View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace at University of West Bohemia

COMPONENT-BASED ARCHITECTURES FOR COMPUTER VISION SYSTEMS

Aris Economopoulos, Drakoulis Martakos

Department of Informatics
Hypermedia Digital Libraries – Internet Applications Group (HyDiLib – INA)
National and Kapodistrian University of Athens, Panepistimiopolis, Ktiria Pliroforikis
157-71 Athens
Greece
pathway@di.uoa.gr, martakos@di.uoa.gr http://hydilib.uoa.gr

ABSTRACT

Research performed in the field of computer vision has steadily ignored recent advances in programming tools and techniques, relying on well-established traditional methods, such as Unix-based C programming. While this can certainly be effective, modern computer vision research may benefit significantly from the new tools and technologies that have recently become available. This paper addresses the use of component-based programming methods and proposes a model loosely based on 3-tier architectures, for the creation of robust and reusable computer vision systems, in order to improve code modularity and reusability, and to ultimately foster cooperation between researchers in the field. It outlines a basic design strategy and exposes the benefits and drawbacks of migrating to component-based code. The model is used to build a component-driven framework that is designed based on the principles of 3-tier applications. Its purpose is to aid in the creation and maintenance of stable, dependable testing and development environments. We have listed the main advantages of this approach and have concluded that although the learning curve for the programming skills required is steep, the benefits to be reaped are worth it.

Keywords: computer vision, applications, components, development, testing, cooperation, COM, MTS

1. INTRODUCTION

Computer Vision is an especially active area of research. The main damper on the development of this highly active field was, until recently, the high cost of the equipment required to set up a vision lab. With the advent of new, powerful microprocessors and low-cost high-quality frame-grabbers, this hurdle has been finally overcome. But even in the midst of an era where technological breakthroughs are considered commonplace, many vision researchers still adhere to the past, using what some consider old but well-proven methods of implementation to experimentally justify their theories. Some of the more typical algorithm implementations are hand-written in C, linked to arcane graphics libraries and tested only on some particular flavor of a free Unix clone, such as Linux or FreeBSD. These methods certainly represent viable alternatives for researchers who merely wish to quickly put their theories into practice. However, such methods are gradually proven inadequate when it comes to several key development factors such as code portability and reusability, or when the object is to allow other researchers to duplicate one's results. The computer vision community is a rather active one and there is an undeniable need to facilitate the sharing of information. And while several authoritative textbooks serve to familiarize seasoned and beginning researchers alike with proven concepts and techniques, it is very rare for code to be made readily available in an easy to use format that does not require the user to jump through a lot of hoops just to get it to compile. This situation could be avoided if the community slowly made a shift towards modern programming techniques, such as component-based programming using a widely known object model, such as COM (Component Object Model) [Micro00a] or CORBA (Component Object Request Brokering Architecture) [OMG99a]. The nature of component-based architectures offers several advantages that a researcher might appreciate, such as programming language independence and code reusability, as described in [Kirtl98a], [Malon99a] and [Eddon99a].

In this paper, we discuss the benefits and drawbacks of a possible migration towards component-based programming, specifically concentrating on 3-tier architectures. Based on our expertise with COM, DCOM and their successor, COM+, we present a basic strategy for the design and implementation of modular, robust computer vision systems, loosely based on the 3-tier programming model (data tier, business tier, presentation tier) that is implemented by Windows DNA (Distributed Internet Application Architecture). In the effort to create a basic programming framework for the sharing of reusable components between researchers, we have adopted a set of documentation and programming guidelines that enforce and enhance portability and stability.

In Section 2, we present an overview of 3-tier architectures. In Section 3, we present the benefits of working with components and offer an overview of the programming framework that we propose. Section 5 gives an in-depth view of the framework, including an example that showcases the methodology's advantages. In Section 5 we present our conclusions and future work.

