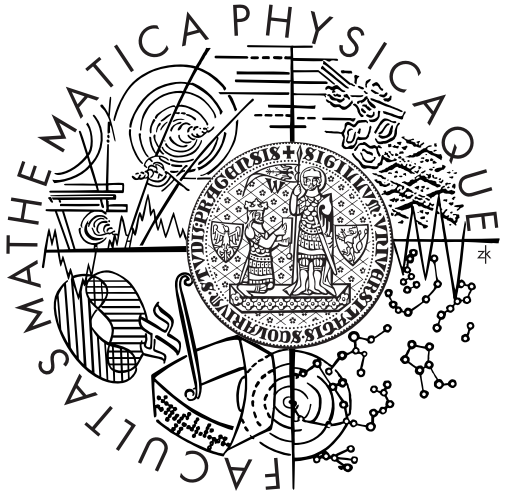
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Milan Burda

Mouse Gestures

Department of Software and Computer Science Education

Thesis supervisor: RNDr. Josef Pelikán
Study program: Computer Science, Programming

2008

I would like to thank my supervisor RNDr. Josef Pelikán, for many valuable suggestions and for all the time spent at consultations. I would also like to thank my friend Andrej Mikulík, for practical use of my application during the development, useful feature requests and bug reports. I am grateful to my friend Dominik Kosorín, for correcting the spelling of the thesis. Finally, thanks to my sister Diana Burdová and my friend Michal Čermák, for mental support during hard times while writing the thesis.

I hereby certify that I wrote the thesis myself using only the referenced sources. I agree with publishing and lending of the thesis.

Prague, 8 August, 2008 Milan Burda

Contents

Title: Mouse Gestures
Author: Milan Burda
Department: Department of Software and Computer Science Education
Supervisor: RNDr. Josef Pelikán
Supervisor's e-mail address: Josef.Pelikan@mff.cuni.cz

Abstract: In the presented work, we design and implement a mouse gesture recognition application. The program integrates transparently with the operating system, thus allowing existing unmodified Windows applications to be controlled by gestures. In an editor provided, the user is able to define a custom set of gesture patterns that the program automatically learns to recognize. The recognition algorithm is based on a preprocessing phase and two different gesture classifiers: back-propagating artificial neural network and k-nearest neighbors. The user is allowed to configure both general and application specific gesture mappings. These specify the commands to be triggered by the individual gestures. Several new features and improvements have been proposed for further development.

Keywords: mouse gestures, gesture recognition, neural network, k-nearest neighbors, C++

Název práce: Mouse Gestures
Autor: Milan Burda
Katedra (ústav): Kabinet software a výuky informatiky
Vedoucí bakalářské práce: RNDr. Josef Pelikán
e-mail vedoucího: Josef.Pelikan@mff.cuni.cz

Abstrakt: V predloženej práci navrhneme a implementujeme aplikáciu na rozpoznávanie tzv. mouse gestures. Transparentná integrácia do operačného systému umožňuje gestami ovládať existujúce aplikácie pre Windows bez akýchkoľvek úprav. Užívateľ si v poskytnutom editore definuje vlastnú množinu vzorov, ktoré sa program naučí automaticky rozpoznávať. Rozpoznávací algoritmus je založený na fáze predprípravy a dvoch rôznych klasifikátoroch: neurónová sieť a k-najbližších susedov. Užívateľ si môže nadefinovať všeobecné, ako aj špecifické mapovanie gest pre rôzne aplikácie. Tieto mapovania definujú príkazy aktivované daným gestom. Pre budúci vývoj bolo navrhnutých niekoľko nových vlastností a vylepšení.

Klíčová slova: mouse gestures, rozpoznávanie gest, neurónová sieť, k-najbližších susedov, C++

Chapter 1

Introduction

This paper deals with the analysis, design, and implementation of a program, which would allow existing Windows applications, without code modification or recompilation, to be controlled by user defined mouse gestures.

1.1 Motivation

On small touch screen devices without traditional keyboard, pen input involving handwriting recognition has become a standard. It is a very efficient input method when the recognition accuracy is high enough. However, a similar concept can be also partially adopted in traditional desktop applications or even the whole windowing system. Mouse gestures are an additional way to provide user input using a pointing device such as mouse or touch-pad.

On Windows, Opera web browser has probably been the first application which introduced built-in in support for mouse gestures [17]. Firefox and Internet Explorer do not support mouse gestures natively, however several extensions providing gesture support exist [13, 18]. Mouse gestures are quite popular among users as they allow certain frequent tasks to be performed in a much more convenient way. For example, to go back in the browser history, it is much faster just to move the mouse cursor a few points to the left while holding the right button, than to click the back button in the tool-bar.

Application specific gesture support is a nice feature. However, a general solution working across all the applications is much more desirable. No such completely free to use application can be found on the Internet, surprisingly. Despite the fact that mouse gestures are not a new idea. This lack of an appropriate existing solution is the main motivation behind taking up the challenge of developing a cross-application mouse gesture recognition engine for Windows.

1.2 Existing solution

Still, one such application called StrokeIt [34] exists. However, it is not completely free and the source code is not available for download. Only the software development kit (SDK) for plug-in developers can be downloaded. Moreover, it has several quite serious limitations, including:

- the last version 0.95 has been released in 2005, no signs of active development can be observed since
- the user interface is old-fashioned and not sufficiently intuitive
- the way the program paints the gesture feedback line on the screen is unreliable and incompatible with the Desktop Window Manager (DWM) [10] introduced in Windows Vista

1.3 Gesture definition

Before moving on, it is important to define the term **gesture**. A gesture is a sequence of mouse movements performed while holding a specified **trigger button**. Let us call this a **regular gesture**. Wheel scrolling while holding the trigger button is also considered a gesture; this will be called a **wheel gesture**.

From the programmer's perspective, a gesture is a sequence of events. Gesture start event is the first one, followed by a series of mouse movements and/or wheel scrolling events, terminated by a finish event. The timespan between the individual mouse movements is not taken into account. Hence we can also represent the gesture as a vector of either relative or absolute mouse cursor positions. The absolute position representation is more convenient and thus will be used for our purposes.

The user, on the other hand, considers the whole gesture as one input event. Using a gesture is essentially very much like striking a keyboard shortcut consisting of multiple key presses.

1.4 Goals

Our task is to create a mouse gesture recognition application, called Universal Gestures. The program has to support seamless integration with the operating system, in order to enable mouse gesture support in all existing Windows applications. Gestures will be detected by intercepting mouse input: button

clicking, cursor movement, and wheel scrolling. The gesture will be triggered by holding a specified so-called trigger button, configured to the right button by default. The user will be able to disable the gesture detection temporarily for the given moment, by holding a specific key or by clicking a mouse button different from the gesture toggle button. The user must also be able to use the toggle button to perform clicking and drag & drop operations. A detailed description of the gesture detection and the associated settings are provided in the Gesture capture library chapter.

The program as well as the system integration should be as stable and reliable as possible, to avoid any potential negative impact on the running applications. Problematic applications like games, where mouse gestures interfere with the way the mouse is used, or those with built-in gesture support, can be added to an exclusion list. Gesture processing will be ignored completely in these applications.

One of the most important objectives is the ability to recognize advanced, user-defined gestures. The user will be able to define a custom set of patterns, in an editor provided. The applications will then learn to recognize these patterns automatically. Moreover, each user will be able to add a set of training samples to each pattern, to increase the recognition accuracy. Simple gestures should be supported as well, to satisfy users that do not demand more sophisticated gesture recognition.

The user will be able to assign individual gestures to trigger commands, provided by the application. Gesture mappings are divided into two groups, the defaults, which are available in any application, and program specific mappings. Applications will be identified by the executable file path, which is simple, though reliable in most cases. The application should include at least the following set of commands:

- basic window control - minimize, maximize / restore, close, resize
- special window attributes - always on top, transparency
- send a general or an application specific command message
- emulate keyboard shortcuts to invoke application commands
- control the Universal Gestures application itself
- execute any user defined application with given command-line arguments
- open a selected special folder, such as the Computer or the Control Panel
- switch between running applications

Universal Gestures will be a resident application running in the background. It will be accessible by a tray icon located in the system notification area.

Right clicking the icon will display the main menu, which provides access to all features of the application, such as the configuration windows. User friendliness is important. The user interface and the configuration should be intuitive and easy to use.

The minimum operating system version supported will be a 32-bit edition of Windows XP, which is the most commonly used version of Windows at present. Older versions, including Windows 2000, are now obsolete as they lack a lot of functionality introduced in later versions. Windows Vista, as well as Windows Server 2003 and 2008 will be supported too. The application as a whole will not be portable, as the system integration involves use of platform specific functions. However, due to the modular design, many parts of the program will be platform independent.

The application will be developed in C++, to achieve good performance and keep the resource use low. The actual tools employed will be described in more detail in the Technologies used chapter. The program configuration will be stored in the convenient XML format. Plug-in support will allow further extension of the functionality.

1.5 Challenges

In addition to laborious tasks such as the user interface design and implementation, we expect having to deal with a few challenges. A way to detect gestures performed in any application reliably has to be found. The development of an accurate gesture recognition algorithm is also expected to be a non-trivial task. On the other hand, we expect to gain a lot of knowledge and experience while developing the application.

1.6 Program structure

Modularity is one of the main goals that have been set. Modular architecture, when properly designed, yields many benefits:

- possible porting to another operating system in the future will be easier
- the source code will be well organized and more readable
- debugging will be easier, as self-contained functional units can be developed and tested separately
- plug-in system can be implemented, allowing 3rd party developers to extend the application's core functionality

Our application will be divided into the following parts:

- gesture capture library
- main application
- modules - gesture recognizers & commands

These parts are described individually in more detail in the following chapters. Design, interface, and implementation information is included.

Chapter 2

Gesture recognition

Our main objective is to design an accurate and efficient gesture recognition algorithm, able to recognize user defined gesture patterns.At the same time, a simple four-direction recognizer is necessary. The design of the latter proved to be a rather trivial task. The principle lies in the detection of cursor movement in one of the supported directions. The movement is only registered when the distance between the actual cursor position and the previous base point exceeds a minimum threshold value. To prevent diagonal motion from being detected as a sequence of horizontal or vertical movements, the angle of the line between the actual position and the base point must be within a specified range. This can be seen in figure 2.1. When a new section is about to be added to the resulting gesture, the previous one is checked, as consequent section must be different.

