

# Building GUI Media Processing Applications: A Technology Options Analysis

Rajat Ahuja

Language and Media Processing (LAMP) Laboratory  
University of Maryland Institute for Advanced Computer Studies  
[rajat@ahuja.name](mailto:rajat@ahuja.name)

1.0	Introduction.....	1
2.0	Desirable Software Attributes.....	2
3.0	Rapid Prototyping Tools.....	2
4.0	Quick and Easy GUI Development.....	3
5.0	Integration with Existing Code.....	4
6.0	Performance Considerations.....	6
7.0	Other Considerations.....	6
8.0	Conclusions.....	7
9.0	References.....	7

## 1.0 Introduction

Media processing research and end-user applications, including those aimed at analysis of document images, are special in the sense that the input, and often the generated output, is visual media (image, video or document image). The ability to visually display the input and output from within the program is hence a very desirable and often indispensable feature. Additionally, such systems often need to provide the user or operator the ability to view intermediate results (images) and accordingly tweak parameters, or provide other forms of feedback. Although console or command-line based solutions *can* be used in conjunction with external media viewers, the best approach for such applications is often a Graphical User Interface (GUI) based solution.

Software libraries and APIs are often available for the most common multimedia-processing or computer vision techniques, in either source or binary form, but a graphical interface still needs to be built around them to provide the desired user experience. This paper explores some of the options available for developing such GUI applications from the ground-up or around application-specific libraries using a variety of different languages and platforms. Note, however, that these comprise a very small set of options which the author has chosen to explore in some detail and is by no means comprehensive.

The motivation for this study comes from a current project for text extraction from mixed text/graphics document images. DOCLIB, an object-oriented software library co-developed by the University of Maryland and Booz Allen Hamilton, written in C++ and designed from the ground-up specifically for document image processing, looks like a suitable library to use for this project. However, it is important to understand how easy it would be to integrate these with the additional code for the project and with the code for developing the GUI, especially if these components were to be developed in disparate languages or frameworks.

Although the motivation is to develop GUI solutions for document processing applications, the ideas presented and analyzed are pretty generic and can be applied to software solutions for other domains as well.

## 2.0 Desirable Software Attributes

Any GUI-based media-processing end-user solution is expected to satisfy, at least to certain acceptable extent, the following requirements:

- **Performance:** Most media processing tasks are computationally demanding on processing power and memory. Hence fast, optimized code is highly desirable.
- **Scalability:** Solutions need to be able to scale across increasing complexity and size of input media to be usable in the real world.
- **Platform Support:** If end-users/clients have a specific business or technological need to run the application on a specific platform (processor type, operating system), the technology used should either be platform-neutral, or the code should be designed, developed and tested on the target platform. Any third-party libraries used by the application should be available, easily deployable, or at least portable (with minimum effort) to the target platform(s).
- **Ease of Development:** Ease and simplicity of development, for both the algorithmic and the interface parts of the problem, is a desirable, if not essential, attribute to ensure timely development, testing and maintenance of code, and hence keep the project on track.
- **Packaging:** Developed solution should be able to be run independently instead of being forced to be launched from within another application. An end-user should be able to launch the application by simply “double-clicking” (in the Windows® world and similar environments) the binary or launcher script, or by executing a single, simple command line at the console.
- **Ease of Deployment:** Once developed, solutions need to be packaged and deployed on the client site. Irrespective of the deployment medium used (physical media, CD, over the Internet), it is desirable that the solution can be electronically “shrink-wrapped” and easily installed on the target machine(s), along with any supporting libraries or frameworks. Smaller installer sizes are preferable to larger ones, but of course not at the cost of performance.
- **Low TCO to the Developer and the Client:** Use of freely available, redistributable, and/or cost-effective components, libraries or frameworks to run the packaged application on or with is preferable to expensive or restrictive libraries. This helps lower the licensing costs involved with redistributing or imbedding third-party technology in the developed software solution, and hence the product costs transferred to consumers. If clients are required to obtain or deploy required frameworks or components on their own, use of low-cost components helps lower the client’s total cost of ownership.