2. AN OVERVIEW OF 3-TIER ARCHITECTURES

The advent of the Internet and the popularity of e-commerce have rendered the basic client-server model of software design obsolete. Programmers are being called upon to produce application code that can scale to serve hundreds of thousands of users and that can work in a completely distributed and often session- and state-less environment. This was a challenge that demanded a paradigm shift in the entire process of application creation, from design on through to implementation. This paradigm shift brought about the concept of 3-tier architectures (Fig. 1). The idea was to design and implement applications in such a way so that features like high scalability and distributed design would be implicitly derived from the design methodology itself. The main difference between 2-tier and 3-tier architectures is the presence of the Business Tier. The first step in the creation of durable client-server applications was the separation of the data services layer from the rest of the application, as depicted below. Separating the core functionality from the user interface was the next logical step, and that is precisely what 3-tier applications are all about: The separation of the Business Logic from the rest of the application. In two-tier applications, the data services layer (or Data Tier) is a distinct logical entity. As such, multiple clients can access it simultaneously, regardless of where in the network it is located. There is no separation between business tier and user interface.

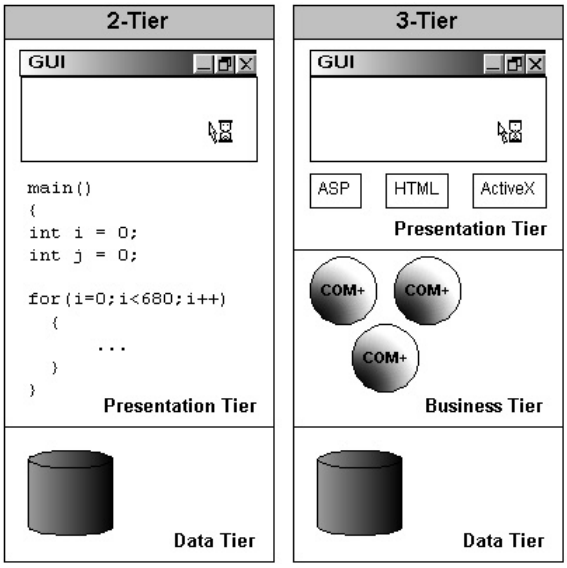

Figure 1 - 2-tier and 3-tier architectures

In 3-tier applications on the other hand, while the data services layer continues being a separate entity, presentation services and business services have also become separated. Indeed, the business tier now consists of several different components. This has several benefits and a few drawbacks. One of the main drawbacks is that a researcher wishing to implement an application using a 3-tier architecture has to acquire new skills that include several optimization techniques, which may seem strange to seasoned professionals. And the learning curve is steep. On the other hand, one gains several distinct advantages. The most obvious of these advantages is the modularity that one is forced to adopt. By breaking down an application into logical components, one adheres to the age-old and well-known method of picking a problem apart in order to solve it in a satisfactory manner. Add to that the fact that component-based applications have a much improved scalability factor than their client-server cousins, and it is clear why lots of software houses are slowly redesigning their applications from the ground up, in order to draw on the numerous advantages of component-based design.

3. COMPONENTS IN COMPUTER VISION

Computer Vision testing platforms are often complex. In order to experimentally arrive at a result, and in order to codify that result within the context of scientific theory, these platforms must be able to conduct tests that can be duplicated, re-run and examined in great depth. It is fruitless and frustrating, being forced to spend hours writing code in order to test one small part of a particular approach or method, then, when and if the results are satisfying, to scrap that piece of code and begin anew, trying to make sense of the next piece in the

puzzle. We believe that many researchers in the field are forced to redo work that they have done before, simply because they are limited by their choice of development methodology. We propose a paradigm shift towards component-based programming for the following key reasons: Using components is in keeping with traditional problem solving techniques. One breaks down the problem into multiple yet smaller problems, and then proceeds to best each one of those in turn. When writing a component, modularity is a prime consideration. This allows for the reuse of a component under different scenarios. Picture a component that performs Gaussian smoothing on input frames. Smoothing is a common requirement in many scenarios. In each and every one of those, the same component can be reused, without having to write a single additional line of code. In addition, since components are usually built according to some particular standard (such as COM), integrating another researcher's code into an application becomes a trivial task. Performance can often be an issue where computer vision is concerned. This is why many researchers opt for low-level languages, avoiding the use of high-level data constructs and the like. With the advent of component servers, however, 3-tier applications outperform monolithically programmed ones through extraordinary management of their resources. This allows programmers to utilize the full extent of a language's capabilities without hampering performance.