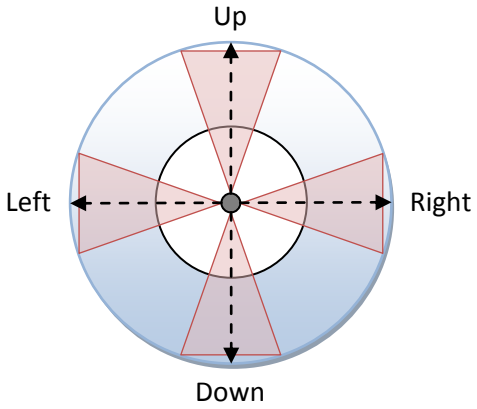


Figure 2.1: Angle range in different directions

Thus, from now on, we only deal with more sophisticated gesture recognition methods, which classify gestures into groups given by the user defined gesture patterns.

The goal of the gesture recognition algorithm is to determine, which gesture pattern, if any, corresponds to the given gesture. As already defined in the Introduction, the program treats gestures as directed sequences of mouse cursor positions, represented by vectors of points. The time span between the individual movements is not considered significant and therefore is not taken into account. As it was observed in [2], gestures are expected to be simple shapes, which can be drawn on the screen easily, in order to be useful. It should be possible to repeat them multiple times with sufficient similarity. Suitable gesture patterns include straight lines, simple geometric shapes such as triangle, circle, square, etc., letters of the alphabet, which can be painted with a single stroke.

The output of the algorithm is a regular gesture, identified by the name of the corresponding pattern. Patterns are defined by their base shape, mainly used by the user interface for graphical representation. However, user entered samples assigned to each pattern are more important. They are crucial for successful gesture recognition, as the variable shape and size of the performed gestures cannot be expressed by a single definition. In case there are not enough pattern samples present, it is possible to emulate them by adding noise to the base pattern shape. However, this trick is unable to substitute real, user-entered samples.

We decided to search for a suitable gesture recognition algorithm, instead of taking the risk of failure, while trying to invent a completely new solution, and to avoid reinventing the wheel. Several different sources have been consulted. However, most of the papers deal with different kinds of gestures, such as hand gestures. The results of the research done in [2] are the most interesting. The algorithm proposed in this paper is simple, yet powerful. Hence, our solution will be based on this algorithm.

2.1 Algorithm principle

The algorithm consists of two separate phases: preprocessing and classification, as outlined in figure 2.2.

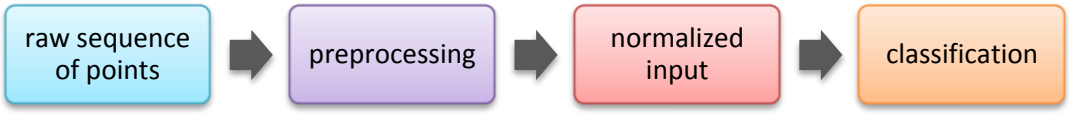


Figure 2.2: Recognition process overview

The purpose of the preprocessing phase is to produce appropriate input for the actual gesture classification. The length of the raw sequence of points varies from gesture to gesture. Therefore, the input has to be transformed into a

vector of fixed length, called the **key point count**. The algorithm, described in [2], breaks the gesture down into a sequence of characteristic points that define significant changes in the shape of the gesture. It works as follows:

- iterate through the list of points in the input sequence. Skip the first and the last point
- remove the point from the result, if:
  - the angle between the consequent segments is close to 180°
  - the distance from the last point kept is less than a given threshold

The remaining points now define the shape of the gesture. However, the number of points can still be different from the requested amount. The polyline has to be interpolated to achieve the given **key point count**, by splitting the longest segments and joining the shortest ones. An overview of the algorithm can be seen in figure 2.3.

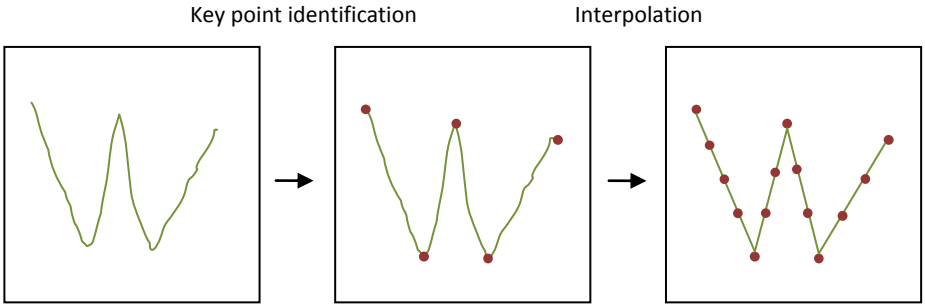


Figure 2.3: Overview of the preprocessing algorithm

Now when the input is normalized, we can proceed with the classification. Two different gesture classifiers will be used. They both use the same preprocessing algorithm. However, the input representation is different.

2.2 Neural network

A standard artificial neural network [4] with following properties will be used:

- three layers - input, hidden, output
- log-sigmoid activation function
- back-propagation training
- variable learning rate with momentum

The data delivered to the neural network is encoded as a sequence of cosines and sinuses of the angles between the subsequent segments [7]. Hence, the number of inputs equals twice the amount of lines in the normalized shape. Each output corresponds to a single recognizable gesture pattern. Thus, the number of outputs is the same as the size of the pattern list. Therefore, every time a new gesture is added to or removed from the list, the neural network has to retrain from scratch.

To recognize a gesture, we transform the normalized sequence of points into the proposed input format and propagate it through the neural network. We find the maximum value, which signals the corresponding gesture pattern, when it is above a defined threshold value. On overview of the neural network can be seen in figure 2.4.

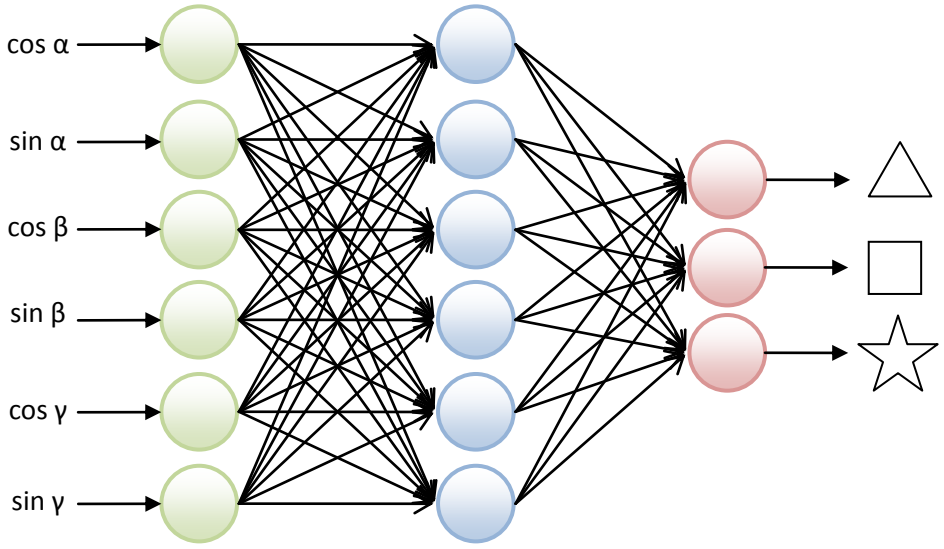


Figure 2.4: Neural network overview

The training is performed by repeating the standard back-propagation algorithm. Preprocessed and transformed pattern samples are used as the training input. Expected output samples are constructed by taking a vector filled with zeros except for a one on the index, which is assigned to the given gesture pattern. The training process is finished when either the error rate reaches the target value, or the number of cycles exceeds the maximum count.

2.3 K-nearest neighbors

The k-nearest neighbors [15] is the second available gesture classifier. Big advantage over the neural network is the lack of the learning phase. The idea of the algorithm is very simple. To recognize a gesture, we compare it with all pattern samples from the list. By applying a specified distance

measure, we get **K** nearest objects. The samples are grouped according to the corresponding gesture pattern. The winner is the pattern with more than half of the nearest samples. In case there is no winner, the recognition is reported to be unsuccessful.

The distance measure will be calculated as the sum of Euclidean distances of the corresponding points. To make the coordinates comparable, the shapes of the gesture as well as all the pattern samples have to be normalized to the same coordinate space after the preprocessing phase. The geometric center of the shape will be in the middle of the coordinate space. An example of a normalized shape can be seen in figure 2.5.

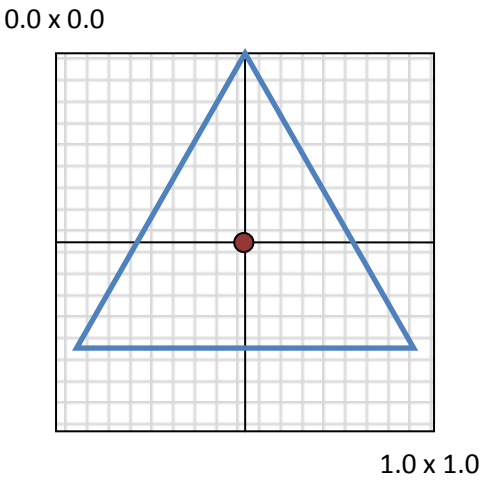


Figure 2.5: Normalized shape example

The results of the algorithm depend on the value of **K**. If the number of samples per pattern is the same, there is no problem. However, when it is not, the **K** has to be calculated somehow. We will use the most frequent number of pattern samples. Patterns, which do not contain the required number of samples, will not be recognized. On the other hand, the patterns containing at least the sufficient number of patterns, will only use the first **K** patterns. The other possible solution would be to take the minimum number of pattern samples. However, this would handicap "well behaved" patterns.

Chapter 3

Technologies used

The application has been developed in C++, as already mentioned in the Introduction. The main reasons for choosing C++ were high performance and the ability to write low-level system code easily. After selecting the programming language, the development tools and libraries had to be decided. C++ is an extremely widespread language offering dozens of development environments and thousands of different libraries and toolkits.

3.1 Visual Studio 2008 + Feature Pack

The choice of the integrated development environment (IDE) was straightforward. Microsoft Visual Studio is a quality product, probably the best choice for Windows development. A student license for non-commercial purposes is available through the MSDN Academic Alliance program. The C++ compiler offers very good performance of the generated binary code. MSDN provides extensive documentation of the Windows API, which cannot be avoided. The program has to interact with and control other user's applications in ways not offered by higher-level frameworks.

