ParleVision•5

Michel Jansen
Richard Loos

Supervisors: Dennis Reidsma,
Ronald Poppe
# Contents

1 Introduction 2

2 Related Work 3
   2.1 Computer Vision Libraries 3
   2.2 Computer Vision Pipeline and Creative Programming Frameworks 4
      2.2.1 Computer Vision Pipeline Frameworks 4
      2.2.2 Creative Programming Frameworks 5
   2.3 Visual Programming and Compositing Environments 6
   2.4 A closer look at ParleVision4 7

3 Context & Task Analysis 9
   3.1 User Profile 9
   3.2 Usage Scenarios 9
      3.2.1 Computer Vision Research 10
      3.2.2 Natural Interaction Research 10
   3.3 Use cases 10
      3.3.1 Interactive Installations 10
      3.3.2 Virtual and Embodied Agents 11
      3.3.3 Computer-Aided (Security) Surveillance 13
      3.3.4 Video and Image Annotation 14

4 Analysis of the ParleVision4 framework 15
   4.1 ParleVision4 architecture 15
      4.1.1 The pipeline model 15
      4.1.2 The type system 16
      4.1.3 The execution model 16
      4.1.4 Serialisation support 17
      4.1.5 Extensibility 17
   4.2 The user interface 17
   4.3 State of the project and code base 18

5 ParleVision5: Design and Architecture 20
   5.1 Goals and System Requirements 20
   5.2 Programming language and application development framework 21
   5.3 The processing pipeline 21
      5.3.1 The pipeline model 21
      5.3.2 The type system 22
      5.3.3 Producers 23
      5.3.4 Processors 23
      5.3.5 The execution model 24
      5.3.6 Serialisation support 25
      5.3.7 Extensibility 25
5.4 The Graphical User Interface ............................................ 26
  5.4.1 GUI Design ......................................................... 26
  5.4.2 GUI Architecture .................................................. 28

6 Evaluation ......................................................... 30
  6.1 Usability .............................................................. 30
    6.1.1 The GUI ......................................................... 30
    6.1.2 The API ......................................................... 30
  6.2 Performance ........................................................ 32
    6.2.1 ParleVision4 versus ParleVision5 ............................. 32
    6.2.2 ParleVision5 single thread versus multiple threads ........ 32

7 Conclusion ............................................................. 34

A System Requirements ............................................... 35

B Acknowledgements .................................................. 37
Abstract

This report describes the design of ParleVision5, an open source, cross-platform framework for rapid prototyping of computer vision applications for Human Computer Interaction purposes. First related work, including the framework’s predecessor ParleVision4, is explored along with the use cases they were designed for. After an in-depth analysis of ParleVision4, the design and architecture of a new framework is described. ParleVision5 is designed to be faster and easier to use by having a solid modular architecture, a clear extension API and graphical user interface and by being optimized for computers with multiple cores and/or processors.
Chapter 1

Introduction

A need exists for building complex Computer Vision (CV) applications which are suitable for Human-Computer Interaction (HCI). These systems are used as part of various research topics, where the goal is not to research computer vision algorithms in itself, but to build computer vision systems which are part of a larger whole in which the interaction between human and machine is the main research topic. These computer vision applications usually have a big overlap in required functionality: the program needs to do image acquisition, pre-process the data and run one or more algorithms on the data before outputting the result to a screen, text file, network or another program and it needs to operate in real-time in order to have fluid interaction between the system and its users.

To help researchers do work which is not the focus of their research, frameworks providing common computer vision functionality have been developed. ParleVision4 [6] is an example of such a framework that was developed internally at the Human Media Interaction (HMI) group of the University of Twente. It provides a graphical user interface (GUI) and facilities to write and re-use discrete components which provide processing functionality in the style of visual programming environments. While ParleVision4 has served its purpose for several years, it has some shortcomings that are causing problems for researchers and students looking to do research that involves a computer vision component.

This paper reports on an effort to build a next generation computer vision framework that can aid in rapid prototyping of computer vision applications. Chapter 2 gives a review of computer vision frameworks, graphical user interfaces and relevant visual programming languages in this field. This is followed by an analysis of the design and implementation of the ParleVision4 framework in Chapter 4. Based on the analysis of the shortcomings of the ParleVision4 architecture, a new framework architecture and design is presented in Chapter 5. Finally, a comparison of how a prototype implementation of the new framework stacks up against its predecessor in terms of ease of use, flexibility and performance is made in Chapter 6.2.
Chapter 2

Related Work

The need for rapid prototyping of computer vision applications in other human machine interaction departments has lead to the development of a wide variety of libraries, frameworks and tools. Many research departments have constructed their own frameworks that often never set foot outside the laboratory where they were created. When looking at the various projects we make a distinction between programming libraries and frameworks that aid with programming computer vision applications and the visual environments which allow non-programmers to create such applications. Sometimes these two overlap as some frameworks both provide users with a GUI and allow more extensive customization using an API or a domain specific programming language. Figure 2.1 shows an overview of the related libraries, frameworks and creative toolkits discussed in this section.

2.1 Computer Vision Libraries

At the lowest level, there are various libraries that aid with programming computer vision applications. These libraries contain basic data structures, algorithms and access methods needed to process video frames.

OpenCV [27] is an open source computer vision library which is used extensively in vision related applications and frameworks. It was originally created by Intel and is now supported and actively developed by a company called Willow Garage, which develops hardware and open source software for personal robotics applications. There are other open source computer vision libraries such as VXL [19], IVT [3] and Mimas [4], but none of these match the scope and functionality of OpenCV or are used as much and are as actively developed. There are also generic image libraries which are not specialised in computer vision but provide image transformation and editing functionality which can be used for computer vision, e.g. ImageMagick [2] and ImageJ [1]. Other libraries are more specialised, such as ReacTIVision [13], which is focused on multi-touch finger tracking and the tracking of fiducial markers.

While this research will only focus on open source or freely available libraries, there are good commercial libraries as well. For instance the the Claron Tech Withinsight [24] Framework is available for the analysis of medical images, Matrox offers the Matrox Imaging Library [17] and National Instruments has the NI Vision [12] toolkit. Finally, interesting research is done at Microsoft Research, which have released the closed source Microsoft Vision SDK [18].

There are many frameworks which use OpenCV internally to provide the low level CV functions while providing higher level functions and integration with other libraries themselves. For example, the popular creative coding framework OpenFrameWorks [14] uses OpenCV internally. OpenFrameWorks also serves as the basis for many project, for example the Nuiigroup CCV [20] framework, which is specialized in multi-touch interfaces.
2.2 Computer Vision Pipeline and Creative Programming Frameworks

There are many projects that aim to present users with a friendlier programming environment – or API – and make it easier to program domain specific things. They all attempt to abstract, simplify or provide other help in the programming process. In this overview, we consider frameworks for computer vision and other “creative programming” purposes.

2.2.1 Computer Vision Pipeline Frameworks

The idea of providing a framework to simplify the task of creating pipelines of computer vision transformations has been around for years and ParleVision4 is just one example of a project in this category. We will take a look at three different projects: Phission, IceWing and EyesWeb.

Phission is a computer vision library developed at the University of Massachusetts Lowell Robotics Lab. The project began as a means to create a vision system that would be easier to extend, improve the use of the capture resource, and provide a low latency response in robotic computer vision applications. It is used by projects within the lab for computer vision research on mobile robots including search and rescue, assistive robotics, and other advanced student projects. The Phission library is written in C++ and defines an API not dissimilar to ParleVision for I/O and the processing thereof. Data is first captured in a capturing device such as a webcam, processed and finally displayed or logged. Phission does not concern itself with the processing of the actual data, but relies on external libraries such as OpenCV for this. The processing of the captured input is done in a pipeline, which can contain multiple filters. There is no inherent parallelism support in the pipeline, the filters are processed in the order in which they are added to the pipeline. The pipeline does however run in its own processing thread, so processing can proceed concurrently with other tasks such as capturing the input and displaying it on the screen. Phission also provides among others classes for communicating over a network, connecting to controllers such as a joystick, for receiving inputs for specialized hardware such as a sonar, and some basic GUI related classes. It is however not a working complete
program, but a library for building a program with and while an example program is included, this has very limited functionality.