Applications that are programmed using a component-based approach are easily maintainable. The latest component servers (such as Microsoft's Transaction Server) provide fully graphical interfaces, through which one can easily manage an application's components, methods and interfaces. The 3-tier approach leads to extensible applications. With each component taking care of specific tasks, it is easy to check, remove or add functionality at will. This can be accomplished either by modifying an existing component or by adding a new one. In either case, the programmer will need merely concern himself with the particular piece of code to be modified. Since each component is an autonomous piece of software that performs a particular function on behalf of the application, there is no reason why one couldn't use more than one programming language throughout the development cycle. Programming language independence is a great asset, especially for teams of researchers with a mixed skill base. All the above benefits can be blended into a programming framework that will enable vision researchers to concentrate their efforts in producing solutions, not in overcoming hurdles imposed by design limitations and poor implementation decisions. An overview of such a framework follows in Fig. 2 below.

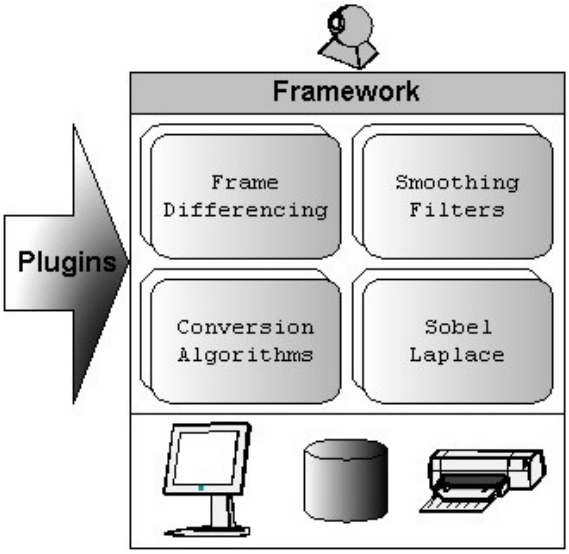

Figure 2 - A framework for Computer Vision programming

As shown above, input frames are captured by one or more cameras and fed into what is essentially the Business Tier. Several different components coexist within this framework, all hosted within a component server. Each component handles one or more specific tasks. In the above example, there exist components to perform frame differencing, Sobel or Laplace edge-detection, Gaussian smoothing and so on. These components could be considered to be 'plug-ins', as they can be added or removed at will. Output can be redirected to the screen at any point during the processing. This enables us to monitor our output at any time. For example, we may want to see the raw input frame, followed by the image of the frame after being subjected to a smoothing filter. Likewise, we can make use of relational or non-relational data-stores, saving raw and processed frames alike for later study. We can also use the programming framework to reverse the process. It is often the case that one may want to test an algorithm on a predetermined piece of video footage. In this case, we can use the data tier as an input as well as an output device.

The disadvantages to the approach described above are few. The most notable among them is that researchers not familiar with component-based programming will have to acquire new skills. While the learning curve is often steep due to the radical differences between component-oriented programming techniques and traditional programming techniques, the possible gains well justify the effort. Purists and advocates of the object-oriented programming mindset will have to learn several new things, but will find it easier to adapt to the necessary philosophy. A question that may be pertinent to ask is what are the tools that one should use to build such an environment. There is a

multitude of tools available. We have thoroughly experimented using Microsoft technologies. Our **Data Tier** consists of SQL Server 7.0 and data components that are hosted within Microsoft Transaction Server. Our **Business Tier** uses transactional business components hosted within Microsoft Transaction Server. MTS is the backbone of the Business Tier. It allows for the creation of 'applications' – packages of components, each with different properties. Since many of our 'applications' utilize the same or similar components, the ability of MTS to share components between applications is very useful. For our **Presentation Tier** we use two different clients. The first is a thick client that utilizes the Windows32 API, while the second is a thinner client that uses Internet Explorer to load an ActiveX control. Both clients include real-time observation of the video stream and offer full camera control through the VISCA protocol.

