
   
 # Apply Gaussian blur  
 blurred = cv2.GaussianBlur(gray, (5, 5), 0)  
   
 # Threshold to find dark regions (pupils)  
 \_, threshold = cv2.threshold(blurred, 50, 255, cv2.THRESH\_BINARY\_INV)  
   
 # Find contours  
 contours, \_ = cv2.findContours(threshold, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)  
   
 # If contours found, find the largest (likely the pupil)  
 if contours:  
 pupil = max(contours, key=cv2.contourArea)  
 return pupil  
 return None

### Multithreading Implementation

The application uses Python’s threading module for concurrent processing:

1. **Main UI Thread**: Handles the user interface and user interactions
2. **Mouse Tracker Thread**: Listens for and processes mouse events
3. **Gaze Tracker Thread**: Processes webcam frames or receives Tobii data
4. **Periodic Save Thread**: Saves accumulated data to disk at intervals

Thread synchronization is handled through: - Mutex locks (threading.Lock) - Event signaling (threading.Event) - Thread-safe data structures

### Data Visualization

The application uses Matplotlib for data visualization:

1. **Heatmaps**: Generated using 2D histograms and Gaussian filtering
2. **Trajectory Plots**: Using line and scatter plots for movement paths
3. **Distance Plots**: Time-series plots of mouse-gaze distance
4. **Real-time Plotting**: Using FigureCanvasTkAgg for embedding in the GUI

## Cross-Platform Compatibility

The application is designed to be cross-platform compatible, working on:

* **Windows**: 10 and 11
* **macOS**: Catalina (10.15) and later, including Apple Silicon Macs
* **Linux**: Ubuntu 20.04+, Fedora, and other major distributions

### Platform-Specific Considerations

#### Windows

* Uses Windows API via pynput for mouse input monitoring
* Takes advantage of DirectShow for webcam access
* Accounts for DPI scaling variations
* Compatible with Windows Terminal or cmd.exe

#### macOS

* Handles permissions requirements for Input Monitoring
* Accounts for Retina display scaling
* Works with AVFoundation for camera access
* Includes workarounds for potential permissions issues
* Provides specific setup for Apple Silicon (M1/M2/M3) Macs

#### Linux

* Adapts to X11 or Wayland display servers
* Uses V4L2 for webcam access
* Accounts for desktop environment variations
* Handles display server permissions

### Cross-Platform Implementation

To ensure cross-platform compatibility, the application:

1. Uses Python’s platform-independent libraries where possible
2. Implements platform detection and conditional code paths
3. Provides platform-specific setup scripts
4. Handles differences in file paths and environment variables
5. Uses relative paths within the application
6. Includes appropriate error handling for platform-specific failures

# Example of platform-specific implementation  
def get\_screen\_resolution():  
 """Get the screen resolution in a platform-agnostic way"""  
 if platform.system() == "Darwin": # macOS  
 try:  
 # Use Tkinter for macOS  
 import tkinter as tk  
 root = tk.Tk()  
 screen\_width = root.winfo\_screenwidth()  
 screen\_height = root.winfo\_screenheight()  
 root.destroy()  
 return (screen\_width, screen\_height)  
 except:  
 # Fallback for macOS  
 return (1920, 1080)  
 elif platform.system() == "Windows":  
 try:  
 # Use ctypes for Windows  
 import ctypes  
 user32 = ctypes.windll.user32  
 screen\_width = user32.GetSystemMetrics(0)  
 screen\_height = user32.GetSystemMetrics(1)  
 return (screen\_width, screen\_height)  
 except:  
 return (1920, 1080)  
 else: # Linux and others  
 try:  
 # Try using Xlib for Linux  
 from Xlib import display  
 d = display.Display()  
 screen = d.screen()  
 screen\_width = screen.width\_in\_pixels  
 screen\_height = screen.height\_in\_pixels  
 return (screen\_width, screen\_height)  
 except:  
 return (1920, 1080)

## Libraries and Dependencies

The application relies on several key Python libraries, each chosen for specific capabilities:

### Core Libraries

| Library | Purpose | Why Selected |
| --- | --- | --- |
| pynput | Mouse event monitoring | Cross-platform, low-level event access |
| OpenCV (cv2) | Computer vision processing | Industry standard, comprehensive feature set |
| dlib | Face detection and landmark detection | High accuracy, pre-trained models available |
| NumPy | Numerical operations and array manipulation | Performance, foundational for scientific computing |
| Matplotlib | Data visualization and plotting | Comprehensive plotting capabilities |
| Pandas | Data analysis and manipulation | Powerful data structures for analysis |
| Tkinter | GUI framework | Cross-platform, included in Python standard library |