Smart pointers, included in Technical Report 1 (TR1) [35], have been used. TR1 is an addition to the Standard Template Library (STL). By using the **shared_ptr** template class, the majority of explicit object destruction could be eliminated. This helps to avoid potential memory leaks and problems related to passing of object instances. There are two ways how to get TR1. Either from the Visual C++ 2008 Feature Pack [24], which has been released recently and will be a part of the upcoming Service Pack 1. The second possibility is to use Boost, a collection of free high quality C++ libraries for various purposes.

3.2 Graphical user interface toolkit

The decision which graphical user interface (GUI) toolkit to use was harder. Nevertheless, the choice that has been made proved to be the right one. The main requirements imposed upon the toolkit were:

- native look & feel, matching dedicated Windows applications as closely as possible
- clean object-oriented design allowing simple extensibility
- well documented, examples and sample applications should be provided
- must be free for use in non-commercial applications; no particular type of license is demanded
- a cross-platform framework is a big plus, as it allows potential porting to Linux or Mac OS X in the future
- extra functionality such as network and XML support would be appreciated, to eliminate the need to use other libraries with a different naming conventions, data types, etc.

There are many widget toolkits and frameworks available for C++. The major ones include: **Qt**, **wxWidgets**, **GTK+**, **Microsoft Foundation Classes (MFC)**, **Windows Template Library (WTL)** and **Windows Forms**. All of them have both pros and cons:

- **GTK+** [14] does not provide the level of conformance to the Windows user interface guidelines [27] as users would expect. Widget rendering differs from the native look & feel. Buttons in dialog boxes are ordered differently and they use non-standard icons and labels. A custom file open/save dialog, which is confusing and less capable, is used instead of the standard one provided by Windows. One of the limitations could also be the use of UTF-8 encoded strings, which have to be converted into the native UTF-16 representation used by the native Windows API functions
- **wxWidgets**' [39] design dates several years back, when C++ compilers were much less capable than they are today. Therefore, the library lacks support for modern C++ features, such as templates or the STL library. The code is not exception safe; the programmer has to be careful to avoid program crashes. wxWidgets do not implement their widgets from scratch, but rather wrap the native ones provided by Windows. This approach offers high performance and native look on one hand, but on the other hand, it limits the possibilities. Moreover, the library does not have a full-featured painting system with vector graphics support. This is essential in order to create appealing user interface elements

- **MFC** [16] is for Windows only. It has a long tradition and vast amounts of documentation. Nevertheless, it is not included with the freely available Express edition of Visual Studio. It is quite heavyweight, large amounts of code have to be written in order to create the user interface. The recently released Visual C++ 2008 Feature Pack would make MFC a bit more attractive, if released before design process has started

- **WTL** [37] is a lightweight alternative to the MFC, an extension of the Active Template Library (ATL). It is an open-source project from Microsoft, however without any technical support. There is practically no documentation available, only some tutorials and applications written in WTL can be found on the Internet. The whole library consists of header files only. The application does not have to be linked to any library. However, heavy use of templates slows down the compilation significantly. Like MFC, this framework is for Windows only

- **Windows Forms** [36] and the .NET Framework offer a comprehensive set of cleanly designed classes covering all aspects of application programming. Visual Studio supports development of Windows Forms applications natively. The **System.Drawing** provides rich painting abilities, but the underlying GDI+ library, itself a wrapper of the GDI, is rather slow. **System.Xml** provides excellent XML support, etc. The biggest disadvantage and the main reason for refusing this toolkit is the dependency on the .NET Framework and managed code. The consequences are increased memory usage and slower performance, especially on lower-end computers

The **Qt toolkit** [32, 33], which has the most benefits has been chosen:

- Qt is cross-platform and free for non-commercial use

- modern C++ features including templates, exceptions, precompiled headers are supported

- Qt is modular; the application is only linked with the modules providing the necessary functionality. Many modules with consistent interfaces are provided beyond the widget system. For example, comprehensive XML support, network connectivity, database access, HTML engine (WebKit), scripting and multimedia playback (Phonon) modules are included

- Qt mimics the native platform look on Windows and Mac OS X as closely as possible by using the platforms' default theme rendering APIs. The Linux version renders the controls on its own, as there are no native system widgets

- Qt extends the limited C++ object model [29] by using a custom preprocessor tool invoked automatically by the build system

- QObject instances are organized into a tree-like hierarchy. They are deleted automatically in their parent's destructor. Larger widget hierarchies can be constructed without having to worry about the memory management
- QObject interaction is based on a powerful signal/slot mechanism, which is an application of the Observer design pattern [1]
- Qt provides native support for Unicode strings. QStrings are encoded in UTF-16, hence the performance is not being degraded unnecessarily by character encoding conversions
- Qt provides its own implementation of container classes. Compared to the STL library, they have more useful methods provided. Conversion functions between Qt and STL containers are provided. Iterators to Qt container classes can be passed to STL functions
- all Qt data types including containers, bitmaps, and others are implicitly shared, using a private implementation and reference count internally. This allows them to be passed as function arguments and returned by value without copying. Thread-safety is provided too
- Qt classes can be easily extended by standard C++ object inheritance; plug-ins can be written to provide support for new image formats, database servers, etc. The plug-in system can be employed effortlessly in user applications too
- Qt has a powerful 2D and 3D graphics support. 2D graphics is vector based with integrated support for Scalable Vector Graphics (SVG), Portable Document Format (PDF), printing, etc. 3D graphics is accelerated by OpenGL
- Qt has a comprehensive documentation. Along with a complete reference, many examples, overviews, and tutorials are available
- the knowledge gained while developing an application in Qt should be a good investment into the future. Many companies developing cross-platform solutions in C++ use the Qt toolkit. As it is being developed by a commercial company, paying customers receive full technical support. The commercial version also offers full Visual Studio integration, which is not provided by the competing toolkits. An alternative IDE has to be used for wxWidgets or GTK+

The most recent version 4.4.1 of the Qt library has been used. A custom patch has been applied, containing bug fixes and feature additions, which have not yet been resolved by Trolltech. The problems were particular to the Windows version of Qt. The following Qt modules have been used: QtCore, QtGui, QtNetwork and QtXml.

Chapter 4

Gesture capture library

As mentioned in the Challenges section of the Introduction, the first and fundamental problem that we have to solve is how to detect gestures in any Windows application. The reason for the decision to separate the gesture detection code into a separate library was not only to comply with the modularity goal. It also proved to be an inevitable consequence of the final mouse input capturing solution.

4.1 Mouse input detection & filtering

According to the gesture definition, a gesture consists of elementary mouse input events. Therefore:

- we need to detect or be notified of the basic mouse input events: key press/release, mouse cursor movement, and wheel scrolling
- we must be able to prevent the affected program from receiving these events while the gesture is being performed. Otherwise, two actions will be performed. The application's default response to mouse dragging as well as a possible command associated with the gesture

The traditional mouse input mechanism in form of Qt events or direct handling of Windows messages cannot be used. The reason is simple; mouse messages are only sent to the window hovered by the mouse cursor. Something more sophisticated has to be used.

While investigating the possibilities, several approaches have been considered, ranging from the simplest ones to the most advanced. The following ideas seem plausible:

1. the most naive idea is the user of a timer (**QTimer** or **SetTimer**), invoking a handler routine in periodic intervals. This procedure would determine the mouse cursor position (**GetCursorPos**) and check whether the individual mouse buttons are pressed (**GetAsyncKeyState**). This solution is very inefficient and unreliable. Events can be lost between two consequent polling intervals. However, the biggest and unavoidable shortcoming is the inability to filter mouse events, rendering this option useless

2. the second idea is to use **Raw Input** API [23] introduced in Windows XP. After registering for input notifications by calling the **RegisterRawInputDevices** function, our application would receive **WM_INPUT** events, generated directly by the mouse. However, there is the same critical problem that we are not able to filter these events

3. **DirectInput** [19] is another option, although not a good one. It is primarily meant to be used in games. When initialized in the exclusive mode, our program would receive all the mouse input events, while no other application gets any. This is a bad idea. All the standard mouse messages supposed to be send by Windows would have to be generated manually and routed to the corresponding windows

4. **hooks** [26] are the right choice. When set, a specified callback function will be called on every mouse input event. The function decides whether to swallow the message or pass it to the affected window. The first available hook type - **WH_MOUSE_LL** may seem to be the best as it does not involve a library injection. The callback function is located in the hooking application. Nevertheless, this means that context is being switched between the hooked and hooking application on every event. This makes this hook type quite inefficient. If used, every single mouse cursor movement would cause two extra context switches. Therefore, the second alternative, **WH_MOUSE** hook type has been chosen as the final solution. A library containing the callback function is injected into the address space of all applications running in the current login session. The function is called directly before the message reaches the window procedure, hence no context switch is involved

5. **subclassing** [25] has also been considered as a potential alternative to hooks. By replacing the original window procedure, we can process all messages including the mouse events and decide whether to pass them to the previous window procedure, effectively implementing a filtering mechanism. But this approach is very problematic, a library containing our window procedure has to be loaded into the target process by some kind of code injection technique [38, 8]. Separate procedure variants have to be written for both ANSI and Unicode window types. The window procedure has to be replaced for all windows separately. Another hook type - **WH_SHELL**, would have to be used to detect new window creation

4.2 Design

Hook libraries are usually written as thin wrapper around the hook callback function, passing events of interest to the main application. The main reason for this approach is to keep the hook library as simple as possible. The code of the library is being executed in the context of other process instances. Any bug in its code, can cause the application to crash or behave incorrectly. This minimalistic approach is worthwhile, as the stability of user applications is not threatened. However, context switches still occur, when the library communicates with its parent application.

In order to maximize efficiency, at the expense of potential instability if not done properly, part of the gesture processing is implemented directly in the mouse hook library. In fact, the whole gesture detection algorithm, based on elementary mouse input events is implemented. The gesture capture library and the mouse hook library are one and the same. This is possible, as the gesture capture library has been designed as a separate dynamically linked module in the program structure. Thus it can be injected to all applications. In fact, the integration simplifies the decision the hook callback function has to make, whether to filter the particular mouse event. The decision can be made directly according to program state without having to perform another callback to the actual gesture-processing library. The overall amount of context switches is minimized. Mouse movement events are only being reported when a gesture is in progress. When the mouse is being moved without pressing the toggle button, no further processing happens outside the hook callback function.