IceWing [15] is a graphical plugin shell. While it is optimized for vision based processing, it is also possible to use it for other fields of processing such as audio. Almost all functionality in IceWing is contained in plugins, which operate on data provided by other plugins or can generate data of their own. It is not based on OpenCV and defines its own image type, which gives it very little functionality out-of-the-box. It provides GUI elements which allows plugins to make use of them to enable online updating of plugin parameters. The plugins can open any number of windows and display in these windows any data in a graphical form. Like Phission the pipeline model itself has not been designed with parallel processing in mind. IceWing is written in C, Unix only and uses the GTK toolkit for its GUI elements. While it offers convenience functions for creating a GUI it does not allow the composing of a pipeline at runtime, but instead needs to be programmed in the C programming language. So a compiler is needed to enable the composition of a pipeline which makes rapid prototyping by non programmers hard.

One of the most extensive and well known frameworks and visual editors for building applications aimed at multi-modal interfaces is (EyesWeb [7]). EyesWeb’ development started in the late nineties and the program is still actively developed today. EyesWeb was designed with a special focus on the analysis and processing of expressive gesture in movement, midi, audio, and music signals. According to the EyesWeb website “the EyesWeb open platform has been originally conceived for supporting research on multi-modal expressive interfaces and multimedia interactive systems. EyesWeb has also been widely employed for designing and developing real-time dance, music, and multimedia applications.” Just like ParleVision4, EyesWeb provides the ability to build a graph out of discrete components in the style of visual programming languages which are called blocks in EyesWeb. It provides many advanced features for building pipelines, such as I/O, flow control, synchronization mechanisms and the integration of many computer vision and audio processing patches and visualisations thereof. It can make use of Symmetric Multi-Processing (SMP) systems and has some initial support for the use of the Graphical Processing Unit (GPU) as general purpose processor, a technique called GPGPU, to speed up certain processor intensive operations. While the actual source code of the program is closed, programmers can develop extensions to the program by using its Software Development Kit (SDK). Although for the computer vision components the OpenCV library is used, EyesWeb is not based on OpenCV as all data types and components are defined as plugins. Viewers are part of the pipeline. The framework is very mature, but suffers from some problems as well. For example, the pin data is defined “in place”, so programmers need to use hard-to-use locks for concurrent operations. Datatypes are extensible but the user needs to implement an interface, resulting in overhead for simple types such as int and double. Finally, documentation of the project is sparse.

2.2.2 Creative Programming Frameworks

Frameworks that are designed to make programming easy, but are not necessarily intended for computer vision applications are ubiquitous. We consider two relevant frameworks: Processing and OpenFrameworks.

Processing [9] is “an open source programming language and environment for people who want to create images, animations, and interactions”. It consists of two parts: a simplified programming language based on a Java and a “sketchbook” environment that helps the user write code. It is free and open source, cross-platform and very well documented with a large and active community. Processing itself does not come with frameworks or libraries for computer vision, but there are many 3rd party extensions that add binding to OpenCV, blob detection and other Computer Vision functionality. Because Processing
Figure 2.2: VVVV

is based on Java, the user does not need to worry about memory management, but this makes the language less powerful for more demanding applications.

OpenFrameWorks [14] is a “creative coding” framework for C++. It is intended to allow for quick experimentation with interactive audio and visuals. Like Processing, it is not strictly intended for computer vision applications, but it uses OpenCV internally and as such can be used to perform visual transformations and recognition. OpenFrameworks mainly focuses on making programming easier, so while it is well-documented and provides clean APIs, it expects the user to be able to understand, write and compile C++ code and provides no graphical interface or development environment.

Neither of these frameworks are intended for computer vision per se, so they do not provide a friendly GUI for non-programmers.

2.3 Visual Programming and Compositing Environments

The idea of representing programming functionality in a graphical environment has existed for some time. Many programs offer the ability to construct a program graphically by connecting logic building blocks to each other. The most well-known examples of visual programming languages are Lego Mindstorms and Scratch [16]. As there are too many visual programming languages too discuss, we only consider those relevant for computer vision purposes. Examples include MAX/MSP, VVVV and Quartz Composer.

MAX/MSP is a commercial software package for creating interactive sounds and visuals. It allows users to visually connect “patches” that perform transformations, represent decisions etc. The software is intended for artists and an important aspect of the program is that everything can be driven by MIDI signals. MAX/MSP has multiple C APIs for users to write extensions, some of which can also be used in conjunction with Java wrappers. There are many spin-offs that replicate or extend on the MAX/MSP concept such as jMAX and PureData. As they are pretty similar, they will not be discussed separately.

VVVV is a multi-purpose toolkit for creating audio, video, 3d animation and other media common in computer arts. Its main focus is providing an environment where transformations on these media are easy to express. Like MAX/MSP, it provides a visual environment where building blocks can be connected to write the program logic. For extensions it offers a plugin interface, in addition to support for VST\(^1\) plugins and DirectX effects. VVVV requires DirectX and as such it works on Microsoft Windows only. It is free for non-commercial use, but the source code is not disclosed.

\(^1\)http://www.steinberg.net/en/products/partner_products/pluginzone.html
Quartz Composer (Figure 2.3) is a visual programming toolkit by Apple for processing and rendering graphical data. It comes with an extensive library of elements that represent low-level operations and high-level transformations that the user can put together using drag and drop to create visual effects. Quartz Composer heavily relies on hardware accelerated processing and Apple’s proprietary Quartz framework. As such, it only works on Mac OS X. It comes with an API for extensions, which have to be written in the Objective-C language. The application can be downloaded for free, but the source code is not available. Although Quartz Composer is not designed for computer vision applications such as object tracking and recognition, it can be used for these purposes if the user writes the proper extensions.

Visual programming can be a powerful concept for computer vision, because it allows users to create applications without having to write code. EyesWeb, discussed in Section 2.2.1, and ParleVision4 already combine a powerful framework with an easy graphical environment for pipeline composition, although neither offers a visual environment that is as expressive as the ones discussed in this section.

2.4 A closer look at ParleVision4

As mentioned in the introduction, the Human Media Interaction group at the University of Twente developed a framework called ParleVision4 [6]. ParleVision4 set out to be an extensible wrapper around the OpenCV library. It extends the OpenCV programming functionality with a graphical user interface (GUI) and facilities to write and re-use discrete components which provide processing functionality in the style of visual programming environments. It has been developed for ease of use and allows programmers to focus on quickly implementing functionality and non-programmers to do rapid prototyping by graphically composing computer vision ‘pipelines’ from modular functions called ‘pipeline processors’. The processing functions’ parameters can be adjusted and their results observed online, meaning one can tweak the functions’ variables while the pipeline is running. This makes it very easy to tweak the complete processing pipeline for specific circumstances. Since functionality is split into ‘pipeline processors’, code written for
one project, can be reused in other projects. Also, processing pipelines composed for one project, can be easily extended or modified in another project. As such ParleVision4 is a computer vision pipeline framework that includes a visual environment for composing pipelines graphically. During recent research projects, it was found that the framework is not only in need of maintenance, but is also suffering from some fundamental architectural problems that prevent it from scaling well on modern, multi-core systems that were simply not present when the framework was conceived. The ParleVision4 framework will be discussed more extensively in chapter 4.
Chapter 3

Context & Task Analysis

“You are not your user”, is an adage found in every textbook on Human Computer Interaction. Although it remains equally true in this case, the redesign of ParleVision was born out of the authors’ own needs and experiences during a previous project. This means that, inevitably, a lot of the insights into the context, the tasks and the users stem from our own experience. As there were not a lot of real current or potential users to draw from, a lot of the design ideas have been tested against our own past experiences.

3.1 User Profile

There are a number of typical aspects to be found in the target audience. Generally, the user will be a student or a researcher. This means that an academic background, with at least some prior programming experience is presumed.

The user will be interested in using the system for research into either computer vision itself or some kind of human computer interaction, in which case it only plays a supporting role. These different scenarios will be discussed in more detail in Section 3.2.

With deadlines approaching, the user is likely to be under time pressure to get everything working in time. There is also not a strong drive to produce a result that works for any case other than the one at hand. Browsing through source code of ParleVision framework shows many instances where corners were cut and parts of user-contributed processors were left unimplemented because the user had ran out of time.

Regarding the programming skills of the user, a lot of variation is expected. Most users will be accustomed to programming in Java, having little or no experience with C or C++. This means that complex subjects like memory management, variable initialisation and preprocessor macros are likely not well understood [10]. Combined with the shortage of time mentioned earlier, it is not likely that users are immediately open to learning these concepts in advance.

Obviously, programming skills are only relevant when the user wishes to write his processors. All users, including those that only wish to compose and control pipelines from existing elements, are presumed to be able to operate a drag and drop-based user interface with a keyboard and mouse.