4. BUILDING THE FRAMEWORK

4.1 Purpose and Overview

The purpose of building a programming framework is to enable researchers to cut down on their development time by providing a centralized testing ground for their various algorithms and methodologies. The idea is to focus on what you are doing, instead of how to do it. The design of this framework, as presented in the previous sections, is generic enough to accommodate any type of computer vision system, whether it is a single-computer monocular vision system or a massively distributed system geared towards binocular vision. Reusability is the prime concern. It is what initially drove us to develop this programming framework. This concern manifests itself both through the ability of a researcher to reuse his own code and through his ability to seamlessly integrate the code of others into his application. The latter may indeed be more important, as it effectively nullifies the need to 'rediscover the wheel'. There is no point in writing code that has been written a hundred times before, such as simple edge-detection or thresholding algorithms. Researchers that adopt similar programming frameworks should have no trouble exchanging code, assuming certain precautions are observed during the coding process. Even the component model being used does not place an absolute limit on component compatibility. Several products, both commercial and not, exist only to 'bridge' different models, such as COM and CORBA. Our implementation of the framework is based on the Distributed Internet Application Architecture (DNA), evangelized by Microsoft. As such, our components have all been designed to meet the specifications of COM+.

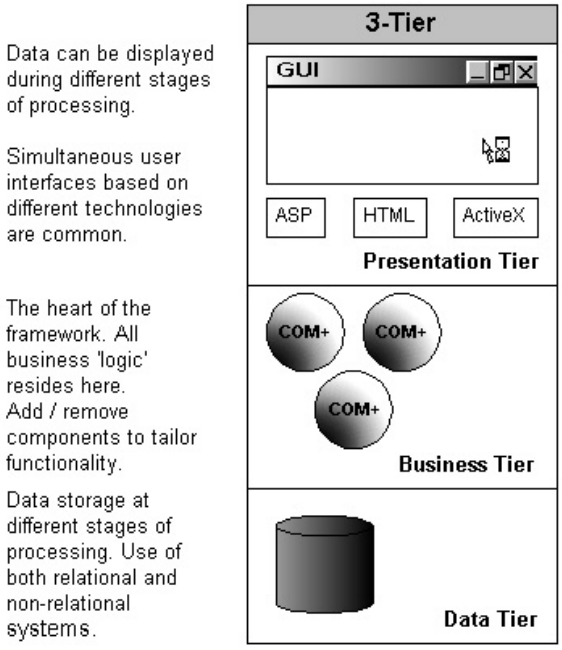

Figure 3 - A 3-Tier Programming Framework for Computer Vision

4.2 The Data Tier

The design of a 3-tier system traditionally begins with the design of the data tier. This is generally the case due to the fact that most three-tier applications being built are meant for commercial use. As such, they are mostly-data driven and hence the proper design of the data services layer is of prime importance. While data is probably of less consequence in our case, we will nevertheless also consider the data tier first, because of its relative simplicity. How much functionality we intend to implement for our data tier depends entirely on our needs. In our case, what we needed was a way to store all the data coming in and out of the system so that we could later review it at leisure. In order to accomplish that goal, we decided to use a relational database system. We allocated a large amount of space for the database and its log files and created a single table to hold all the data. Each row in this table consists of the following fields: *flag, path, image, application, date, and producer*. The flag field simply exists to indicate whether we chose to store the data in the image field provided, or whether we chose to store it directly on the hard disk. In this case, the image field is left blank and the path of the file is entered into the path field. The application field stores the name of the application to which the data belongs. The date field holds the date and time at which the entry was made. The producer field indexes the component that actually produced the data. We have programmed a component that simply takes data in a specified format from higher-level components. It then stores that data in the database. Options specified when calling the component's

methods are used to indicate whether the data should be stored in the database itself, or whether it should be stored on the disk with a pathname stored to point to its location. Our second data component is similar to the first, but works in exactly the opposite way. Its purpose is to retrieve data from the database and feed it to a higher-level component for processing.