Modularity of the gesture capture library simplifies potential porting to another operating system. The whole library providing the same interface would be implemented using the platform specific facilities. In addition, the library could be used to implement a different gesture recognition application that would not have to deal with gesture capturing details. To keep the library as small and versatile as possible, Qt cannot be used; all the code has to be written using standard Windows API only. In order to maintain stability, the library has to be tested thoroughly.

The gesture capture library has to meet the following criteria:

- application behavior should not be influenced negatively, when gesture recognition is enabled. Problematic windows should be detected and ignored automatically
- a callback to the main application to determine whether the affected application is to be ignored completely has to be made
- the toggle button can be selected by the user. All possible mouse buttons have to be supported. Both horizontal and vertical wheel scrolling should

be supported too

- the user must be able to use the toggle button to perform a single click, without starting a gesture
- gestures should be activated only after the cursor has been moved at least a given distance, so-called activation distance, from the starting point. It is designed to prevent accidental activation when a single click has been intended, but the user moved the cursor a few pixels
- holding a specified ignore key on the keyboard suppresses gesture activation
- the gesture while in progress can be cancelled by clicking an opposite button. Left and right buttons are opposite. However, when the middle or extra buttons are configured to be the toggle button, the left button is considered opposite
- when the cursor has not been moved for a specified amount of time, the gesture is cancelled automatically. The timeout feature can be disabled
- possible mouse button swapping configurable in Windows mouse control panel has to be supported correctly
- only one application in the login session can use the library at the same time to avoid conflicts

4.3 Interface

The gesture catcher library contains a main class called **GesturesHook** with respect to the hook library nature. The class is a singleton with only one globally unique instance possible.

GestureEvent structure:

- **eventType** - GestureBegin, GestureEnd, GestureMove, GestureWheel, GestureCancel
- **windowHandle** - handle of the top-level window in which the gesture is performed
- **cursorPos** - cursor position at the moment of the event
- **wheelDirection** - the mouse wheel direction on GestureWheel event (up, down, left, right)
- **timeout** - on GestureCancel specifies whether the gesture has been cancelled automatically by a timeout or manually by the user

Callbacks:

- **Callback_GestureEvent** - this function is being called to notify the main application about the gesture capture phases
- **Callback_WindowIgnored** - the application is asked every time a gesture should be activated whether the given window should be ignored

Methods:

- **instance** - return a reference to the GesturesHook singleton instance
- **initialize** - set the mouse hook and start capturing gestures, the return value indicates success or failure
- **terminate** - stop capturing gestures and release the mouse hook
- **setCallback_GestureEvent** - gesture-event callback setter
- **setCallback_WindowIgnored** - window-ignored callback setter

Properties:

- **enabled** - gesture capturing can be disabled temporarily
- **toggleButton** - toggle button (left, right, middle, X-button 1 & 2)
- **activationDistance** - activation distance in pixels
- **timeout** - timeout interval in milliseconds

4.4 Implementation

Every application instance has its own virtual memory address space. Therefore, all instances of the gesture catcher library are separated from each other. The code segment, which is read only, is mapped to the same physical memory to save resources. The data segment is private to each instance. As we need to have a globally shared instance of the **GesturesHook** singleton class, a shared data segment has to be created using compiler directives. The instance is being instantiated into a pre-allocated buffer in this memory area using C++ placement new operator. Dynamic memory allocation from heap must be avoided completely, as pointers valid in the context of one application are no longer valid in other applications' address spaces. As the result, STL data structures such as strings and containers cannot be used.

Global uniqueness of the statically linked library instance inside the main application is guarded by a system-wide mutex. The mutex is destroyed automatically in case the application crashes. Hence, the library initializes correctly after the application has been restarted. No false reports of a previous instance are reported. In case a previous instance exists, the **GesturesHook::instance()** method throws an exception.

The gesture detection algorithm is based on a finite state machine. Transitions between states occur in response to mouse input events, in context of the mouse-hook callback function. The states are listed and described in figure 4.1. The actual finite state machine, showing all possible state transitions, can be seen in figure 4.2.

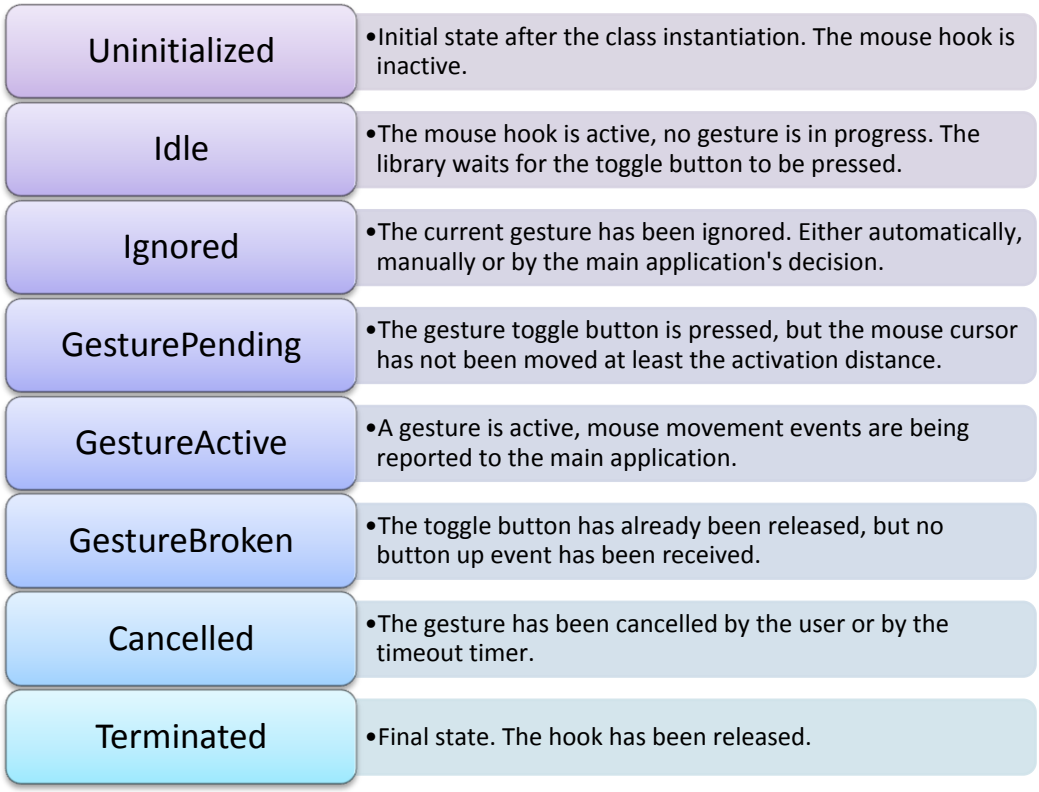


Figure 4.1: Finite state machine states

According to the specification, the user must be able to click the toggle button without starting a gesture. When the toggle button is pressed, the event is filtered and the affected window does not receive the message. When the button is released, this event has to be filtered too. If not, the application would only receive a button up message without a preceding button down message. The button down message has to be sent manually. Care has to be taken not to activate the gesture processing again. The state switches to **Ignored** meanwhile. Otherwise, the event handling would hang in an endless loop. At first, a simple **PostMessage** call has been used. However, the results were not correct, as seen in the Spy++ application included with Visual Studio.

SendInput function had to be used to emulate the mouse click. The effect is the same as if a real mouse button was clicked. Windows sends a correct message to the underlying window for us.

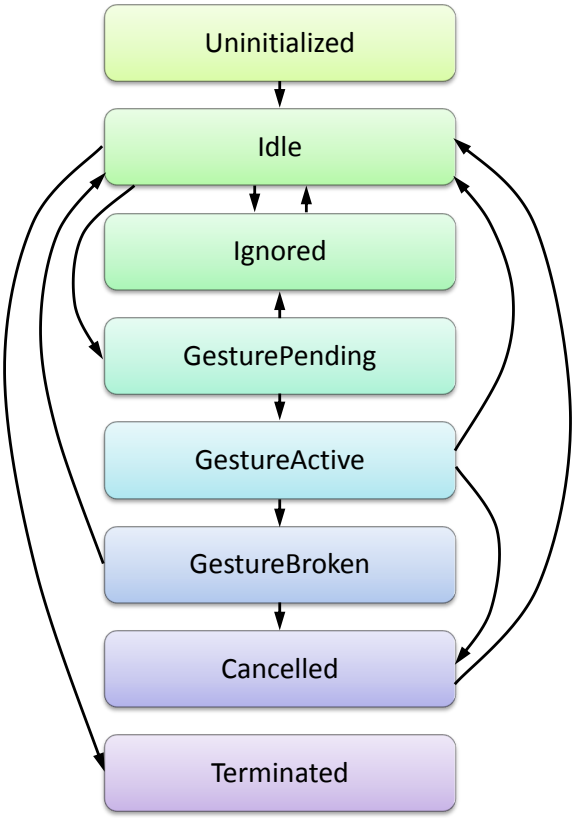


Figure 4.2: State transitions

Without event filtering, inactive windows are normally activated when clicked on, immediately after the button has been pressed, not when it was released. Nevertheless, since the window does not receive the button down message, it is not activated automatically. It has to be done in code by calling **SetForegroundWindow**. Furthermore, **SetCapture** must be used in order to receive mouse movement events when the cursor moves over to another window. Theoretically, this should not be necessary. The hook callback function should be called for the other window. And the library state is globally shared. However, in reality, the gesture only worked in the original window, no mouse messages have been received from the outside.

As already mentioned, all program instances have their own address space. In order to invoke a callback function in the main application, the code flow has to switch the address space of the main application, where the callback function resides. To achieve this, message passing is used. The **initialize** function spawns a new so-called message thread running in our process. This thread creates a hidden message-only window receiving standard window messages.

The function supposed to invoke the gesture-event or window-ignored callback checks whether the current thread is the message thread. If not, the code is being executed in context of a foreign process. By sending a message to the hidden window, the context switches to our application, where the message is handled. Now the same function is dispatched, but now running in the correct context and thus it is able to call the callback function this time directly.