## 3.0 Rapid Prototyping Tools

Engineering tools such as MATLAB® provide support for the most common mathematical operations using a dynamically-typed scripting language with inherent support for mathematical constructs such as matrices. Additionally, they provide toolboxes with common functions for specific applications (such as the Image Processing Toolbox), and often a developer-friendly GUI development environment (such as ‘GUIDE’). [1]

The ease and speed of development that these tools provide often comes at the price of poor performance when compared to native code generated with traditional programming languages such as C and C++, or even byte-code and virtual-machine based languages such as Java™ (on the Java VM) or C# (on the .NET CLR) [2]. Performance aside, lack of easy deployment options, inability to scale, and prohibitive licensing costs also render these quick-and-dirty “scripts” almost unusable as an end-user application.

Although these prototyping tools often provide the ability to generate native code in source or binary form, this is often not the most preferred option to build shrink-wrapped applications owing to the quality of generated code (performance, obfuscation etc.), and licensing complications and costs for embedding/redistributing libraries.

Hence, once the product is off the drawing board, and the proof-of-concept prototype is ready and tested, the next step is to transfer the technology to a more suitable development platform, as discussed in the sections that follow.

## 4.0 Quick and Easy GUI Development

A multitude of frameworks, libraries and programming languages exist today for GUI development, offering varying levels of ease and support. [3]

Frameworks such as the Java™ platform and the .NET framework offer perhaps the easiest option to develop and maintain graphical interfaces, because:

- GUI libraries are part of the standard distribution (Swing and AWT for Java; Windows Forms for .NET) and can be used by (one of) the supported programming language(s) (Java for the Java platform; C#, VB.NET, J# & C++/CLI for the .NET platform).
- Being highly object-oriented, all graphical components, including windows, panels, and buttons, are objects, and fit-in with other code in a very modular way.
- The graphical framework is designed to offer a certain level of abstraction, saving the developer from the intricate details. Anyone who has worked on, or even looked at code for, Windows GUI development with the Windows API using C, or even using MFC, would agree.
- Mature visual GUI editors exist for these platforms (NetBeans for Java; Visual Studio for .NET), providing the developer the option to implement the graphical layout visually and automatically generate corresponding code, which could then be edited, tweaked, or customized by hand.

Some of these advantages are partly language-dependent. For example, developers familiar with GUI development with Windows Forms using C# may not be as comfortable as doing the same using Managed C++ (Visual C++ .NET), even though they use the same libraries, owing to language constructs and semantics.

In the traditional C/C++ programming world, several mature and even cross-platform libraries exist for developing GUIs, for example, wxWidgets, GTK+ etc.

Other options include Tk, a widget toolkit that can produce rich, native applications and supports rapid development from a variety of languages, such as Tcl and Python (using the binding, “Tkinter”). Visual layout editors exist for Tk as well.

In general, high-level widget toolkits such as AWT, Swing, Windows Forms, GTK+, wxWidgets, Tk, or even MFC, offer a better development experience than lower-level ones, like the Windows API, or the Carbon API on MacOS X.

## 5.0 Integration with Existing Code

Quite like software applications in any problem domain, often there are software libraries for common media processing techniques and algorithms available in either source code or binary form. It thus makes sense to use, where possible, existing libraries while building a new media-processing application. However, for reasons of performance, ease of development, or specific client requirements, it may be necessary to use the pre-existing libraries in a different programming environment. For example, a computer vision library written in C++ may have to be used in a new application being written in Java.

If the development platform for a new application is pre-decided it might help to what options exist to integrate new code with the existing libraries. If the software architect is free to choose the platform, still, knowing what integration options exist and how well they rate in terms of ease and compatibility might help in making a wise design decision.

We now explore some common development platforms and how well they would integrate with pre-existing code.

The simplest scenario is the one where both the existing library and the new application are written in the same language or on the same platform. For example, a class library written in Java can easily be used by a new Java application, assuming that version compatibility and other such issues have been taken care of. It is fairly straightforward to use classes defined in the library and instantiate their objects in the new application, and then apply methods associated with these classes on these instances. In fact, it may even be possible to run a backend and a front-end on two different machines (or different VM's on the same physical computer), and have one instantiate objects on the other using Remote Method Invocation (RMI).

Similarly, a C++ program can use and instantiate objects of classes defined in a (compatible) library, and the same holds true for other OO languages such as Python (i.e., when both the library and the consuming application is in Python). An example would be writing a GUI application in C++, using wxWidgets for GUI and a compiled C++ .lib for the core image processing tasks.