3.2 Usage Scenarios

When designing a framework, rather than a concrete system, it is tempting to fall into the trap of over-engineering. It is impossible to envision all the things potential users might want to use the system for, but at the same time it is necessary to define the boundaries for what the framework will and will not solve. To do this, two broad classes of usage scenarios have been defined: (1) computer vision research and (2) natural interaction research. The
difference between these two usage scenarios are that in (1), developing, implementing and testing a computer vision algorithm or pipeline is a goal, whereas in (2) it is more likely to be the means to an end.

3.2.1 Computer Vision Research

The most obvious class of scenarios is the one where a user wishes to do research into computer vision subjects, such as testing new methods or algorithms, or applying existing ones to new problems. The main characteristic shared by these scenarios is that the goal fulfilled by the framework is enabling the user to focus on their own clearly defined problems, without being concerned about surrounding logic that is of no academic value.

If the user is implementing a number of different algorithms to compare them, the framework should handle controlling the camera, displaying the results, providing a consistent environment to run the comparisons in and so on. If the user wants to analyse an algorithm in different kinds of situations, the framework should enable this by making it easy to introduce new situations or input. If the user is looking to find out what hardware (cameras, filters, lighting etc.) works best for a given situation, the framework should give the right kind of feedback to be able to draw sane conclusions. It should be noted that these scenarios do not necessarily require real-time operation. For some cases, it is perfectly acceptable to perform some kind of off-line analysis, as shall be further discussed in Section 3.3.

3.2.2 Natural Interaction Research

A much harder to define class of scenarios is made up of all kinds of ways the framework might be used to support research into various kinds of natural interaction. There are many shapes this research may take, but it is important to note that the main goal they all share, is that the computer vision part is merely there to serve and support a greater purpose, not as a goal in itself.

In the scenarios where the framework is used for natural interaction research, there is often a prototype system, virtual agent, robot, or installation that somehow needs to analyse or respond to the user’s movements. This will generally involve capturing data from one or more cameras, cleaning up the data and then processing it to gather information about or detect blobs, limbs, faces, people etc. In many cases, these systems will need to be able to respond in near real-time.

3.3 Use cases

For more insight into the needs of past, present and future users, some concrete, high-level use cases have been investigated. Rather than attempting to discuss all possible uses of computer vision frameworks, we will focus on applications that are known to have used ParleVison4 in the past to distil general usage patterns.

3.3.1 Interactive Installations

One class of applications where it is often desirable to use computer vision to respond to user motion is in interactive installations. These installations have various purposes, from museum exhibits to digital works of art, but all of them are generally more focused on natural interaction than on the computer vision itself. The user in this case, is the artist, designer or researcher that wishes to build a specific installation. Generally, knowledge of computer vision techniques is basic at best and as the goal is to recognise and respond
Figure 3.1: Screenshot of ParleVision 4.0 pipeline used in Anemone

to people, objects and events, the complexity and nature of required techniques can be expected to be very similar.

To better understand the requirements that users interested in building interactive installations may have, we consider one such installation, Anemone [25], as an example. Anemone is an interactive bar that attempts to bring patrons standing at a bar into contact by projecting a virtual underwater world on top of the surface of the bar. The virtual world responds in real-time to the user’s hands and glasses and other objects above and on the bar as observed using a hidden near-infra-red camera.

To accomplish this, the system employs a CV pipeline consisting of some basic image adjustment steps (smooth, warp) followed by a background subtraction step and some filters that remove shadows and noise. All of this is preprocessing for a monolithic ‘HMIBarBlobDetector’ processor at the end of the pipeline, that detects, identifies and tracks blobs and sends them to the GUI over a network connection.

All of this, however, is hidden from the user and only a small portion of a bigger system. What the user experiences is the combination of the concept and interaction design, the artificial intelligence and dynamics of the world, the physical form and the graphics and animations of the projection. How this is accomplished is of secondary concern to the audience of the installation and thus to its builder, our user.

3.3.2 Virtual and Embodied Agents

Another area where computer vision is merely one critical component of a larger system, is that of virtual and embodied agents. If an agent is to interact with the real world, it is important that it can perceive things in that world. Seeing its environment and other actors that occupy it requires some form of computer vision and this makes it an interesting use case. To understand this use case better, three projects are examined: 1) The Virtual Rap Dancer [22], (2) Obie [8] and (3) The Virtual Trainer [23].
The Virtual Rap Dancer [21] is an on-screen 3D avatar that performs rap moves in response to music and the movements of the user. It detects beats from audio, input from the user stepping on a dance mat and by analysing the poses of the user from a webcam feed in real-time. The information from both channels is then combined and used to control the movements of the agent.

For the visual beat detection, The Virtual Rap Dancer uses a computer vision pipeline implemented in ParleVision4 (see Figure 3.2). The pipeline reads the image from a camera, detects the hands and face of the user and subsequently tracks when they cross certain implicit boundaries.

A “ServerSink” at the end of the pipeline sends the information outside the framework over a TCP socket, where it is combined with the beats detected from the audio and dancing mat for use by the agent.

Obie [8] is a virtual agent that attempts to attract people using non-verbal signals. He lives on a screen in the hallway and waits until someone passes by to look and smile at them. The goal of the project was to research the effects of non-verbal signals on elicited responses and to determine the nature of those responses.

To keep an eye on passer-bys, Obie analysed the pictures from a camera in real-time using a pipeline implemented in ParleVision4. The pipeline, shown in Figure 3.3, reads images from the camera and pulls them through a series of processors that clean up the image, remove shadows and so on. Then, a SimpleTrackProcessor performs blob
tracking to detect objects and their position in the image. This information is then sent simultaneously to two destinations: SaveToAviGaze saves the video when a person is detected for later (manual) analysis and ServerGaze exports the data to a Java-based program controlling the agent’s responses.

**The Virtual Trainer** [23] is an intelligent virtual agent that mimics a real personal trainer that presents the user with physical exercises and provides feedback to the user. The Virtual Trainer is a complex system and Computer Vision is just a small part of its ‘Motion Analysis’ module:

The Motion Analysis consists of a Feature Extractor and Feature Interpreter. The Feature Extractor is a Parlevision pipeline that captures the video and looks for special coloured markers (gloves and socks), normalises them with respect to the user’s body and tracks their position. The Feature Interpreter then takes this data and judges the position according to the exercises.

These applications all have several things in common. Most importantly, the analysis needs to be real-time. If the agent is to respond to a user convincingly, it needs to do so quickly; especially when there is music involved. Also, the computer vision is generally just lightly processed input for a bigger system, where it is fused with other data and used to reason about the world; the relevant part. This is well illustrated with a quote from one of the papers: “The creation of a tracker was not the main goal of this research and there was limited time available to create one”.

### 3.3.3 Computer-Aided (Security) Surveillance

Another area that is of interest, is that of computer-aided surveillance. With security cameras becoming increasingly ubiquitous, a lot of effort is put into processing their data. This data-processing can be done either in real-time to generate alerts, or off-line to perform data analysis or for retrieve information.

One example of such an application is the Quo Vadis Object Tracker [26]. The system, based on ParleVision4, is able to track objects and people in security video footage. The goal of the research project was to use various computer vision techniques, such as the detection of skin, blob detection and temporal analysis; to address challenges in tracking caused by occlusion etc.

Compared to the Interactive Installation and Virtual and Embodied Agent applications, the pipeline used for computer-aided surveillance is much more complex. It contains processors that delay and compare frames to do temporal reasoning, processors that convert color spaces, generate histograms and look for skin color and processors that merge the results of different algorithms to detect objects. Figure 3.4 shows just a part of the pipeline used to separate objects from the background. Unlike other applications, all
research might be done inside the framework, especially if it does not need to run in real-time.

3.3.4 Video and Image Annotation

A final class of applications and use cases is off-line video and image annotation, where information is added to existing video or image segments.

One example of such an application where ParleVision4 was used is the Nijntje detector [11]. The author built a system to detect and extract occurrences of Nijntje, a famous children’s cartoon character, from scans of books. The system consists of two parts: an image analysis stage and a reasoning stage. The image analysis stage was implemented as a ParleVision pipeline that loads the images one by one, runs a set of filters and processors on them and outputs the results to a text file, which are subsequently used in the reasoning stage. The exact nature of the operations in the pipeline are unknown, but include contour finding and joining, shape finding and colour transformations.