The timeout feature is implemented using a timer. There are two types of timers in Windows, standard and multimedia. Standard timers have been tried first. The main advantage of the multimedia timer [22] being more accurate is not required. However, problems have been experienced with standard timer behavior, while the multimedia one worked correctly. Hence, this type of timer has been selected. The code setting and clearing the timer has to be executed in the context of the main application. Therefore, a timer handler routine is called just before the gesture-event callback function. We have to use a single shot timer to prevent multiple invocations of the handler function. The timer is started on the **GestureBegin** event, restarted on every **GestureMove** event a finally destroyed on all other events. If the timer handler procedure happens to be executed, the timeout interval has been reached. A **GestureCancel** event is sent to the main application.

4.5 Issues experienced

The development of the gesture capture library proved a challenge as predicted in the Introduction. Some parts of the actual implementation had to be revised a few times, to solve some problems, which have been discovered only when the fully functional application started to be used practically. The experienced issues include:

- some kinds of windows are problematic, such as the Start menu. These have to be detected reliably and ignored automatically. It is not an easy task though, window classes and styles have to be checked. The detection has to be as accurate as possible. Otherwise, windows that use a non-typical combination of attributes may be accidentally ignored
- useless mouse movement events are being reported in some cases. In some cases, the mouse-hook callback is being called periodically, even if the mouse cursor does not move, reporting the same cursor position. The last position has to be saved and compared with the "new" position every time, to avoid reporting fake events to the main application
- to support mouse button swapping correctly, logical and physical buttons had to be distinguished. The logical button identifiers, reported to the mouse-hook callback function are swapped. However, the physical

buttons as used by the **SendInput** and **GetAsyncKeyState** functions remain and thus have to be remapped manually

- in addition, the timeout timer has originally been implemented in the main application. However, problems due to thread scheduling and shifted time-spans between events when processed in the main application have emerged. The other reason was that the library did not know about the timeout actually. Gesture events were still being sent and ignored immediately by the main application. Therefore, the timeout had to be implemented fully in the library

- high processor usage condition is also problematic. The application consuming the CPU can prevent the program, in which a gesture is being performed, from being scheduled sufficiently regularly. This leads to a significant number of points missing in the gesture shape, preventing any possible recognition. This problem has not been solved, as it proved to be a rare issue, although quite complex and time consuming to deal with

Chapter 5

Main application

5.1 Application design

The application has a modular, object oriented design taking advantage of class inheritance and virtual methods. More advanced C++ features, such as templates, exceptions and smart pointers have been used to increase productivity and reliability. We also tried to exploit the potential of the Qt toolkit as much as possible by taking advantage of the modules and features it provides. These include **QPointer**, **QObject**, signals and slots, custom events, multithreading, XML, etc. However, there were many cases where the platform dependent Windows API had to be used to achieve things not possible in Qt. This is mainly due to the nature of the application. We have to control other applications and deal with the operating system.

The application implementation is divided into separate more or less independent classes. The majority of the code belongs to class methods. Only small helper routines used all over the application are standard functions. By their nature, classes can be separated into four distinct groups:

- core logic
- helper classes
- user interface
- command implementations

Application configuration is based on Extensible Markup Language (XML) [12] in order to be modular. It proved to be beneficial during the development, when the user interface has not yet been implemented. As the format is user readable, the configuration could be easily modified using a simple text editor

only. Now that the application has been finished, it allows adjustment of advanced parameters, which are not accessible from the user interface. To achieve the concept, classes that need to retain persistent state have to implement:

- constructor accepting an XML element (in some cases a separate method **loadFromXml** is provided)
- a method returning the actual state represented by a hierarchy of XML elements (**toXmlElement** / **saveToXml**)

Hierarchical object structure used in the application maps perfectly to the tree-like structure of XML documents. State of an object corresponds to an XML element, which is a sub-tree in the document. The root of the document represents the state of the whole application. While saving or loading the state, parent objects delegate the XML parsing / assembling to their child elements. It is also possible to store only a part of the configuration to a separate document. This is used in the user interface to implement export and import functions.

5.2 Gesture capture library wrapper

The gesture capture library communicates with the main application using callback functions. This is a low-level interface provided by the library. The capture library has been designed to be independent of any frameworks or toolkits that could be used in the hosting application. The callback interface had to be wrapped for two main reasons:

- it is reasonable to translate the low-level library events into Qt events, to allow further processing using standard Qt event loop mechanism. This also makes the dependency on the capture library modular. The rest of the application would not be affected, if we decided to replace the gesture capture library. Only the wrapping code would have to be modified
- the callback function is executed in the context of the message thread created by the capture library. By posting an event to the default Qt event loop running in the main thread, we switch the context in which the event is being actually handled to the main thread. Thus, we prevent possible synchronization problems and the limitation of Qt, which does not allow certain operations to be performed outside the main thread

Only the gesture-event callback function posts events to the main event loop. The window-ignored callback has to be processed immediately. As the operation is constant and thread-safe, the decision can be delegated directly to the **ActionMapping** class, mentioned later.

The gesture capture library identifies the affected window by a top-level window handle used by the Windows API. **Application::Window** and **Application** classes have been created to wrap the concept of a window and the corresponding application. The window handle is wrapped as soon as possible in the event handler and never used directly in the rest of the application.

The **Application::Window** object provides higher-level methods to manipulate other windows. These are mainly used in the commands implementation classes. The **Application** object provides access to the program's executable icon and version information mainly used in the user interface. It is also used in the action mapping process as described later.

5.3 Engine

The main application class is the **Engine**, which descends from **QApplication**. It represents the application instance and provides the main event loop. Only one instance is allowed. In Qt based applications, this is the first class to be instantiated. However, before the **Engine** is initialized, we need to make sure that the following conditions are met:

1. a previous instance of our application must not be already running. Multiple instances would be sharing the configuration file, one overwriting changes by the other. In addition, the gesture capture library can only be used by one application at a time
2. the gesture capture library must initialize properly. Setting up the mouse hook can fail for some reason. A personal firewall solution may be preventing applications from setting system wide hooks for example

If any of these conditions fail, it is considered a fatal error and the start-up process is aborted. When all conditions are met, the Engine class is instantiated and initialized. Then the main event loop is entered. It is being executed until the application quits, either by the user or automatically when the operating system is being shutdown.

The **Engine** is responsible for many aspects. It has to:

- set-up various application specific parameters, such as the application name and icon, used throughout the user interface
- manage the configuration. The XML based configuration as well as a set of special configuration independent settings which have to be loaded at start-up. Configuration is saved automatically in certain situations, such as when the applications exits or the user clicks a save button

- instantiate important classes, used in the gesture processing, and provide access to their instances. The important objects, as can be seen in figure 5.1, include:
  - **PatternList** containing a list of user-defined gesture patterns and corresponding samples
  - **RecognizerModule** instances, one per each gesture recognizer available
  - **ActionMapping**, which holds the actual configuration of gesture to action mappings
- enumerate and load plug-ins containing gesture recognizers or command implementation
- handle gesture-detection events, delegate the gesture recognition and further related tasks
- provide methods to control the application, including:
  - enabling or disabling the gesture recognition
  - quitting the application
  - forcing the configuration to be saved immediately
  - showing individual configuration windows
  - replacing the currently used **PatternList** or **ActionMapping**
  - setting the active gesture recognizer

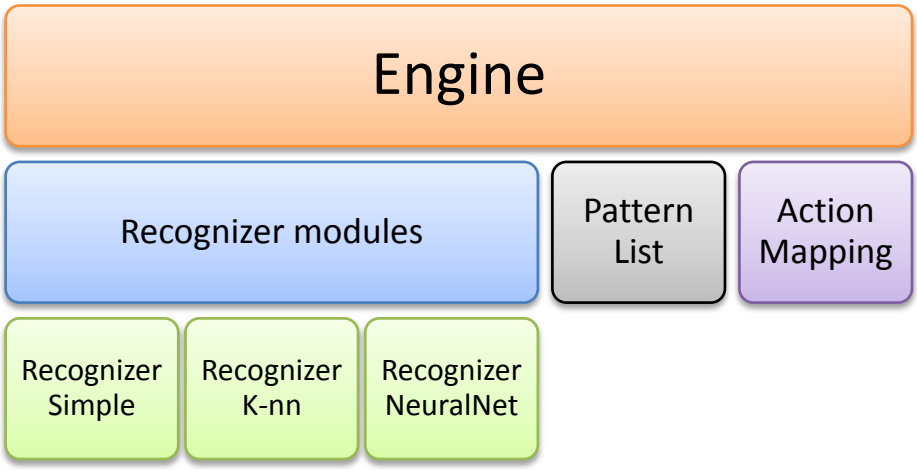


Figure 5.1: Engine and the important classes

The gesture processing is a rather straightforward process. While the gesture is being performed, the cursor position as reported by the gesture capture library is being reported continuously to the active gesture recognizer. The recognizer

is given the opportunity to perform the recognition on the fly. In case a wheel gesture has been performed, no recognition is necessary. Wheel gestures are processed immediately. Regular gestures are processed right after they have been finished, unless a timeout or user cancellation has occurred. The active **ActionMapping** object determines the action to be executed according to the recognized gesture and the affected application. A simplified overview of the code flow can be seen in figure 5.2.