Both components were pretty simple to code and it would truly be trivial to modify them so that they could work with any type of table underneath. To improve performance and to ensure the integrity of the data, both components are hosted in Microsoft Transaction Server (MTS).

4.3 The Business Tier

The business tier of our framework is entirely hosted within MTS as well. Although we only utilize the transaction options that MTS offers to a small degree, we have programmed all our components to support those options. Naturally, the component model that we are implementing is Microsoft's Component Object Model (COM), in its latest incarnation, COM+. All components that are part of this tier have the capability for both input and output of data. Also contrary to the tenets of object oriented analysis and design, none of our components have any properties whatsoever. They only implement methods. This is in keeping with several component statelessness guidelines that exist in order to facilitate easier distribution of components among several different physical servers. In addition to the above, each component in the business tier can call upon another, in order to utilize its functionality. Instead of hard-coding an object hierarchy, we opted to implement a 'root' component for each testing scenario. This 'root' component is charged with the task of calling the necessary components in the proper order. It is also responsible for communicating with the presentation tier. In most of our testing scenarios, we are also using the 'root' component to call the data components. Although all the other business components also have the capability to call the data components directly, we often find it more convenient to perform all data input and output in the root component. The MTS runs as part of the Windows 2000 Advanced Server operating system that hosts the business tier of the framework. Most of the components were written in Visual Basic 6.0 and Visual C++ 6.0, with a couple of them having been written in Java. For each scenario that we wish to explore, for each hypothesis that we need to test, all we have to do is create a new application from within MTS' graphical control tool. An 'application' in the MTS context is simply a package, a placeholder for components that may be hosted either within a single ActiveX DLL (Dynamic Linked Library), or within multiple DLLs. Since several such applications might need to use the same component, MTS also allows for the sharing of components between applications. In this way, we simply pick and choose which components we need, according to the functionality that our scenario requires. If the situation demands functionality that is not available through any of our existing components, we design and implement a new component. This procedure is fast and painless and allows for virtually infinite flexibility.

4.4 Programming considerations

On the down side, there are several considerations when writing code for components. After a little research, we decided to go with the consensus that claims that MTS components should be completely stateless. MTS provides the SPM (shared property manager), a mechanism through which one may implicitly force an application's components to hold state, but its use is not recommended. Another important point is that objects must always be destroyed immediately after they cease to be useful. In order to fully take advantage of a component server's exceptional resource handling, one must get used to the idea of acquiring resources late and releasing them early. All our procedures for object creation and destruction are governed by this maxim. In order to support transactions, components have to be properly set up. Unless specific requirements exist, the generally accepted practice is to set up those business components that may at some point initiate a transaction as 'requiring a new transaction'. Components that will never initiate a transaction, but which will be called by other components higher in the hierarchy will need to be flagged as able to 'use an existing transaction'. This will enable these components to work within the same transaction context as their callers. It is also important to never let a user directly manage resources in any way. This will dramatically decrease the performance of any 3-tier application and directly violates one of the primary design goals of the DNA architecture, Autonomy (the ability of an application to maintain total control over its critical resources – such as database connections). Instead, one should always force the user to go through the business objects in order to accomplish what he needs. If a component is likely to interact with another component often, it pays to implement them as classes in the same DLL. We did this for several of our components and watched the performance increase, as components made out-of-process calls less frequently. This gave our application a hefty performance boost and contributed towards proper marshalling of the data.