What distinguishes video and image annotation from other applications is that it is likely not real-time, off-line and that the same filters might be run against many different input sizes and formats, while providing output for each of them so that it can be used in a later stage.
Chapter 4

Analysis of the ParleVision4 framework

Parlevision4 has served the Human Media Interaction group at the University of Twente well for several years. It has delivered on its promise of being extensible since it has seen much new functionality added since its inception and has been successfully used for a number of projects. Parlevision4 does however suffer from a number of issues. This chapter analyses the strong points and weak points of the Parlevision4 architecture and implementation.

4.1 Parlevision4 architecture

4.1.1 The pipeline model

Parlevision4 uses the model of a processing pipeline, in which data is routed along a graph. The graph does not need to be acyclic and as such may contain feedback loops. The nodes of the graph consist of discrete processing functions which can have multiple input and output ports called pins. The edges of the graph consist of directed connections between these processors’ input pins and output pins. The complete graph is called a processing pipeline or pipeline for short.

The basic building block of the pipeline is the processor: all processors inherit from AbstractProcessor. There are two subclasses of the AbstractProcessor interface: PipelineProducer and PipelineProcessor. Producers produce data items which are fed to subsequent processors or consumers in the pipeline. AbstractProcessor contains the basic functionality needed for both producers and processors and define an interface which derived classes need to implement.

A single pipeline processor is usually a direct mapping to a single OpenCV function and is as limited in functionality as possible. Complex functionality arises from composing a pipeline from multiple processors. This enables the user to compose a pipeline in a similar fashion to how one would build a custom piece of code using OpenCV functions. More complex processors are possible when needed and a plugin system is defined to allow the program to load extra functionality at run-time.

For passing data along to other pipeline processors, Parlevision4 defines input and output pins. The number of pins and the pins themselves are defined at compile time by an id chosen by the programmer. The application does not check the correctness of the id and as such this is the responsibility of the programmer. There are three pin types, null pins, output pins and input pins. Output pins hold a pointer to a buffer which should be allocated and initialised by the pipeline processor at initialisation time. Input pins get a pointer to the pointer to the output pin buffer of the connected processor at processing
time. Pin data is in-place and does not move along the graph. This entails that when a processor wants to just pass data from its input pin to its output pin and do nothing with it it needs to copy the buffer at the input to the buffer at the output. Pointers passed are not validated.

4.1.2 The type system

To verify the data passed along pins ParleVision4 defines a type system. The ParleVision4 type system is very limited and not extensible. All known types are defined in an enumeration. If one were brave enough to add a new type one has to edit the code in multiple undocumented places and recompile the entire application. Type safety is guaranteed because the pin type is checked to be equal when making a connection between two pins. There is no mechanism to pass data along as type-less data. Since C-style casts are used and void pointers are passed there is no way to do dynamic casting. This means that while processor functionality need not be restricted to OpenCV functions and processors can use 3rd party libraries, the data types which can be passed along are very much restricted. This limits greatly the extensibility of the whole design. There is no serialisation support for data types.

4.1.3 The execution model

The execution model of the pipeline and as such the scheduling of pipeline processors and synchronisation between them is very basic. A valid pipeline needs to contain a single producer, having more than one producer in the pipeline is not supported. The execution of the pipeline always starts with this producer and is driven by the producer. Producers use a callback function which is usually defined external to the framework. For instance, the camera producer uses the now obsolete camera callback function of the CvCam library and the image loader loads images using a time callback which is called by the Win32 API on time-out of the timer.

A processor is called when all dependencies of the processor have been resolved, which means when all processors connected to the processor’s input pins have executed with the exception of optional pins. All processors are synchronous, there is no support for asynchronous processing of events. Only one processor can be processing at a time. A processor is required to write a result to all its output pins, but this is not checked nor enforced. As such, while all inputs are assumed to be valid, this can also not be checked or enforced. As long as all processors have initialized their output pins correctly, this does not lead to a crash but can lead to invalid data being processed. In the case of a cycle in the graph, this entails that the processor which gets the feedback from later on in the pipeline will get invalid data for at least the number of processors in the cycle.

Most computers today have multiple CPU cores: for workstations the presence of 4 cores has become the standard, while servers can have up to 16 cores per socket at the time of writing. ParleVision4 was originally developed in a time when systems would often have just a single processing core. The architecture contains some fundamental problems which prohibit parallel execution. The lack of parallel processing features serialises pipeline execution and limits the usage of processing power to just a fraction of the processing power available. In several projects this became a bottleneck when using ParleVision4 to do real-time high resolution computer vision.

Individual programmers have tried to solve this issue themselves by multi-threading single pipeline processors, but this is not a solid solution and introduces a new set of problems. Because pin data is in-place and not buffered a processor which is executing and writing into its output pin data buffer is also writing into the input pin data buffer of all processors connected to this output pin.
If one was to parallelise the execution of the pipeline, the data buffers at the pins would have to be buffered or protected by mutexes, which was the approach used by the previous version of the EyesWeb framework. Viewers of data are synchronized with the pipeline, since they use the same buffer, and would lock the pipeline as well.

Another consequence of the fact that pin data is in-place is that when the format of the input data changes, the processor is responsible to detect this and reinitialise the data buffers accordingly. For example, when the camera changes resolution, the images passed along the pipeline also change resolution and thus the buffers holding the image data need to be accommodated accordingly. Since the changing of the resolution was outside the program control, this has lead to the addition of check input functions, which not all processors have, leading to unexpected crashes.

4.1.4 Serialisation support

Serialisation of the pipeline processor and its parameters is done with a callback function in the code of the processor. This makes the processor programmer responsible for correctly handling serialisation. For instance, when adding a member variable to the class, the programmer has to remember to also add this variable to the serialisation function. As a consequence of this, many processors do not correctly serialise and restore their state of deserialisation. The serialisation method is specific to XML and a specific XML library so changing the serialisation format involves changing the code in all processors. As such serialisation to another format is not easily supported. Ideally, one would like the processors to use reflection to serialise themselves, using compiler knowledge of the class instead of requiring the programmer to take care of it manually.

4.1.5 Extensibility

ParleVision4 was designed to be an extensible framework, as such it supports the loading of plugins at run-time. These plugins can contain both PipelineProducers and PipelineProcessors. Because of the limits of the type system, it is not possible to load new types at run-time. Also the data type viewers are not extensible. This has limited the use of other libraries in plugins since their types can not be passed along to other processors or visualized at all.

4.2 The user interface

ParleVision4 contains a rudimentary graphical user interface that allows the user to compose pipeline graphs from existing processors, without writing code. Here too, there is room for improvement; both from a usability as from a technical point of view.

The GUI, shown in Figure 3.1, consists of a single window with a menu, three panes and a status bar. The menu provides access to loading and saving of pipelines, adding new processors and contains buttons for play, and pause. The three panes show the pipeline graph, the output of selected processors and the system log output.

The pipeline graph is generated automatically when elements are added or removed, but allows for only limited direct interaction. Connections between processors are made indirectly through a pop-up window with drop down menus of pins and the processors cannot be rearranged. The pipeline is always drawn from left to right, in the direction of the flow of data. Adding connections between two processors results in the graph being redrawn. This is not animated and can therefore sometimes lead to unexpected results.

Some processors have parameters that can be configured. This is done by double clicking on the processor’s name, resulting in a pop-up window with contents provided by the processor implementation.
The user can view the output of each processor pin in real-time by double clicking on its name. The processor’s output is then shown in the middle pane in one of three view ports. Up to three pins can be inspected at once. The three view ports fill up from left to right, after which a fourth output pin is shown in the leftmost view port is recycled. The view ports cannot be reordered and it can be cumbersome to control which output is shown.

Finally, the log pane and status bar show respectively what the processors are doing (through log statements in the processor’s code) and whether the system is in a running or stopped state. The window has a fixed size; it is not possible to resize it.

In addition to the usability quirks mentioned earlier, there are some technical problems with the current GUI implementation. First of all, ParleVision4 has not been designed according to the Model-View-Controller (MVC) design pattern. As such, there is no real separation between the code responsible for rendering the user interface, checking the user input and the rest of the code. The GUI drawing code is included in the base class of each processor, and each sub classed processor has to define its own GUI callback function. There are issues with the implementation of the GUI which causes the whole application to become unresponsive at times. Because of the lack of separation between GUI code and other code, these issues have been very hard to track down.

The GUI is programmed using the Win32 API and the Microsoft Foundation Classes (MFC). This means that ParleVision4 is bound to a single platform and user interface: Microsoft Windows. Preferably, the framework should be platform agnostic as other platforms such as Linux and Mac OS X are used inside the university departments and by students.

Pin data viewers are not pipeline processors and are separated from the pipeline model. Because data is not buffered or locked, they are synchronized with the pipeline and redrawn after each processing call. This introduces another dependency between the user interface and the scheduling of the pipeline. Just like the type system, the data type viewer relationship is hard coded and not extensible.