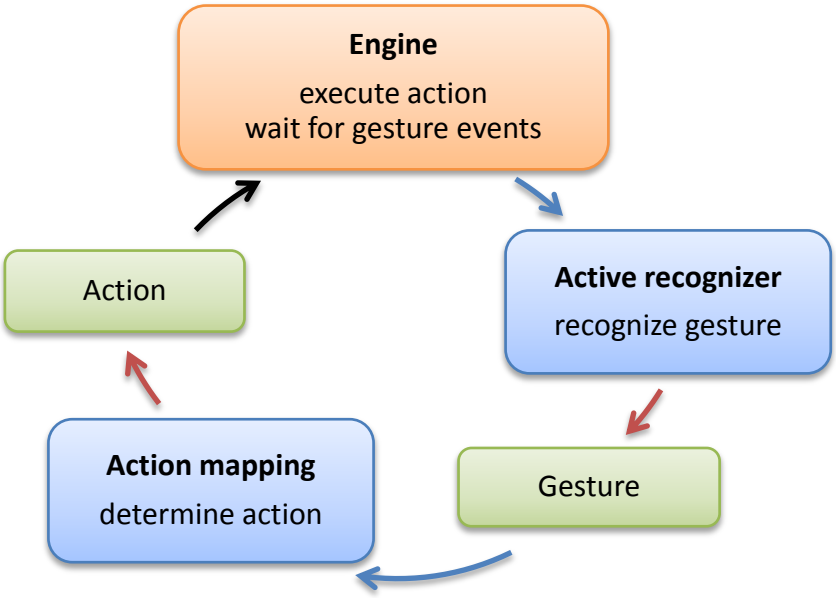


Figure 5.2: Simplified code flow

5.4 Gesture recognizers

Gesture recognizers implement the actual gesture recognition, according to the analysis done in the Gesture recognition chapter. There is one **RecognizerModule** instance per each recognizer available. An instance of the module exists during the whole application life cycle. However, the recognizer itself does not have to be instantiated. The instance is created on demand. Recognizers can be loaded and unloaded dynamically to conserve system resources occupied by the class instance. In case the recognizer is a plug-in, the corresponding library is only mapped while in use. The mechanism has been implemented manually using **QLibrary**, even though Qt supports plug-ins natively and provides a **QPluginLoader** class. However, it is not able to return any plug-in identification data without returning the class instance. In our case, when the plug-ins are enumerated, the recognizer itself is not instantiated, only meta-data information is retrieved from the library. The library is then unloaded when no longer needed. To retain the recognizer state, **RecognizerModule** stores the configuration for the recognizer, while it is not

instantiated.

There are three levels of gesture recognizers, prescribing the interface:

1. **IRecognizer** is a simplified recognizer, which does not use user defined gesture patterns. The gesture identification has to be generated from the actual gesture shape. The **RecognizerSimple** implements this interface

2. **IPatternRecognizer** is supposed to recognize gestures corresponding to user defined gesture patterns stored in a **PatternList** provided by the **Engine**. When the **PatternList** is modified, the owning **RecognizerModule** notifies the recognizer. The recognizer should respond by updating any internal structures that may be affected. From this moment on, only patterns from the new list must be recognized. **RecognizerK-nn** is the actual implementation of this recognizer class

3. **ITrainableRecognizer** extends the **IPatternRecognizer** interface to support training. Training is considered a lengthy operation, which can take a relatively long time to complete. Therefore, a **IPatternRecognizer::Trainer** class is provided. The code of the training procedure is executed in a separate low-priority thread to keep the application responding. While being trained, the recognizer is still able to perform recognition, if it is up-to-date with the current **PatternList**. The **RecognizerNeuralNet** implements this most advanced interface

The actual class hierarchy of the provided recognizers is shown in figure 5.3.

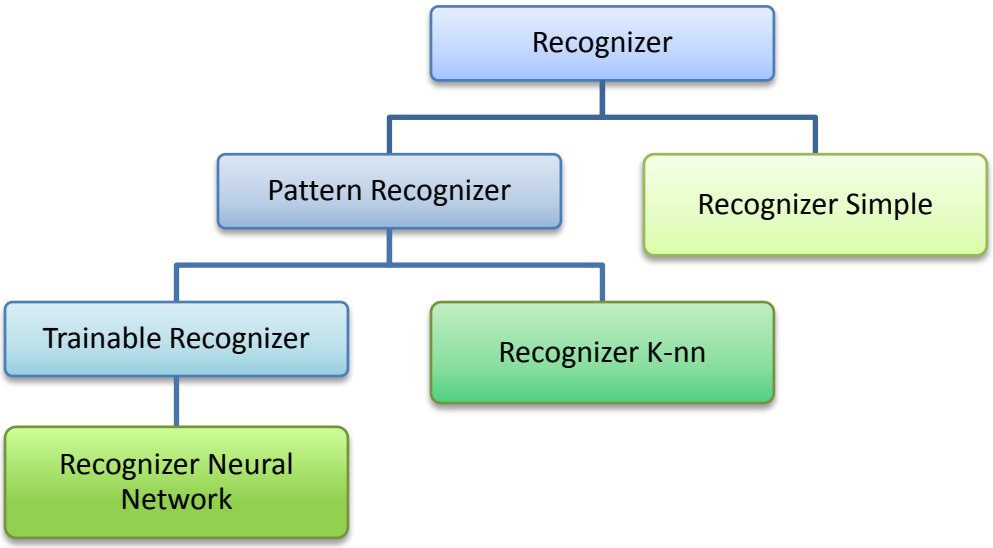


Figure 5.3: Recognizer class hierarchy

The **PatternList** is a container for **Pattern** classes. A **Pattern** represents a single user defined gesture pattern. It is actually a named polyline defining the

shape of the gesture. It can also contain a variable number of pattern samples, defined by individual user to allow better recognition. Pattern recognizers can use the samples either optionally or exclusively. The current **PatternList** owned by the **Engine** cannot be edited; it can only be replaced completely by a new one. Whenever the **PatternList** is modified, all trainable recognizers have to be retrained. If changes were allowed, new training would be started on every single modification. Moreover, the user would not be able to discard the changes without affecting the state of the recognizers.

The recognized gesture is represented by a **Gesture** class. It is a simple structure containing a flag to determine whether it is a regular or wheel gesture. Regular gestures are identified by their name, wheel gestures by the wheel direction.

5.5 Action mapping

The purpose of the **ActionMapping** class is to store and provide access to user-defined configuration:

- application **exclusions** specify a list of programs, in which the gesture processing is ignored. The **isApplicationIgnored** method is a simple look-up to determine whether the exclusion list contains the given application. Individual exclusions can be enabled or disabled
- gesture to action mappings, which are divided into three different groups:
  - **default settings** are general mappings valid in any application
  - **desktop settings** are a special application like group, they match gestures performed on the Desktop
  - **specific settings** define separate application specific list of mappings

  Applications in specific settings, as well as individual gesture mapping items in all three groups, can be enabled or disabled

The main method provided by the **ActionMapping** class is **determineAction**, which is given an **Application::Window** and a **Gesture**. The result is an **Action**. This class is a convenience wrapper for **ICommand** instances, described in the following section. The decision process is simple. Firstly, the application specific or desktop settings are consulted. If specific settings for the given application exist and the list contains the gesture, the corresponding action is returned. Otherwise, the default settings are used. However, there is a property called **inheritDefaults**, which affects further processing in case the gesture has not been found on the list. Unless cleared, the default settings

are used. In any case, when no corresponding action can be found, an empty **Action** object is returned.

Unlike the currently used **PatternList**, the active **ActionMapping** can be modified safely. The next time a gesture is performed, the corresponding action is determined according to the current state. This is beneficial as the user is able to see the changes done in the user interface immediately. To achieve good performance, the internal implementation uses **QMap** classes, which can be indexed directly by an **Application** and a **Gesture**.

5.6 Commands

Action commands have been designed to be modular and easily extensible. The interface is based on the command and prototype design patterns [1]. There are two classes of commands:

- **ICommand** is a basic command, which does not support configuration. It must implement the **execute** method. A **finalize** method, which can be overridden optionally, is also provided. This method is called when wheel gestures are performed. The purpose is to finalize the sequence of multiple command executions, each for one wheel-scrolling event. It gives the command an opportunity to restore any state it might have altered, such as releasing emulated key presses, etc.
- **ICommandConfigurable** is an extended variant of **ICommand**, which is user configurable. It must implement methods, which store and restore the actual command configuration in XML. To present the configuration options in the user interface, the command implementation is supposed to fill a provided container with any widgets it may need. It must be also able to fill the widgets with the current configuration and to retrieve the configuration from the widgets. This is done by implementing a few virtual methods, prescribed by the interface

Instances of the actual command implementations are managed by the **CommandFactory**. The factory is a globally accessible singleton, initialized by the **Engine** at start-up. The factory owns prototypes of all available commands. Other objects can browse the list of commands or directly request a certain command by its name. If it has been found, the prototype of the command is returned to the caller. Prototypes cannot be executed; they must be cloned to create a separate instance.

A list of all commands with their descriptions can be found in the user's manual. Some of the commands have been proposed in the goals, others have been designed while implementing the application.

5.7 User interface

Universal Gestures are a resident application running in the background. It can be configured to be launched automatically on system start-up. It is accessible by a tray icon located in the system notification area. Right clicking the tray icon displays the main menu, containing important commands that control the application. The application configuration interface is divided into two main windows accessible from the tray icon's pop-up menu:

Configuration provides access to the current **ActionMapping**, which can be edited on the fly. All changes take effect immediately. The window is divided into three main parts, as can be seen in figure 5.4. The tree on the left displays the application exclusions, default, desktop and application specific settings. The list on the right side shows the gesture mappings of the item selected in the tree. Configurable commands display their configuration interface in the panel below the gesture-mapping list. Two smaller dialog windows can be seen. **Gesture Selector** provides an interface to select a gesture. Both regular and wheel gestures are supported. **Program Settings** dialog configures the gesture-capture library parameters as well as toggles the use of special user interface elements that are described later.

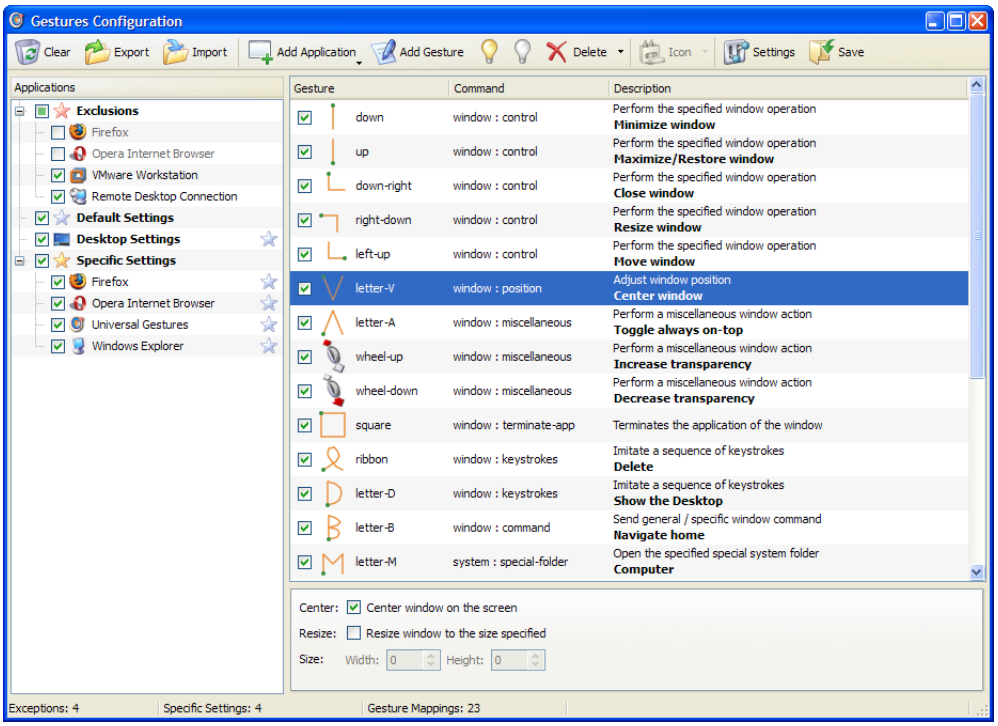


Figure 5.4: Configuration window

Pattern-list Editor (figure 5.5) is an editor for the **PatternList** object. It works with a copy of the current **PatternList**. All changes are independent from the main list, which is replaced only when the editing is done by invoking the save command. A modified unsaved list can be discarded at any time.