### Additional Libraries

| Library | Purpose | Rationale |
| --- | --- | --- |
| threading | Multithreaded processing | Standard library, reliable threading primitives |
| dataclasses | Structured data representation | Clean syntax, built-in serialization |
| logging | Application logging | Standard, configurable logging |
| json/csv | Data serialization and storage | Standard formats for interoperability |
| argparse | Command-line argument parsing | Standard library, robust option handling |
| datetime | Timestamp handling | Standard library, comprehensive date/time support |

### Library Selection Criteria

Libraries were selected based on:

1. **Reliability**: Stable, well-maintained packages
2. **Performance**: Efficient processing for real-time applications
3. **Cross-platform support**: Works across Windows, macOS, and Linux
4. **Community support**: Active development and issue resolution
5. **Documentation**: Comprehensive documentation and examples
6. **Licensing**: Compatible open-source licenses

## Best Practices

The application implements several software development best practices:

### Code Organization

* **Separation of Concerns**: Each module has a clear, specific responsibility
* **Encapsulation**: Implementation details are hidden within classes
* **Interface Segregation**: Clean, focused interfaces for each component
* **Single Responsibility Principle**: Classes and functions do one thing well

### Error Handling

* **Comprehensive Exception Handling**: All operations that can fail are wrapped in try/except
* **Graceful Degradation**: Falls back to alternative modes when preferred options aren’t available
* **User Feedback**: Clear error messages explain issues and suggest solutions
* **Logging**: Detailed logs for debugging and audit trails

# Example of robust error handling  
try:  
 # Attempt to use webcam  
 cap = cv2.VideoCapture(0)  
 if not cap.isOpened():  
 logger.error("Could not open webcam. Falling back to dummy mode.")  
 self.mode = 'dummy'  
 self.\_init\_dummy\_tracker()  
 return  
 # Webcam opened successfully  
 logger.info("Webcam-based gaze tracker initialized")  
except Exception as e:  
 logger.error(f"Error initializing webcam tracker: {e}")  
 logger.error("Falling back to dummy tracker mode")  
 self.mode = 'dummy'  
 self.\_init\_dummy\_tracker()

### Resource Management

* **Proper Cleanup**: Resources like file handles and camera connections are closed
* **Memory Management**: Large datasets are processed in chunks to manage memory usage
* **Thread Management**: All threads are properly joined when stopping tracking
* **Configuration Management**: Settings are stored in consistent locations

### Testing and Validation

* **Fallback Mechanisms**: Dummy mode provides testing without hardware dependencies
* **Graceful Error Handling**: Robust handling of unexpected conditions
* **Input Validation**: User inputs are checked for validity
* **Cross-platform Testing**: Application tested on multiple operating systems

### Documentation

* **Code Documentation**: Docstrings explain purpose and usage
* **Type Hints**: Python type annotations improve clarity and IDE support
* **User Documentation**: Comprehensive guides for users
* **Architecture Documentation**: This document explains system design

## Future Improvements

Several potential improvements could enhance the application:

1. **Machine Learning Integration**:
   * Use ML to identify patterns in mouse and gaze relationships
   * Implement attention prediction models
   * Add anomaly detection for unusual interaction patterns
2. **Advanced Visualization**:
   * 3D visualizations of interaction patterns
   * Interactive exploration of captured data
   * Comparative visualization of multiple sessions
3. **Extended Platform Support**:
   * Mobile device support for touch and gaze tracking
   * Tablet support for stylus and gaze correlation
   * VR/AR integration for immersive environment analysis
4. **Integration Capabilities**:
   * API for real-time access from other applications
   * Browser extension for web-specific tracking
   * SDK for embedding in other research tools
5. **Enhanced Accessibility Features**:
   * Gaze-controlled cursor for motor-impaired users
   * Attention-based UI adaptation
   * Assistive technology research tools

## Conclusion

The Mouse and Gaze Tracking Application provides a powerful, flexible framework for capturing and analyzing human-computer interaction through both physical and visual inputs. Its modular design, robust implementation, and cross-platform compatibility make it suitable for a wide range of research, usability testing, and accessibility applications.

The architectural decisions emphasize reliability, usability, and extensibility, creating a foundation that can be built upon for specialized use cases while providing immediately valuable functionality out of the box.