A corollary that we drew from our experience with the previous point was that it also pays to try to strike a balance between having complex components with really small hierarchies and having

simple components with very deep hierarchies. Based on small-scale evaluations and measurements, we believe that calling *eight* nested components should be the limit. If one finds the need to call more than eight components in succession, a re-evaluation of the application's design may be necessary. We took enough time designing this development environment so as to make sure that it would properly satisfy our needs. In particular, we paid great attention to the design of each component's interface – namely its methods and their arguments. One can easily change the code of a method in the future. If however, one is forced to add a new method, or modify an existing one (by adding a new parameter for example) after having deployed the application, binary compatibility with the old version of the component is broken and a lot of work has to be done in redeploying the application. We also avoided hard-coding several options into our components. We may be using 320x200 images, but other researchers that may wish to use these components might need to run them for 640x320 images. We tried to always implement the most generalized solution possible and let the component's methods accept arguments concerning the specifics of its operation. Where that was not possible, #define statements or Const declarations were used. Finally, we decided to document our system as best as possible. Well-placed comments within the code itself will allow those who may use our code in the future know what we did and why. In all cases, the minimum documentation that is provided is a purpose declaration for the entire component and a listing of its methods. Each method is also documented properly, its purpose stated, its methods and arguments shown. An example of use is often provided, and any assumption made (image size, color depth, specific data format), is also noted.

4.5 The Presentation Tier

We decided that we wanted to be able to control our system both locally and via our intranet. This led us to the design and implementation of two separate clients. The first client is a full-blown thick client that utilizes the Win32 API. The second client is a thin, browser-based client that uses Internet Explorer to load an ASP-based (Active Server Pages) interface. The presentation tier is probably the most customized part of the framework and it is likely that each researcher will model it according to his or her needs. With rapid application development tools such as Visual Basic being widely available nowadays, one can either opt to implement a minimum-functionality user interface (as we have done), or may choose to design and implement a fully-configurable and all-encompassing user interface that can adapt to any scenario with little trouble.

4.6 Physical deployment and the role of COM+

COM+ has been successfully described as the glue that binds the three tiers of a DNA application together. It is the model that the majority of component-based applications are based on. However, it is far more than that. COM+ is actually a set of services that programmers can use in order to code components that are dependable and portable. These services include support for transactions and queuing, as well as the often-misunderstood COM+ events. By harnessing the power of COM+, developers can create powerful infrastructures such as the one presented in this paper. The possibilities are truly limitless. The flexibility of COM+ also allows for developers to deploy their applications as they see fit. 3-tier architectures preach independence between the three tiers, but with COM+, it is even possible to host the components of the business tier within different servers. This way, components that are expected to be processor or memory intensive can be deployed on dedicated servers, while 'lighter' components can be grouped together. Since our resource requirements are still relatively small, we have chosen to deploy the entire framework on a single server.

5. EXAMPLE OF DESIGN AND IMPLEMENTATION

In order to provide the reader with an example of how a computer vision application may be designed and implemented using components, we present an outline of how a well-known face-recognition technique, described in [Turk91a], would be coded within the framework we propose.

The goal in the system presented in that paper is to detect the presence of faces in an image and then to classify those images by comparing what the authors have dubbed the images' *eigenvectors*. It is beyond the scope of this paper to go into detail about how the process of recognition works. Epigrammatically, however, we shall mention the steps involved in that process. During system initialization, a set of characteristic face images of known individuals is collected. The set should contain more than one image per person, with some variations in facial expression and lighting. The eigenfaces are calculated from the training set. All but M images that correspond to the highest eigenvalues are discarded (definition of the *face space*). Each individual's facial image is projected into the face space, and the corresponding distribution in weight space is calculated. One presented approach to face detection involves spatiotemporal analysis, thresholding, motion blob analysis, and rescaling. Finally, for the process of face recognition, an input

image is projected onto each of the eigenfaces and a set of weights is calculated. The system determines whether the image is actually a face. If it is a face, the image is classified as a known or unknown person.

The Data Tier is pretty simple to put together here. All that is needed is storage and retrieval of image frames. A data component with store and retrieve methods that communicates with our RDBMS suffices. Assuming coarse component granularity [Micro97a], the tasks listed above might be carried out using the following components in the Business Tier: An acquisition component (acquires a frame and stores it in the database), a frame-differencing component (performs spatiotemporal analysis), a thresholding component (produces a binary motion image), a rescaling component (estimates scale based on blob sizes and rescales), an initialization component (calculates eigenvectors and eigenfaces) and a recognition component (projects images into face space and performs analysis).