The actual pattern editing is done in a separate window called the **Pattern Editor** (figure 5.6). A custom widget able to display and edit a polyline has been developed. The user can also force re-training of all available trainable recognizers and display the training progress.

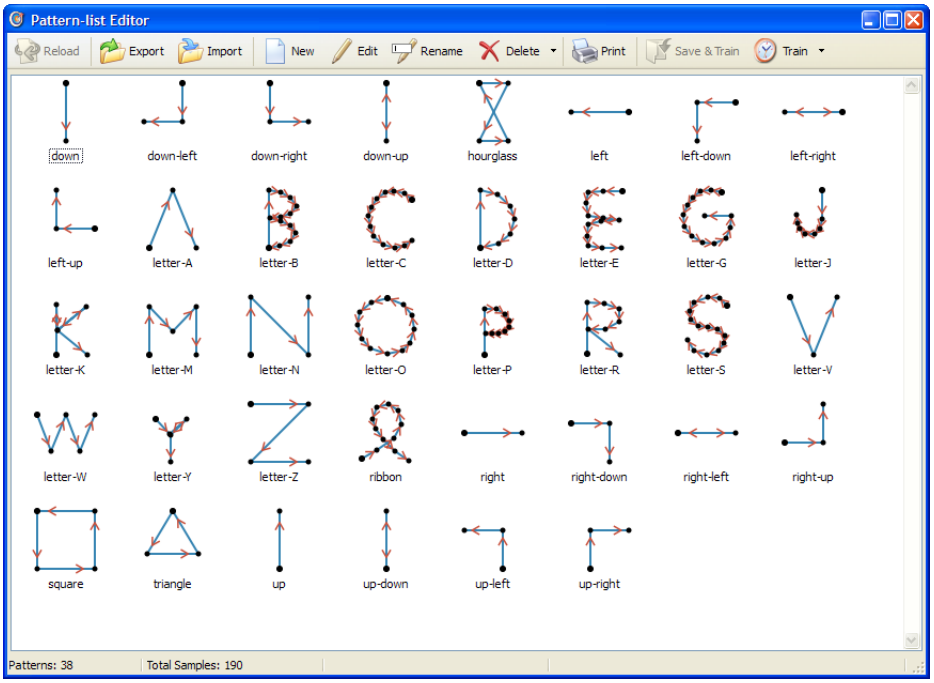


Figure 5.5: Pattern-list Editor window

The tree and list controls in the applications use a model/view/controller (MVC) framework provided by Qt. Custom models and item delegates have been implemented:

- **ApplicationTreeModel** provides data for the application tree in the **Configuration** window
- **GestureMappingModel** is responsible for a particular gesture mapping from an **ActionMapping** object
- **CommandDescriptionDelegate** had to be created to customize the way the command descriptions are displayed on the list
- **CommandSelectorDelegate** provides a combo-box with a list of all available commands for each item on the list
- **PatternListModel** is used in the **PatternListEditor** window to display all user defined gesture patterns, as well as in the **GestureSelector** to provide a list of gestures corresponding to these patterns
- **PatternSamplesModel** displays the list of patterns samples for a given **Pattern** in the **PatternEditor** dialog

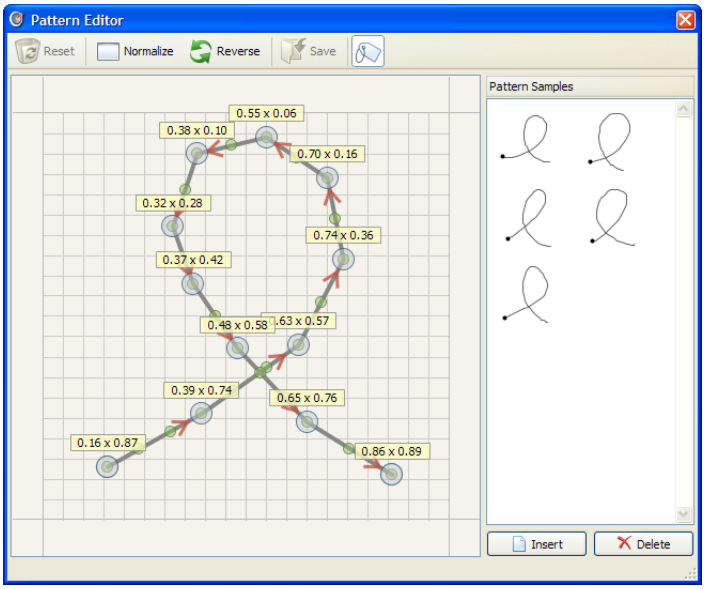


Figure 5.6: Pattern Editor window

The user interface tries to be as user-friendly as possible. Alpha blending has been used to make special purpose notification windows good looking. Unfortunately, the Qt toolkit does not provide a native support for this feature. A custom **QAlphaWidget** class had to be implemented. This was, however, quite simple as widget painting can be redirected into an off-screen bitmap, which is then used to set the shape of the layered window [21]. The following special purpose windows can be observed:

- **GestureOverlay** displays the shape of the active gesture. It is being updated continuously as the mouse cursor moves
- **OnScreenDisplay** notifies the user of various events such as successful gesture recognition, an unknown gesture, etc.

Detailed description of all windows and commands, as well as the installation process, can be found in the user's manual.

Chapter 6

Further development

6.1 64-bit Windows support

The amount of system memory in personal computers increases simultaneously with their processing power. It is due to the permanently increasing demands of applications such as graphics processing software or computer games. This leads to the increasing popularity and adoption of 64-bit operating systems, which are necessary if we want to take advantage of more than 4 GiB of system memory.

CPUs implementing the most popular 64-bit architecture - x86-64 [40] are able to run both 32-bit and 64-bit applications at native speed while running on a 64-bit operating system. These include Windows, Linux, BSD variants, Mac OS X, Solaris, and others. The ability to use existing applications is essential. Many of them work correctly without any or with only small modifications. However, certain kinds of programs, especially those which force other applications to load their libraries, are problematic. The reason is that in Windows, 64-bit application are unable to load 32-bit modules and vice versa [28].

Our application uses a system wide mouse hook. The callback function resides in a dynamically linked library, which is being loaded into the address space of all running application instances. Since the application and the library is compiled either as 32-bit or 64-bit binary, it is unable to hook all the applications [26]. The second problem is caused by a feature called File System Redirector [20]. File and directory names are being translated transparently in order to avoid conflicts between 32-bit and 64-bit file versions. Thus when compiled as a 32-bit binary, the application has to disable this feature in order to see the real file-system structure. This is necessary, because executable file names are used to identify applications.

To provide full support for the 64-bit version of Windows, the following simple

steps would have to be taken:

1. modify the build system to compile the main application, the hook library and all the plug-ins in 64-bit mode
2. create a separate 32-bit helper application, able to load a 32-bit version of the hook library
3. modify the hook library to use dynamically allocated shared memory, so that both the 32-bit and 64-bit versions of the hook library can share the common state information
4. skip creation of the helper thread catching messages in the 32-bit version, send the messages to the 64-bit version's message thread

The main reasons why this feature has not been implemented were time constraints and the lack of a computer with a 64-bit CPU running a 64-bit version of Windows such as Windows XP Professional x64 Edition or Windows Vista 64-bit.

6.2 QtScript

QtScript is a Qt module, which provides powerful scripting support to Qt based applications. The scripting language is based on ECMAScript 3rd edition [11], an internationally recognized standard. JavaScript, which is implemented in all major web browsers to allow client-side scripting in rich web-based applications, is also based on this standard. QtScript features include the ability to use the standard Qt signal/slot mechanism, **QObject** descendants can be exposed to the script, and values converted to their equivalent C++ type and vice versa, etc.

Using QtScript, a new type of command, called script or macro can be implemented. This would allow execution of user defined macros, written in a well-known and powerful ECMAScript/JavaScript like language. Taking advantage of the QtScript abilities, macros would not only be able to access all the existing commands, but could also retrieve various window or system properties.

However, with ability comes the responsibility. The design of this feature has to be well advised in order to preserve the stability of the application, for example in case the script falls into an endless loop. The script command would have to be executed in a separate thread. A mechanism to allow termination of unresponsive scripts has to be provided. The script could be possibly terminated automatically after a specified timeout interval. A watchdog timer mechanism could be also implemented.

6.3 D-Bus

D-Bus [9] is a remote procedure call (RPC) and inter-process communication (IPC) mechanism. It was originally developed on Linux, to unify existing incompatible solutions. Windows port is under development. The specification itself is open, as well as the underlying technologies, such as a XML based interface description language (IDL). Binary message passing protocol promises low overhead and low latency.

The goal of D-Bus is to allow applications:

- communicate with each other within the same user's login session (session channel), providing an efficient IPC mechanism
- communicate with the operating system, system services and device drivers (system channel), applications can listen to various system-level events, such as printer status changes, connection of new hardware, etc.

D-Bus is implemented in several layers. The main component is the central server - bus daemon, whose responsibility is to route messages between the sender and the receiver in the many-to-many paradigm. Direct peer-to-peer connections are possible too. Client applications are linked with **libdbus**. However, this low-level library is usually not used directly, but different programming languages and framework provide high-level wrappers. Applications provide services by exporting objects. Each object must have a globally unique name. Reverse domain path-like names are preferred. Qt provides D-Bus support in QtBus [30] module. It is fully integrated with **QObject** and signal/slot mechanism.