Event based asynchronous viewers which would not have to be redrawn after each processing call would give the option to reduce the frame rate of the viewers independently of the frame rate of the pipeline.

4.3 State of the project and code base
ParleVision4 is kept in an internal CVS repository at the department of EEMCS of the University of Twente. When researchers or students need to use or adapt the framework, they are usually given access to the repository, but the source code is generally not available outside of the university. It is also unclear what the license of the framework is or what its terms of use are.

There has been virtually no project management in place. Since its original conception, multiple people have made changes to the source code in an uncoordinated fashion. Those changes have mostly been in the core ParleVision4 libraries because of a lack of understanding of the plugin infrastructure and the lack of documentation on how to extend the framework properly. They mostly involved adapting the software to specific needs or tasks, preferably with as little effort as possible, without a clear vision on the direction of the framework. Because of this lack of project management, the code base of the framework is in a dire state and in need of maintenance.

Because of the state of the code base, the framework can only be compiled using an old compiler (Visual Studio 2003) and it depends on an old version of the DirectX library and an old version of the OpenCV library (version 1.1) which is incompatible with Windows version which enable Data Execution Prevention (DEP) such as Windows 7. This makes
it increasingly harder to use it to build new applications.

There are many known issues and bugs in the ParleVision4, but these are not tracked anywhere except for TODO statements in the code itself. This has made the code increasingly harder to understand and maintain over the years, because of the multiple TODO’s scattered around the source files. The whole code base contains many places where code has been duplicated, so that it is now very hard to make changes to the framework without having to alter many processors.
Chapter 5

ParleVision5: Design and Architecture

As chapter 4 discussed, there are several problems in the ParleVision4 code base and architecture. While the pipeline model has been a strong point of the ParleVision4 architecture and enabled the implementation of new processors and the reuse of old processors for use in new projects, the way it is implemented causes serious problems. Some of these problems can be fixed, while other more fundamental problems cannot without massive rewrites to the code base. If we want to build a framework to exploit modern hardware, what is really needed is a pipeline architecture built for parallel scaling on multi-core hardware. Fixing the current model is difficult to do, more so because of the sorry state of the code base. ParleVision5 has therefore been redesigned from the ground up.

The framework as a whole has been separated in different modules, so there is less interdependency between different components (see Figure 5.1). The data type system has been revised so data types and their viewers are extensible and loadable at runtime via a plugin system. They were also given a serialisation mechanism so they can be easily serialised and deserialised. The pipeline as a whole can be serialized through reflection; programmers no longer need to explicitly write code for this. Additionally, the user interface has been completely redesigned to be easier to use and reimplemented with a revised architecture in a cross-platform user interface toolkit. This chapter discusses the design choices and architecture of the new ParleVision5.

5.1 Goals and System Requirements

Based upon the analysis of the ParleVision4 framework the goal of this project is the research and development of a new framework featuring a new pipeline architecture which is able to scale on modern multi-core CPUs. All parts of the framework should be completely
extensible without modifying the code of the framework allowing custom processors which use other libraries or features to be loaded at runtime. Users working with ParleVision4 can easily migrate to the new framework. The framework will be cross-platform and its code will be available under an Open Source Software (OSS) license and hosted on a public source code repository where issue tracking and bug tracking will also be done. This will hopefully foster a community around the project and expose the University of Twente’s name to the CV community.

These goals have been reformulated into system requirements and can be found in the appendix A.

5.2 Programming language and application development framework

ParleVision4 was written in C++ using a C like style and uses the MFC library and the C based OpenCV library. OpenCV has been moving from C to C++ for its primary interface and most new features are implemented in C++. There are bindings for OpenCV to other languages, but these generally lack the newest features and the move to C++ has made the making of bindings to other languages more difficult.

Because ParleVision5 builds on the strengths of OpenCV and most other CV libraries are written in C++ the language choice for ParleVision5 is the 1998 ANSI/ISO C++ standard informally known as C++98.

To fulfill the requirements of cross platform support we chose the cross platform C++ Qt application development framework. Qt is based on C++ but adds some extensions to the C++ language such as reflection support. It supports all major platforms and delivers a very acceptable native like look-and-feel on all platforms.

5.3 The processing pipeline

5.3.1 The pipeline model

ParleVision5 builds on the proven model of a processing pipeline as defined in ParleVision4, in which data is routed along a directed graph, where the nodes of the graph consist of discrete processing functions and the edges of the graph are directed connections between these functions’ input and output ports. To simplify the implementation of the scheduling algorithm, cycles in the graph are not allowed, making the graph a Directed Acyclic Graph (DAG). This is in contrast with ParleVision4 which did allow cycles in the graph. Future version may choose to reintroduce feedback loops into the pipeline model.

The basic building block of the pipeline has been renamed from AbstractProcessor to PipelineElement. This decision was made because the term AbstractProcessor was also the base class of the PipelineProducer, which is not a processor. PipelineElements define an interface which derived classes need to implement. This interface defines initialisation and deinitialisation methods and calls certain methods when the pipeline is started and stopped. This allows processors to initialise resources and deinitialise resources. The basic framework supports PipelineProcessors and PipelineProducers, where producers produce data items which are fed to subsequent processors in the pipeline.

One of the goals of the new pipeline model was going from a single threaded to a multi-threaded pipeline model. To enable parallel processing of pipeline processors, a number of techniques are used. These include a new type system, a parallel execution pipeline model and shared resource management.
5.3.2 The type system

One of the biggest flaws in the extensibility of the ParleVision4 was the type system. ParleVision5 therefore introduces a new type system, with extensibility and thread-safety as primary design goals.

When broadcasting data on output pins to multiple processors which can all be processing concurrently and to an unknown number of viewers, the most important issue is the thread-safe sharing of that data. To solve the issues with the ParleVision4 method of passing raw pointers to programmer managed buffers and to prevent badly programmed processors from introducing hard to detect data corruption bugs, pin data is no longer in place and moves along the pipeline. Processor programmers are not responsible for managing, allocating and deallocating static buffers for pins. Because data is moving along the pipeline, is buffered in the connections between pipeline elements and sent to viewers using asynchronous signals, there is no longer strong ownership of the data. To solve the issue of who has to deallocate the data, the data is automatically deallocated using a reference counting technique.

Thread-safe reference counting can be implemented in a number of ways. One way is to hold a reference count inside the object which is increased when an object needs to be retained somewhere. A smart pointer which is allocated on the stack can then be used to increase and decrease the reference count automatically. A disadvantage is that using this approach with the signal and slots mechanism in Qt breaks the inheritance structure and automatic down casting leading to lots of casting code. Another way to do reference counting is to use a smart pointer which stores the reference count inside the pointer itself, increasing the count when the pointer is copied. ParleVision5 uses a combination of these techniques.

As a reference, EyesWeb has a type system where all types are based on a single base class. To guard the shared data from being accessed by multiple threads it is protected by a mutex. This has the disadvantage that classes which are defined in external libraries cannot be passed between processors without writing a wrapper around them which derives from the base class. Also, primitive types such as boolean and integer need to be wrapped in a class which derives from the base class leading to unneeded overhead.

To support dynamic pins which do not know their data type at compile time, there needs to be a way to determine the type of data at run-time. Because of the way C++ is designed it lacks a base class such as the Java Object class and it does not support dynamic casting on void pointers. Passing data as void pointers would also remove the possibility to pass data as value type. A solution to this problem is to define a complex union type, a variant, which can serve as a data container for passing along data. Qt defines such a variant type in its meta object system called QVariant. All of Parlevision’s shared data classes use the Qt meta object system, which allows them to be stored in QVariant. The advantage of this scheme is that all of Qt’s classes such as QString or QSet can automatically be passed along the connections in the pipeline without writing any additional code. All primitive types such as int, long and double can also be passed along as value types.

To make it possible to use complex data types safely in multiple threads, Parlevision5 uses a technique which is also used extensively in the Qt library called implicit data sharing which implements a thread safe copy-on-write technique. The data elements consist of a header object and an object which holds the actual data. The header object does not contain any data and is passed by value, where its copy constructor and destructor manage a reference count which is kept in the data object, increasing it when a header is copied and decreasing it when a header is destructed. When a thread writes to a data element which has a reference count higher than one, the data is implicitly copied. This copy-on-write scheme enables data to be shared between multiple processors, guards data integrity.
and does so implicitly.