These components will obviously consist of several different methods. For example, the recognition component may contain one particular method for projecting an image into the face space, another to determine whether the image actually represents a face or not, and yet another to classify a facial image. Likewise, the initialization component may contain a method that calculates the average face Ψ of the training set, another to perform principal component analysis and so on.

Let us assume now that we wish our data to be presented to the user of the application both through a Windows client (Win32) as well as through the Internet. This can be achieved by designing and coding two different user interfaces. Fig. 4 below presents a logical diagram of the system, as it would have to be configured. However, what is worth noting here is that many of the components shown above are fully reusable. Let us take the acquisition component for example. This component, if written in a sufficiently generic style, could be used in any number of applications that require image capturing. Thus, if we had need of similar functionality in the past we will already have this component ready for use. It can even be simultaneously shared between two or more applications. Likewise, frame differencing, thresholding and rescaling are operations that many applications utilize. It is more than likely that if we have employed such components in a previous application, we will be able to use them for this one as well. Essentially, every piece of the application shown above could be drawn from a library of components, either custom-built, purchased, or downloaded. The only exceptions to that are the two components that are involved directly

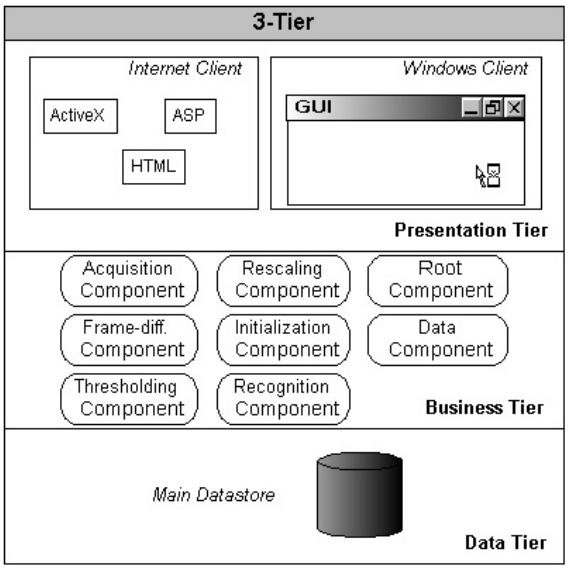

Figure 4 - Example overview

with the theory of eigenfaces – namely the initialization and recognition components. And that is precisely what we wish to show – that the process of applying a new theory is simplified and shortened, through reuse of existing application building blocks, which is a by-product of adopting a component-based methodology. The second important consideration is the requirement for two different user interfaces. Normally, this would require a substantial amount of additional development and programming. Using a three-tier approach however, the presentation tier, and hence the GUIs, are completely discrete from the functionality of the application and from its data. The latter two reside in the Business and Data tiers respectively. Thus, as long as certain rules are obeyed, both GUIs can be based on the same set of business components.

What was presented in this section was merely an example of how an application that performs specific tasks may be put together by developing new components and using already-existing building blocks. This component-based approach to software engineering furthers two main goals, as shown through this example: reusability and modularization. The benefits of these are argued throughout this paper.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a programming framework that advocates the use of components. It is loosely based on the 3-tier architectural model and its purpose is to aid in the creation and maintenance of stable, dependable testing and development environments. We have listed the possible advantages of this approach and have concluded that

although the learning curve for the programming skills required is steep, the benefits to be reaped are worth it. The main advantage of this framework is its flexibility and modularity. This is achieved through the extensive use of components, which empower researchers to rapidly create customized environments in which to experiment and test their theories and algorithms. We believe that there is a need for standardization in this area and we argue that this can be achieved by adopting widely used component models, such as COM and CORBA. We have shown how we have constructed such a framework using Windows-based technologies such as Windows 2000, the Component Object Model and the Microsoft Transaction Server and have outlined the choices we had to make and the reasons why we made them. We believe that the computer vision community stands to gain a lot by migrating to new technologies and by blending new programming techniques with existing ones. The programming aspect of the science has long been regarded as lesser in importance and has therefore remained dependent upon old practices.