In procedural languages such as C, there is no concept of objects, but structures and other data types can likewise be passed to functions defined in the library.

Further, since C and C++ run "natively", it is fairly easy (again, assuming version and other compatibility issues resolved) to call functions defined in a C library from C++. The other way around is a bit involved since C has not concept of classes, and so using a C++ library would mean that effectively, the API will have to expose methods which are merely functions and not part of any user-defined class.

Other natively running languages such as Python can still use code written in C or C++ by appropriately wrapping code where required. For example, the "Boost.Python" library provides a way to wrap a C++ class and expose it as a Python class so that code written in Python can instantiate objects on them.

The trickiest part comes when there is a need to mix native or “unmanaged” code with “managed” code, for example, when a Java program needs to call a function written in C, or when a C# program needs to call a function from a DLL written in C.

The Java platform offers the Java Native Interface (JNI) as a way to call native code from Java. This basically involves writing some C code which accept (as arguments) and return special JNI objects (technically, “structures”), does some processing, such as converting a Java string to a compatible character pointer for C, and then calls the actual C function provided by the library. Tools such as SWIG make this “code-gluing” easier by automating some glue-code generation; but clearly, some glue code needs to be written (or generated).

Calling C functions from C# is similar. In fact, if the C libraries have been exported using the appropriate identifier and compiled into a Windows DLL (Dynamic Link Library), no glue code may need to be written in C. It is possible to simply declare the DLL to be used, and call methods from this DLL and pass data to them. C# achieves this by means of a interoperability class defined in .NET, called “PInvoke”. It is common to use these for using Windows system level functions which may otherwise not be available via the .NET framework. Statically linked libraries (.lib) may not be used, but it may be possible to generate a DLL by modifying the original library code appropriately, provided it is available.

Note, however, that since native and non-native code is being mixed, the developer might be responsible to perform garbage collection at appropriate places, which would not be required in a pure “managed” environment (like Java or .NET).

Also, although calling C-style functions from DLL’s is fairly simple it is not possible to directly instantiate objects of C++ classes in C# code. The only way to get around it is to wrap C++ classes and expose them as COM (Microsoft’s Component Object Model) objects. However, developing with COM is far from trivial and hence, this may not be the easiest way out for someone not-too-experienced with COM.

It is, however, possible to use C++ objects and yet be able to exploit the ease of GUI development that Windows Forms, by using C++/CLI (previously known as Managed C++), which provides access to the .NET libraries and the ability to design the GUI in a visual editor (Visual Studio), while also providing the option to use classes defined in a C++ library. Again, garbage collection needs to carefully considered, and being essentially C++, it does not offer the same semantics as C#. Also, C++/CLI is a new language and offers a fair amount of new keywords, operators and language constructs, which a C++ developer would need to familiarize with.

For Java/C++ interoperability, NoodleSource makes using C++ classes in Java possible by generating glue code on both Java and C++ side, using JNI.

None of these interoperability techniques is perfect though, and often handling advanced language features such as templates or generics might be complicated or simply not supported.

Finally, it is always possible to make two modules, for example a front-end GUI image viewer, and a back-end program that does the actual image processing, to be developed as two separate programs that run in separate processes, and perhaps even on separate machines. These may then be made to exchange data, or call methods on each-other, using one of several options such as Inter-Process Communications (IPC), UNIX® sockets, TCP/IP sockets, or XML-RPC. These may operate at different layers of abstraction but ultimately allow exchanging of data between two programs.

## 6.0 Performance Considerations

As mentioned earlier, media-processing applications are usually computationally expensive and demanding, and hence performance and scalability are important criteria to make the developed solution usable in a practical scenario.

Although memory requirements and processing time (in short, the “computational complexity”) are highly dependent on a particular algorithm or process, often implementation specifics dictate how efficient the developed solution would be. An experienced developer may, for example, be able to utilize the resources more efficiently, by parallelizing tasks or de-allocating memory where possible. However, it is possible that the inherent structure or design of the programming platform may not allow such flexibility, thus minimizing the option for any optimization.

For example, consider an algorithm used to extract text from a document image, and let us assume that its current implementation in an interpreted language takes 1 second for the process to be applied on an image of size 256x256 pixels, and uses 128 MB of memory before the final output is generated and the memory could be freed. Also, let us assume that time taken and memory required per image grows as a cube of the image dimensions and the complexity of the features present in the image. It is likely that this implementation will not scale very well for an image of size 2048x2048 pixels, i.e., the process may either become too slow to be practically useful, or memory requirements may exceed the available resources.