Because ParleVision 5 builds strongly on OpenCV, one of the most important data objects for Parlevision 5 is the CvMatData data object. It is an encapsulation over the OpenCV Matrix (cv::Mat) data type as defined in the new OpenCV C++ interface. The encapsulation implements implicit data sharing which makes the sharing of OpenCV Matrix data in the pipeline much easier and guarantees data deallocation. The OpenCV Matrix type is used for all data matrices including images.

5.3.3 Producers

While the ParleVision 4 framework only defined a single type of producer, which was real-time and synchronous, the ParleVision 5 framework defines two basic types of producers, real-time and non-real-time.

Producers in the ParleVision 4 framework are driven by a programmer defined callback function which could be external to the framework. This callback would not return until all data was processed, which could introduce non-real-time behaviour. In ParleVision 5 producer are polled using the non-blocking readyToProduce method and as long as the producer returns false, the process method will not be called. Producers therefore never have to produce faster than they can supply data. The polling rate is controlled by the framework’s heartbeat, which is configurable and is at 100Hz by default.

Real-time producers are producers which are fed by a real-time device such as a camera. The camera’s frame rate will determine the speed at which the producer produces. Because the frame rate might be higher than the processing pipeline can sustain, it is up to the producer to take this issue into account and drop frames accordingly. For instance, a camera producer might only produce the latest frame when the process method is called or a movement producer might interpolate several movements and send the interpolated movement.

Non real-time producers are producers which are fed by a non-real-time device such as a directory of pictures on a hard disk. These producers never produce faster than the processing pipeline can sustain.

The distinction between real-time producers and non-real-time producers is not made in the current implementation of the framework. As mentioned, real-time producer have to solve the issues with overproduction themselves. This entails that when multiple producers are used in a single pipeline, all producers have to be ready before process is called on any of them. There is no explicit synchronisation between producers, which has some implications for pipeline design. For instance, if two cameras need to be synchronized to match frames exactly, one should use a single stereo camera producer which drives two cameras and handles the synchronisation between the cameras itself. Combining producers for asynchronous events in with synchronous producers forces the synchronous producers to wait for the asynchronous producer to be ready. This is a limitation which might be corrected in a future version when support for a distinction between synchronous and asynchronous producers is added.

5.3.4 Processors

Like the ParleVision 4 framework, pipeline processors are where the actual processing work takes place in the processing pipeline. The way data is delivered is completely different however. As mentioned, data types are now moved along in the pipeline using reference counting to tell when to deallocate. The processor programmer does not have to reserve any buffer at its output pins to store data any more such as was the case in ParleVision 4.

Processors can take input from one or multiple input pins, process it in their process method and output it to one or multiple output pins. The processor programmer generally
has to implement the process method in which the actual processing is done. This method
is called by the framework when the processor is ready to process, which depends on the
type of processor which is determined by the types of input pins it has defined. There are
two basic input pin types: synchronous and asynchronous. Processors can have multiple
input pins of different types defined.

Synchronous pins are the default option and the easiest to use. The framework will
guarantee that when process is called on a processor, input is available on all synchronous
input pins. This data is synchronised meaning that all pins have data originating from
the same production run. Synchronisation is done internally in the framework with serial
numbers. When producers are called by the framework, all data they produce will have the
same serial. The processor will not allow process to be called before all data available on all
synchronous input pins has the same serial number. Generally, the processor programmer
does not have to care about serial numbers since they are transparently handled by the
framework.

Asynchronous pins require more effort from the processor programmer. The framework
does nothing to guarantee that data is available on an asynchronous pin or that if data is
available it has the same serial number as data available on other pins. This is up to the
processor programmer to verify.

Processors can use the `put` method to put output on the output pins. Only one object
can be put on an output pin per process call. To output tuples or collections of data
e.g. multiple images one has to use a data type which can store multiple objects such
as a set. Calling the `put` method more than once in a single process run will result in
an error. Output pins are always synchronous in the sense that they will always produce
output. When the process implementer does not explicitly put output on the output pin,
the framework will insert a NULL data item with the correct serial number in that place.
This ensures that there are no missing serial numbers in the pipeline.

Processors should use the method `get` to collect data from their pins. Failure to do
so will trigger an error and the framework will stop the pipeline. Calling `get` to retrieve
data is mandatory for all types of pins. On asynchronous pins the processor programmer
will have to check if data is available and call `get` at least once if it is. For synchronous
pins, `get` should be called exactly once on each pin. Calling `get` twice will result in an
error on synchronous pins. There are three basic pin mixes possible: a processor with only
synchronous input pins, a processor with both synchronous and asynchronous input pins
and a processor with only asynchronous pins.

For all processors with synchronous pins, whether they have asynchronous pins or not,
the framework determines if it is ready by checking if data is available on all synchronous
pins. Asynchronous pins are completely ignored by the framework when one or multiple
synchronous pins are defined. It is up to the processor programmer to check if data is
available on an asynchronous pin manually. When a processor only has asynchronous
input pins, it is determined to be ready by the framework as soon as there is input on at
least one of the asynchronous input pins. Again, it is up to the processor programmer to
check which of the pins has data available and handle all possible cases.

5.3.5 The execution model

The pipeline model is now safe for parallel execution. Because data items are buffered
and protected, we can now promote the pipeline to a true pipeline where multiple work
units are in different stages of being processed concurrently. A single processor can not be
executed concurrently because its process method is not reentrant. The same limitation
holds true for producers.

The dispatcher manages multiple threads which execute queued processors. Generally,
an ideal number of threads is created were this number is dependent on the hardware,
mostly a single thread per CPU core. Techniques such as Symmetric Multi Threading (SMT) which most modern Intel processors have will increase the ideal thread number. The framework polls processors continuously if they have valid input according to the rules explained in this chapter. When a processor has valid input it is queued for execution in the dispatcher. The framework will queue a processor only once, because the process method is not reentrant and the dispatcher itself does not do any checks to see if a processor is already running or not. After execution the return value of the process function is checked and if an error is detected the execution is stopped, if not the processor is available for queueing again.

The dispatcher dispatches work units to be executed on threads managed by a global thread pool. It is used to schedule work units from the pipeline as well as the GUI. GUI work units are for instance the viewers, which sometimes have to convert or process data in order to display it.

![Figure 5.2: A typical pipeline layout](image)

5.3.6 Serialisation support

In ParleVision4 serialisation of processors was supported by implementing a callback function in the processor class where each member had to be explicitly added to an XML file. Because this relies entirely on the skill and discipline of the processor programmer, many serialisation functions were incomplete. Also, because an XML node was passed in this function the only possible serialisation format was XML and it was hard-coded to a single library. ParleVision5 supports automatic serialisation of class properties by using the introspection features Qt adds to C++. Using the Qt meta object information class properties and their types can be queried at run-time which allows different back-ends to serialise the processor class properties to different formats. The introspection support allows the complete separation of the serialisation code from the processor implementation code. While the current implementation only features XML pipeline serialisation, new back-ends can be added to support binary or plain text serialisation without changing the code in the processors itself.

5.3.7 Extensibility

As described in this chapter, the ParleVision5 framework is designed from the ground up with extensibility in mind. Because of the new type system and the MVC design, data types, processors which use them and data type viewers can be added to the framework at run-time. External libraries can be easily integrated into ParleVision5 because of the use of the new type system, which lacks a common base class and as such can use types defined in external libraries. To facilitate this extensibility at run-time, ParleVision5
features a plugin loader. Processors, data-types and their viewers can all be defined in shared object files which are loaded when the program starts. Because the serialisation code is now separated from the processor implementation, processors added at run-time can be serialised to various formats with back-ends which can also be added at run-time.

5.4 The Graphical User Interface

As concluded in Section 4.2, the old ParleVision left some room for improvement of the Graphical User Interface, in both usability and architecture.

5.4.1 GUI Design

The interface, Shown in Figure 5.3, was given a complete make-over. The application is now a document-based editor, much like Microsoft Word or Inkscape. The main view in the center of the screen is the pipeline editor, or scene canvas, and it is surrounded by a menu- and toolbar at the top, a status bar at the bottom and several utility panels containing the library, log, configuration and output of the pipeline or selected items.

Assembling Pipelines: The Canvas and Library

The user assembles the pipeline from a library panel (shown on the left) by dragging the elements onto the canvas and arranging them at will. The drag and drop-style interaction works much like that in many vector graphics editors. Making connections between elements is done by dragging a line from the output pin of one element to the input pin of another. The connections are made “live”: the pipeline is constructed while processors are added to the scene. As a result, any errors are caught at the time they are relevant and the user interface shows the status of the elements (green for OK, red for incomplete) in real time.