D-Bus is a promising technology, which could potentially increase the possibilities of our applications. It would allow much more flexible invocation of user's applications provided functions. These could even be launched on demand, if not running at the given moment. Although not yet used on Windows, the forthcoming release of KDE 4.1 applications for Windows should start the adoption of D-Bus on this platform. Plug-ins adding D-Bus support for existing extensible applications such as Firefox can be written. In case the application was ported to Linux, many Linux applications are already D-Bus enabled. Moreover, system-level commands, such as volume control, which does not have a common API generally, can be invoked easily.

6.4 Miscellaneous

As multi-core processors are becoming a standard in personal computers, multi-threaded programming is gaining importance. In our case, paralleliza-

tion can be exploited in some parts of the application, most notably the recognizer training process. Training is the most time-consuming process in the whole application. Commands can also be executed in a separate thread to avoid glitches in the user interface. Some of them already create a new thread to perform a lengthy operation. However, there is no common mechanism for threaded execution. Qt toolkit has an extensive support for multi-threaded programming, including **QtConcurrent**, which is an implementation of the **MapReduce** [3] algorithm invented by Google. This technique might be utilizable by our application.

The user interface has been designed to be as simple as possible, to provide the best possible user experience. Nevertheless, there will always be space for improvement. Many new features can be added, such as:

- more configuration options for the special purpose windows described in the User interface section
- floating window to display the training progress. The window would show a graph of the recognition error value and buttons to control the training process
- separate configuration dialogs should be provided by each gesture recognizer to allow advanced users to adjust recognition parameters

The list of built-in commands can be expanded to support relevant functions that would be convenient if triggered by mouse gestures. Moreover, some of the existing functions may be improved by adding more or better configuration options.

Finally yet importantly, the gesture recognition accuracy is a subject of future improvement. There are many different methods how to perform the recognition, ranging from trivial ones to more advanced solutions involving artificial neural networks or other complex data structures. Hence, there are several possible directions. The existing gesture recognizers can be surely improved, by using a better gesture pre-processing algorithm, choosing a different configuration of the recognition parameters, more suitable neural network input representation, etc. Moreover, the development of a new recognizer, based on a completely different algorithm, should also be considered. Application of the **Bayesian network** [5] seems to be one of the available options.

Chapter 7

Conclusion

Our task was to create a mouse-gesture recognition application for Windows, called Universal Gestures. It would enable faster and easier control of other Windows applications, thus greatly enhancing user experience. In order to function as intended, Universal Gestures had to satisfy a number of criteria. The application was supposed to be integrated transparently into the operating system, in order to allow gesture support in existing unmodified applications. It had to intercept mouse input in order to detect gestures. The system integration as well as the program itself was required to be as reliable as possible, to avoid negative impact on other applications. The user was supposed to be able to prevent the gesture recognition both temporarily at a given moment, as well as permanently for a defined set of specific applications.

The main goal of the program was to recognize advanced, user-defined gestures. An editor, serving as a platform for defining custom gesture patterns and associated pattern samples, was therefore necessary. The application had to implement a reasonably accurate gesture recognition algorithm, which would automatically learn to recognize the given set of patterns every time the list changed. At the same time, a simple four-direction algorithm was implemented, in order to satisfy less demanding users.

The user was supposed to be allowed to configure both general and application specific gesture mappings. These would specify the commands to be triggered by the individual gestures. To keep the configuration simple, applications were designed to be identified by an executable file path. A certain basic set of commands was desired, including the ability to minimize, maximize, restore and close windows, set special window attributes, send general application commands corresponding to special purpose keys found on multimedia keyboards, emulate any user defined shortcuts, control the Universal Gestures themselves, execute other applications, open special folders such as the Computer, switch between running applications, etc. The program was designed to run in the background, and be accessible through a tray icon, giving access to the main

menu. One of the main goals was modularity and extensibility based on plug-ins. User friendliness has been regarded important as well.

The choice of the programming language and the integrated development environment was straightforward. The application was written in C++ to achieve good performance and low resource consumption. Microsoft Visual Studio 2008 with Feature Pack was used, since a free non-commercial license is available for students. The choice of the user interface toolkit was a harder one. Several frameworks have been evaluated. The Qt toolkit was finally selected and it proved the right choice. The development with Qt was pleasurable and convenient, despite the fact that some minor issues have been experienced, leading to a custom set of patches applied to the source code. Bugs found were reported to the developer of Qt, and will be fixed in future versions.

As expected, design of an accurate gesture recognition solution proved a challenge. Two different gesture classification algorithms with a common preprocessing phase were implemented, resulting in two separate gesture recognizer modules. However, it is hard to compare them reliably. Each classifier is good for a certain kind of gesture patterns, while being less suitable for others. There are many factors influencing recognition accuracy, including the number of key points used, classifier input representation, adjustable preprocessing phase parameters, amount and quality of user defined pattern samples, neural network training method utilized, etc. During the development, a reasonable configuration of parameters has been found. These were set as defaults in the initial application configuration file.

The development of a reliable gesture detection solution was expected to be, and in fact became, a challenge as well. The initial idea was to create the gesture capture library first, and then focus on the rest. However, parts of the actual implementation had to be rewritten during the development, in order to respond to the issues experienced only when the application already started to be used practically. Despite all the obstacles encountered, a reliable yet not perfect result has been achieved. There are many cases to be handled in order to avoid incorrect behavior.

The main application was implemented according to the proposed design. The interface is user friendly and easy to use. Even a couple of features not originally planned have been implemented. The gesture line indicator may be considered the best example.

Thus, we can consider the overall result a success. The expectations set at the beginning have been met. Nevertheless, there will always be room for improvement. Complete support for 64-bit editions of Windows is probably the most important and will be added in the future. Other directions of further development include the research of better gesture recognition algorithms, improvements in the user interface, introduction of scriptable commands, D-Bus integration, etc.

Bibliography

[1] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[2] Hofman, P., Piasecki, M.: *Efficient Recognition of Mouse-based Gestures*, Proceedings of Multimedia and Network Information Systems, p. 89-98, Wyd. PWr., 2006

[3] Dean, J., Ghemawat, S.: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004, `http://labs.google.com/papers/mapreduce-osdi04.pdf`

[4] *Artificial neural network - Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/wiki/Artificial_neural_network`

[5] *Bayesian network - Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/wiki/Bayesian_network`

[6] *Boost C++ Libraries*, `http://www.boost.org/`

[7] *CodeProject: Mouse gestures recognition*, `http://www.codeproject.com/KB/system/gestureapp.aspx`

[8] *CodeProject: Three Ways to Inject Your Code into Another Process*, `http://www.codeproject.com/KB/threads/winspy.aspx`

[9] *D-Bus - Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/wiki/D-Bus`

[10] *Desktop Window Manager - Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/wiki/Desktop_Window_Manager`

[11] *ECMAScript - Wikipedia, the free encyclopedia*, `http://en.wikipedia.org/wiki/ECMAScript`

[12] *Extensible Markup Language (XML)*, `http://www.w3.org/XML/`

[13] *FireGestures*,
http://www.xuldev.org/firegestures/

[14] *GTK+ - About*,
http://www.gtk.org/

[15] *k-nearest neighbor algorithm - Wikipedia, the free encyclopedia*,
http://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm

[16] *Microsoft Foundation Class Library - Wikipedia, the free encyclopedia*,
http://en.wikipedia.org/wiki/Microsoft_Foundation_Class_Library

[17] *Mouse Gestures in Opera*,
http://www.opera.com/products/desktop/mouse/index.dml

[18] *Mouse Gesture for IE*,
http://www.ie7pro.com/mouse-gesture.html

[19] *MSDN - DirectInput*,
http://msdn.microsoft.com/en-us/library/bb219802.aspx

[20] *MSDN - File System Redirector*,
http://msdn.microsoft.com/en-us/library/aa384187(VS.85).aspx

[21] *MSDN - Layered Windows*,
http://msdn.microsoft.com/en-us/library/ms632599(VS.85).aspx#layered

[22] *Multimedia Timers*,
http://msdn.microsoft.com/en-us/library/ms712704(VS.85).aspx

[23] *MSDN - Raw Input*,
http://msdn.microsoft.com/en-us/library/ms645536(VS.85).aspx

[24] *MSDN - Standard C++ Library TR1 Extensions Reference*,
http://msdn.microsoft.com/en-us/library/bb982198.aspx

[25] *MSDN - Subclassing Controls*,
http://msdn.microsoft.com/en-us/library/bb773183(VS.85).aspx

[26] *MSDN - Using Hooks*,
http://msdn.microsoft.com/en-us/library/ms644960(VS.85).aspx

[27] *MSDN - Windows Vista User Experience Guidelines*,
http://msdn.microsoft.com/en-us/library/aa511258.aspx

[28] *MSDN - WOW64 Implementation Details (Windows)*,
http://msdn.microsoft.com/en-us/library/aa384274(VS.85).aspx

[29] *Qt 4.4.1: Qt Object Model*,
http://doc.trolltech.com/4.4/object.html

[30] *Qt 4.4.1: QtDBus module*,
http://doc.trolltech.com/4.4/qtdbus.html

[31] *Qt 4.4.1: QtScript Module*,
http://doc.trolltech.com/4.4/qtscript.html

[32] *Qt Cross-Platform Application Framework - Trolltech*,
http://trolltech.com/products/qt/

[33] *Qt (toolkit) - Wikipedia, the free encyclopedia*,
http://en.wikipedia.org/wiki/Qt_toolkit

[34] *StrokeIt - Mouse Gestures for Windows*,
http://www.tcbmi.com/strokeit/

[35] *Technical Report 1 - Wikipedia, the free encyclopedia*,
http://en.wikipedia.org/wiki/Technical_Report_1

[36] *The Official Microsoft WPF and Windows Forms Site*,
http://windowsclient.net/

[37] *Windows Template Library - Wikipedia, the free encyclopedia*,
http://en.wikipedia.org/wiki/Windows_Template_Library

[38] *Working with the AppInit_DLLs registry value*,
http://support.microsoft.com/kb/197571

[39] *wxWidgets*,
http://www.wxwidgets.org/

[40] *x86-64 - Wikipedia, the free encyclopedia*,
http://en.wikipedia.org/wiki/X86-64

Appendix A

User documentation