In general, traditional programming languages such as C and C++ offer the developer the greatest amount of flexibility and power for designing and optimizing the application (e.g. by allowing direct memory manipulation using pointers), and hence often offer the best performance.

Byte-code-based languages such as Java provide better support for the OOP paradigm, free the programmer from the responsibility of managing memory by providing automatic garbage collection, and try to hide constructs such as pointers which may make the code leak memory if not used wisely. Hence, and for other reasons which are beyond the scope of this paper, these languages do not offer the same kind of performance as C and C++. Some languages, such as C#, offer the ability to use pointers in an “unsafe” mode, allowing the programmer to exploit direct memory management where absolutely necessary, but offering “managed” memory otherwise.

Some studies [2] have shown that with the increasing amount of processing power and memory available in modern computing environments, even managed languages such as Java and C# offer reasonable performance. Some of these claims have been personally verified by the author, albeit to a limited extent.

Other languages such as Python have not been analyzed or researched for the purposes of this article, but they can be expected to perform as well as Java, C# or C/C++. Rapid development tools such as MATLAB® and Mathematica®, however, are not designed with speed or scalability as the top priority, and hence cannot be expected to offer the same performance.

## 7.0 Other Considerations

Having considered the technical issues of performance, scalability and ease of development, we now look into the other desirable attributes and how well the technologies discussed above rate with regards to them.

Hardware/OS platform support is not so much of an issue anymore in choosing the right set of development tools, since almost all modern development tools (language, compilers, debuggers, libraries) support most common modern platforms. While some technologies such as Java™ allow a platform-agnostic development approach, others may require development on the specific target platform. In either case, compilers and libraries are often easily available for any target platform. Programming languages such as Python™ and libraries such as wxWidgets, GTK+ and Tk are available for Microsoft® Windows®, Apple®'s MacOS™ X, and most UNIX® flavors. Some platform-specific frameworks such as .NET have been (partially) ported to their equivalents on other platforms by independent projects such as Mono, providing developers more choice than ever. Glue-code technologies such as SWIG allow a plethora of languages to reuse code written in other languages making interoperability at least possible, if not simple.

Most of these technologies, including programming frameworks, libraries and development tools, are available for free (or at a very low cost) and with very liberal licenses, thus minimizing the costs transferred to the consumer and lowering the TCO of the software solution.

Using a good software and installer design, packaging and deployment of the solution developed with technologies discussed in sections 4-7 can be made as simple and elegant as desired.

## 8.0 Conclusions

- A multitude of options are available for developing GUI-based media processing software solutions, some of which have been analyzed and discussed.
- Rapid development tools are good for prototyping and research, but do not offer the best option for a shrink-wrapped software application.
- These options offer varying levels of ease development, performance and OOP support. Generally, high-level widget toolkits that ship with most modern OOP-based platforms offer the best developer experience.
- Using existing libraries and API's in another language or framework would certainly require adding some glue code, or at times, re-wrapping classes to a form usable by other modules.
- Considering performance and scalability attributes, whenever pre-existing libraries for the core media processing written in C/C++ are available, one solution might be to develop the front-end using Java™ Swing or Windows Forms, or another high-level widget toolkit, and integrate the two.
- In any case, any attempt to integrate existing code with new code in a different framework will not be trivial and will require design considerations, and careful thought and planning.
- Ultimately, the decision on using a particular development platform will have to take into account any specific platform requirements requested by the client or the end-user.

## 9.0 References

- [1] GUIDE: MATLAB's Graphical User Interface Development Environment ([http://www.mathworks.com/access/helpdesk/help/techdoc/creating\\_guis/gu\\_intr2.html#998352](http://www.mathworks.com/access/helpdesk/help/techdoc/creating_guis/gu_intr2.html#998352))
- [2] M. Jankowski and J-P. Kluska, "Connected components labeling - algorithms in Mathematica, Java, C++ and C#", 2004 International Mathematica Symposium (<http://library.wolfram.com/infocenter/Conferences/6040/>)
- [3] Widget Toolkits ([http://en.wikipedia.org/wiki/Widget\\_toolkit](http://en.wikipedia.org/wiki/Widget_toolkit))