Having the ability to rearrange the pipeline at will, rather than doing this automatically based on the underlying data, has the advantage that it is much more predictable.
In ParleVision4, pipeline elements occasionally jumped unexpectedly while making connections. Additionally, the pipeline graph can be organized spatially across the whole canvas, rather than being fixed from left to right; something that makes sense especially for parallel pipelines.

The direct manipulation metaphor of the pipeline scene canvas also extends to adding new elements: whereas in the old system this was done through a menu option and a dialogue window, in ParleVision5 the user can browse visually through the available elements and drag it from the library onto the canvas. The input and output pins are shown in the library panel, so it is easy to determine the nature of the element.

**Configuring the pipeline: The Inspector**

Another big change in the GUI is the way pipeline elements are configured. In ParleVision4, this was done by double-clicking on the element’s name, which showed a pop-up settings window. Changes to the element’s configurable properties could be made in that window and were applied when the user hits “OK”.

In the new GUI, these settings windows are abolished in favour of a new “inspector” panel. This panel is based on a user interface convention prevalent in Mac OS X applications. A single “inspector” panel is always active and shows the information of the current selection. The user can inspect multiple items by selecting them in sequence. The disadvantage of this pattern is that the user can only inspect a single item at once. On the other hand, the inspector window stays in the same predictable place, does not get lost or occluded behind other windows and the user does not have to close and reopen the individual windows all the time.

The user can select anything in the pipeline graph, but currently it only shows a configuration tab for elements. Changes made to the parameters are applied instantly, so users can fiddle with the controls until satisfied. If the processor makes changes to the parameters on its own, these changes are also immediately shown in the inspector. This can be useful for monitoring the state of a processor element.

Finally, the inspector panel provides multiple tabs. Currently, only the configuration and information tabs are implemented, but building on the concept of inspecting a single thing, it is not hard to imagine future extensions. Viewing the log of a single pipeline element, having quick access to the output of all its pins or getting an overview of its performance metrics are all valuable additions.

**Viewing output**

The transformations performed by the different processors in the pipeline are generally highly visual. As such, it is important that the user can view the output at each step to see what is happening and determine if, for example, certain parameters need to be adjusted.

Much like in the previous version, the output of any output pin can be viewed by double clicking on its name. What has changed, is that rather than appearing in a fixed three-viewport panel, each pin gets its own “viewer” panel. These panels can be ordered by dragging them around, and there is no limit on how many of them can be shown at any time.

Also new, is that processors can now output non-image data, as discussed in more detail in Section 5.3.2. Viewing the output of non-image data pins works exactly the same, but the way the data is rendered differs.
Cross-Platform and Native

One of the requirements was that the framework was not tied to a single platform, but could be used on multiple operating systems. That includes the GUI. Section 5.2 elaborated on our choice for Qt as a cross-platform application framework. As it happens, Qt has an excellent GUI and widget toolkit with support for major operating systems.

Thanks to Qt, the UI widgets automatically gets a native look and behaviour on Windows, Mac OS X and Linux. This means, for example, that on Windows the menu is located below the window title, whereas on Mac OS X the menu is by convention at the top of the screen. The same is true for things like the open and save file menu, which are native to each platform. Qt also takes care of ensuring that the “recent files” menu that was implemented, works the same on all platforms.

Of course there is also a down side to developing one GUI to run on all platforms. Users of different platforms are used to different paradigms and different operating systems have different conventions for solving the same problems. For example, on Mac OS X dialogues are generally phrased as a question with buttons for the different answers [5], while Windows presents the user with the options “OK” and “Cancel”. While it is best to consider each platform separately while designing, this was not feasible or desirable for this project.

In some design decisions, we simply went with one option. The “inspector” panel might be a bit unusual for a user of a platform where this is not a common paradigm, but it is so explicit in its behaviour that even users that have never seen this type of interaction before can see how it works with some trial and error.

In other choices, an explicit difference in behaviour was chosen to accommodate for different platforms. Windows users, for example, often work with a single window that contains all the controls in the interface. Mac users, on the other hand, are used to having a lot of separate windows. As a result, the library and inspector are floating windows on Mac OS X, whereas on Windows they are “docked” on the left and right. Qt allows windows to be dockable and as such, the user of either platform can customize their set-up and choose to have the library, inspector, log and viewers either as floating windows, or docked inside the main window.

5.4.2 GUI Architecture

Rewriting the GUI from scratch allowed many of the technical problems of the old framework to be fixed. As explained in Section 4.2, the GUI code in ParleVision4 was mixed with the framework and processor code. This lack of separation between GUI and logic resulted in concurrency problems, where the interface becomes unresponsive, and makes it difficult to write good extensions.

In the new framework, all code adheres strictly to the Model View Controller (MVC) paradigm. The GUI is but a view on a model (the pipeline) that it can modify through controllers. Models have no knowledge of the GUI and can be written in complete isolation, without having to worry about what the GUI might look like.

Additionally, Qt extends C++ with mechanisms such as signals & slots and reflection, that make it very easy to glue components together at run-time without having to write a lot of code and even allows us to generate a rudimentary configuration UIs automatically. This makes it very easy to write processors and other pipeline elements for ParleVision5. The developer simply declares the properties that can be configured and the framework automatically generates UI widgets and wires them into the pipeline element. A developer can always choose to privide a custom UI, in an MVC-adherent manner.

When a user clicks on a pin to view its output, the framework automatically determines its data type and creates a corresponding DataRenderer (a view) to observe its content.
(the model). Because this is done in an asynchronous manner, the pipeline does not wait for the UI to do its job and the processing does not slow down like it would if rendering and processing were intertwined.

The strict separation of views and controllers from the model layer has another consequence: The framework can run completely headless, without any views and without any GUI. This allows the processing to be run, for example, in a background process of a powerful server. It also means that the GUI can be easily replaced or swapped out by something else. One can simply link their own program to libplvcore and use the library in their own code.
Chapter 6

Evaluation

Comparing the new ParleVision5 to its predecessor, a lot can be said. As most of the improvements have already been discussed in Chapters 4 and 5, this chapter compares just the most important differences in usability and performance.

6.1 Usability

One of the most important goals in creating ParleVision5 was making a framework that is easy to use for novices and non-programmers. This translates in both a redesigned graphical user interface and a revised API.

6.1.1 The GUI

Where the graphical user interface of ParleVision4 was cumbersome and data-oriented, requiring the user to add and connect pipeline elements through drop-down menus, the new GUI allows much more direct manipulation. Adding elements is done using drag and drop; making connections means simply drawing a line from an output pin to an input pin and clicking elements selects them for inspection and configuration. The visualisation of the pipeline is also a lot richer; allowing the user to rearrange the components at will to clarify their relations.

Many of the quirks present in the old ParleVision are also fixed. The views on processor output can be rearranged, the GUI does not freeze all the time and responds as expected and can be resized by the user rather than being fixed to a predefined size.

The GUI has the ability to run on other platforms than just Windows such as Mac OS X and Linux and integrates well with the native look-and-feel on all platforms supported.

We feel these improvements make the UI a lot more intuitive. We have not done any formal evaluation to confirm that it is easier to use; this is left as future work.

6.1.2 The API

Another significant part of the user experience of a framework is writing code. ParleVision5 makes programming extensions a lot easier. It takes care of difficulties such as memory management locking, scheduling, so the user can focus on writing just the code that contains the processing logic.