This component-based approach to computer vision programming offers significant advantages. By eliminating the need to constantly rewrite code to handle routine application needs, it allows researchers to think more about the problem and its solution. It also fosters cooperation between researchers by promoting consistency in code. We have offered an example of how this sort of development environment could be utilized to design and implement a vision system. Our example was inspired by a well-known paper by Turk and Pentland [Turk91a]. Code will be available at http://www.hydilib.uoa.gr/vision.

The next steps involved in the process of coming up with a generalized computer vision testing and development environment should lead us in two directions. The first of those would be to continue to improve the platform and to make certain decisions that we have so far avoided, because of several thorny connotations. A primary, pre-determined image format should be one of those decisions. Particular sub-designs for alleviating some of the concerns and problems connected to binocular vision systems should also be one of our next steps. In addition, other protocols besides COM+ will be investigated. The objective of course, is to develop a platform that the majority of researchers can use and benefit from. A platform that is usable, infinitely extensible, easy to program and add to, but above all, one that can be used to span the entire spectrum of research needs – from displaying an image after the application of an edge-detection algorithm, to handling a distributed application that can provide functionality to other researchers across the Internet. It is also our intention to perfect our platform and transform it, so that our Lab can become what is known as an Application Service Provider (ASP), exposing the platform to the Internet. Under this software schema, other researchers will be able to use components that we have developed, or which we are hosting for others, under the guise of universally available *services*. These services will be available to Internet-aware applications around the world in a format that is easy to understand and use.

REFERENCES

[Bortn99a] Bortnicker M., Conard J: *Professional Visual Basic 6 MTS Programming*, Wrox Press, 1999

[Davie97a] Davies, E.R.: *Machine Vision: Theory, Algorithms, Practicalities,* Academic Press, 1997

[Eddon99a] Eddon G., Eddon H.: *Inside COM+ Base Services*, Microsoft Press, 1999

[Fauge99a] Faugeras O.: *Three Dimensional Computer Vision: A Geometric Viewpoint,* MIT Press, 1999

[Jain95a] Jain R., Kasturi R., Schunck B.G.: *Machine Vision*, Academic Press, 1995

[Kirtl98a] Kirtland M.: *Designing Component-Based Applications*, Microsoft Press, 1998

[Klett98a] Klette R., Schluens K., Koschan A.: *Computer Vision, Three Dimensional Data from Images,* Springer Verlag, 1998

[Malon99a] Maloney J.: *Distributed COM Application Development Using Visual Basic 6.0 and MTS,* Prentice Hall, 1999

[Micro97a] Microsoft Corp.: *Business Logic in Microsoft Transaction Server Components,* http://msdn.microsoft.com/library/backgrnd/html/msdn_buslog.htm, 1997

[Micro00a] Microsoft Corp.: *COM Specification,* http://msdn.microsoft.com/library/specs/S1CF83.HTM, 2000

[OMG99a] OMG: *CORBA/IIOP 2.3.1 Specification,* http://sisyphus.omg.org/technology/documents/formal/corba2chps.htm

[Parke97a] Parker J.R.: *Algorithms for Image Processing and Computer Vision,* Wiley & Sons, 1997

[Steve00a] Stevens R., Miller C.: *Wrapping and interoperating bioinformatics resources using CORBA,* Briefings in Bioinformatics, Vol. 1, No. 1, pp. 9-21, 2000

[Sundb00a] Sundblad S., Sundblad P.: *Designing for Scalability with Microsoft Windows DNA*, Microsoft Press, 2000

[Trucc98a] Trucco E., Verri A.: *Introductory Techniques for 3-D Computer Vision,* Prentice Hall, 1998

[Turk91a] Turk M., Pentland A.: *Eigenfaces for Recognition,* Journal of Cognitive Neuroscience, Vol. 3, No. 1, pp. 71-86