The plug-in API is simple and adding a processor is a simple matter of filling in the blanks by overriding and implementing some methods. ParleVision4 required the user to manually check image formats etc. in the process method, or else the program would crash. The user still has to check for these, but there is now a logical place to do it. There is also no longer a need to write serialization support: the framework handles this
automatically. This ensures that all pipelines can be saved and loaded; they never break because serialization code was omitted, as used to happen with ParleVision4

The new framework is also more modular and uses the Model-View-Controller pattern to separate GUI from logic. This results in code that is more organized and does not require GUI code to be interwoven with processor code. ParleVision5 can generate a rudimentary configuration interface for processors using reflection, so in most cases the user does not even have to provide a custom user interface. A simple processor in ParleVision5 can be implemented in just a few lines:

```cpp
#include "HelloWorldProcessor.h"
#include <plvcore/CvMatDataPin.h>
#include <opencv/cv.h>

using namespace plv;

HelloWorldProcessor::HelloWorldProcessor() :
m_someBool(true)
{
    m_inputPin = createCvMatDataInputPin("input", this);
    m_outputPin = createCvMatDataOutputPin("output", this);
}

HelloWorldProcessor::~HelloWorldProcessor() {}
bool HelloWorldProcessor::init() { return true; }
bool HelloWorldProcessor::deinit() throw () { return true; }
bool HelloWorldProcessor::start() { return true; }
bool HelloWorldProcessor::stop() { return true; }

bool HelloWorldProcessor::process()
{
    assert(m_inputPin != 0);
    assert(m_outputPin != 0);

    // get input on input pin
    CvMatData src = m_inputPin->get();

    // allocate a target buffer
    CvMatData target = CvMatData::create( src.properties() );

    // do a flip of the image
    // m_someBool is a GUI controlled class property
    const cv::Mat in = src;
    cv::Mat out = target;
    cv::flip( in, out, (int)m_someBool);

    // publish the new image
    m_outputPin->put( target );

    return true;
}

31
6.2 Performance

Being optimized for cross-platform, multi-threaded operation rather than an input-driven
Windows-only pipeline makes ParleVision5 a rather different beast than its predecessor.
This section nonetheless compares the two in terms of performance.

6.2.1 ParleVision4 versus ParleVision5

Unfortunately making a quantitative performance evaluation of the ParleVision5 framework
compared to the ParleVision4 framework turned out to be very hard. We did not
succeed in running the ParleVision4 framework and the ParleVision5 framework on the
same hardware specifications, compiler and operating system within the time set out for
this research. Casual observance does hint at a large performance advantage for the multi-
threaded ParleVision5.

6.2.2 ParleVision5 single thread versus multiple threads

As described ParleVision5 has been designed from the ground up to use multi-threading.
It uses a dispatcher which dispatches work units to be executed on threads managed by
a thread pool. To test the speedup of using multiple threads the thread pool has been
configured to use 1, 2 or 4 threads. The pipeline used is a pipeline build to do blob
detection in a video of small creatures called asellus aquaticus on the bottom of a tank of
water. ParleVision5 processes the video as fast as it can, there is no limit on the frame
rate. The pipeline consists of a video producer, which loads frames of an AVI video and
decompresses them. Next a Gaussian smoothing operation is performed on the input
image followed by an average function which takes the average pixel value of 10 frames
to suppress noise in the output. A single image producer, which loads a pre-calculated
average background for this particular video. The background subtractor takes the pre-
calculated average background and subtracts it from the averaged video input leading to
a difference images. This difference image is dilated and eroded to remove single pixel
noise, converted to black and white and thresholded to a binary image. A mask is used
to monitor the algorithm as it combines the original image with the binary image to leave
only the pixels of interest. Finally, a blob detector is run on the binary image which
detects and labels blobs. Normally a blob tracker processor is used to track the blobs, but
this processor was unavailable at the time of running these benchmarks. The complete
pipeline can be seen in figure 6.1.

The benchmarks are performed on an Intel Core i3-370M processor. This processor
runs at 2.4Ghz and has 2 physical cores with SMT, so there are 4 hardware threads. SMT
is a nice addition to the processor, depending on the workload it should yield about a 20%
performance improvement on average. So we should expect a maximum performance of
about 2.5 when scaling from one to four threads. For each thread count the video was
processed multiple times and the average was taken.

The results of the benchmark runs are listed in table 6.1. As can be seen from the
results the performance scales with the number of threads used. Going from one thread to
two even results in a speedup factor larger than two. Going from one thread to four results
in a speedup factor of almost 4. These benchmarks results are therefore better than should
be expected, which probably means that the single threaded performance is lower than
should be expected. A probable cause is the scheduling algorithm used in ParleVision5
which is very rudimentary in the current implementation. Running a single thread still
incurs the overhead of the locking of shared resources, which could explain the scaling
factor being larger than two going from one to two threads. To determine accurately the
quality of the ParleVision5 multi-threading implementation and the overhead it incurs it
Figure 6.1: Screenshot of ParleVision5 pipeline used to benchmark

<table>
<thead>
<tr>
<th>number of threads</th>
<th>frames per second</th>
<th>relative performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>2.2</td>
</tr>
<tr>
<td>4</td>
<td>37</td>
<td>3.7</td>
</tr>
</tbody>
</table>

Table 6.1: An example of table

should be compared to a pipeline programmed manually using the C++ API of OpenCV. This is left as future work.
Chapter 7

Conclusion

When doing research into Human Computer Interaction, building interactive installations, art expositions or other systems that need to keep an eye on their environment, it is desirable to be able to rapidly prototype computer vision applications.

There are many libraries, frameworks and toolkits that can be used for this purpose. Unfortunately, most are either very low-level – thus requiring the user to write a lot of code, poorly maintained, documented or designed. Other toolkits are limited because they are commercial and/or closed and yet other software is so extensive that it takes a lot of time to get started.

ParleVision4 is a simple computer vision framework that offers both an API for writing extensions and a graphical user interface that allows users to compose a pipeline out of predefined transformation processors. ParleVision4 has been used for many purposes, from interactive installations to virtual agents and even computer aided surveillance. Unfortunately, it suffers from some fundamental problems with regard to its architecture, user interface and implementation.

After an extensive review of ParleVision4, the use cases one might need such a framework for and the related work that exists in this area, we designed a new framework from the ground up: ParleVision5.

ParleVision5 is a cross-platform framework for rapid prototyping of computer vision applications. Like its predecessor, it allows users to compose a pipeline graphically, yet this time completely using drag and drop interaction, and it also allows them to write extensions using a much simpler API. ParleVision relieves the user from much of the memory management, scheduling, locking, serialization, GUI definition and type checking that made developing for ParleVision4 so cumbersome. Additionally, it is more modular, has better separation of code and was designed to make use of multi-core and multi-processor capabilities from the beginning.

This makes ParleVision5 faster on complex multi-threaded tasks, easier to use and more versatile in its application.
Appendix A

System Requirements

1. New pipeline model

2. Parallel paths in the pipeline can be synchronised.

3. A pipeline may have multiple producers.

4. The system allows programmers to write extensions such as processors etc.
   (a) The system has a clear and formalised API for writing extensions.
   (b) The API of the system can be understood by inexperienced programmers.
   (c) The system prevents common implementation errors and makes them easier to detect.
   (d) The system can automatically handle memory management of processor data.
   (e) The system can automatically generate user interface components for configuring processors.
   (f) The system can automatically handle saving/loading (serialisation) of pipeline configuration

5. The system benefits from multi-core and/or multi-processor capabilities.

6. The system abstracts away the managing of threads from the processor programmer, such that no advanced knowledge of multi-threaded programming is needed to program basic extensions.

7. The system can benefit from GPGPU technologies.

8. The system can load additional pipeline producers and processors dynamically (at run-time) using a plugin architecture.

9. The system allows new components such as processors and producers to be added without requiring modifications to the framework itself.

10. The system allows new data types to be added without requiring modifications to the framework itself.

11. The system allows new data type viewers to be added without requiring modifications to the framework itself.

12. The design of an extensible data type system which enforces type safety throughout the program and allows for dynamic typing.

13. Data types need to be loadable at run time using a plugin architecture.
14. Data types should be serialisable.

15. Data type viewers should be loadable at run time using a plugin architecture.

16. Processors should be serialisable without the need for serialise functions for every format in the processor itself.

17. The system allows pipelines to be saved to and loaded from a file.

18. The system can be used as a library, for use in other applications.

19. The system can be ran headless, without a GUI.

20. The system can be interfaced with by other programs, possibly running on different platforms and/or environments (networked?).

21. The system works with OpenCV 2.0.

22. The system has a clear migration path for ParleVision 4.0 users.

23. The system should provide a new GUI based on an analysis of the old GUI. This GUI will be separated from the main code base and will use the Model-View-Controller design pattern.

24. The system can be operated by non-programmers.

25. The system allows pipelines to be manipulated graphically.

26. The system allows pipelines to be composed from a library of reusable components.

27. The system allows the user to see the output of individual processors in real time.

28. The system ensures that users cannot compose a pipeline from incompatible processors.

29. The system uses a free, open source license.

30. The system is cross-platform and will work on all three major operating systems: Linux, Windows and Mac OS X.

31. The code will be available under an Open Source Software (OSS) license and hosted on a public source code repository. This will hopefully foster a community around the project and expose the University of Twente’s name to the CV community.

32. Issue tracking and bug tracking should be done at the same place as the public source code repository.
Appendix B

Acknowledgements

We would like to thank Dennis Reidsma, Ronald Poppe, Wim Fikkert and Daniël Tetteroo for their advice, patience and wisdom during the process of designing ParleVision5.
Bibliography


