Plug-ins: A Software Model for Biomedical Imaging & Visualization Research

R. Mullick, Ph.D.,
Diagnostic Radiology Department, Clinical Center, NIH
S. V. Warusavithana, B.Sc., V. Shalini, B.Sc., and P. Pang, M.S.
Centre for information-enhanced Medicine (CieMed), National University of Singapore

ABSTRACT:

A user-oriented application architecture using plug-in technology is proposed as an optimal solution for biomedical imaging and visualization research. This plug-in approach offers a two-tier solution for application development, separating core components of data manipulation and visualization from customized user interaction required for specific research. This methodology encourages code reusability and significantly reduces development time.

INTRODUCTION:

Imaging and visualization are now synonymous with research in biology and medicine. Researchers in all fields are exploring, analyzing and quantifying their data using advanced techniques in scientific visualization and image analysis. In order to gain insight into their data, researchers employ numerous computer applications often with great difficulty due to varying levels of incompatibility. On the other hand, as the needs of the researchers get more complex, application developers commit increasing amount of resources to accommodate many of these features. This leads to very complex user-interfaces and sets many constraints on the end-user. These factors in addition to development time commonly lead to the demise of such systems.

In order to alleviate these limitations and offer easy expendability and reuse of existing software resources, a plug-in architecture is presented. The plug-in option has been incorporated into popular photo editing (Adobe Photoshop™), web browsing (Netscape Navigator™, Microsoft Explorer™), and animation (Alias/Wavefront Maya™, Softimage™) packages and seems most suitable for use in medical image analysis. The concept of plug-ins is to offer supplementary applets, which are driven by a main core application. They are invoked on-demand for specific sub-tasks and are similar in concept to protocols in medical imaging lingo.

METHODS:

The plug-in paradigm being presented here differs from the past methods. Traditional plug-in implementations use a serial black-box approach, data is sent to the plug-in, processed by it and returned to replace the current data. This model entails data duplication which is memory intensive especially for multi-dimensional data and limits their possible use on personal workstations. Furthermore, many plug-in approaches develop their own interface, which does not seamlessly integrate with the core application.

The prototype presented here addresses these issues and creates a plug-in application-programming interface (API) for a general-purpose imaging and visualization application. It allows for third parties and independent researchers to expand the functionality of a plug-in enabled core application without increasing the size and complexity of it. It also gives the flexibility to rapidly offer custom applications to the end-user in his own familiar work environment. The plug-in architecture has been designed in such a way that the plug-in has direct access to the data within the core application and also exposes to the plug-in the window handles of the main application. This is organized via a collection of classes, which separates the dynamic and static components of the parent (Core) and child (Plug-in) application using a plug-in share object. The current implementation of the plug-in architecture uses the Windows DLL model.

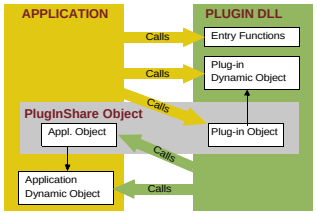

Fig 1: Plug-in Architecture & Communication

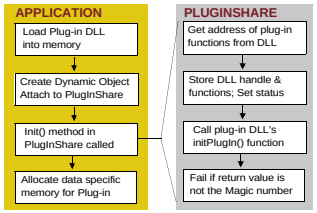

Fig 2: Plug-in Initiation

ARCHITECTURE:

The Interface

We have attempted to keep the plug-in architecture as simple and flexible as possible. The An overview of this architecture is shown in *Figure 1*. The model consists of five objects described below:

PluginShare Object: This is basically a container object for the application and plug-in object.

Application Object: This object contains information about the application side of the plug-in API interface. It has a few methods (set/get application version, set/get API version) which are considered essential and should be present in every version of the plug-in API. The application object also contains a pointer to the application dynamic object.

Plug-in Object: This object contains information about the plug-in side of the plug-in API interface. It has a few methods (set/get plug-in version, set/get API version, active status, init/shutdown) which are considered essential and should also be present in every version of the plug-in API. It also contains a pointer to the plug-in dynamic object.

Application Dynamic Object: This contains the API methods which the plug-in will call to request data and services from the application.

Plug-in Dynamic Object: This contains the API methods which the application will call to request services from the plug-in.

Object Creation

So which process creates which objects? The PluginShare object is the interface between the application and the plug-in. At application start-up (See *Figure 2*), the application creates the PluginShare object, the application object and the plug-in object. When a plug-in DLL is invoked, the application creates and initializes the application dynamic object, initializes the application object, and passes the address of the PluginShare object to the plug-in DLL. The plug-in DLL then creates the plug-in dynamic object and attaches it to the plug-in object.

Entry/Exit Functions

These are a set of functions to be exported by the DLL. These functions are used as an entry/exit point into the plug-in and also to obtain information about the plug-in so that the plug-in descriptions can be queried and displayed in a menu in the application rather than displaying plug-in DLL names in a menu.

Communication

The idea is that the application will call methods in the plug-in and plug-in dynamic objects in order to obtain information from the plug-in and request the plug-in to perform services for it. And similarly, the plug-in DLL will call methods in the application and the application dynamic objects to obtain information and perform services for it. In addition, the application will call certain entry functions within the DLL to invoke the plug-in DLL and identify the DLL to be an approved application DLL.

The application provides the following data and services to the plug-in:

- Direct access (pointer to the data) to the currently loaded data;
- Allows the plug-in to query and set parameters such as lighting, volume of interest (VOI), rotation etc.

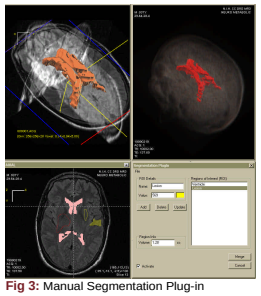

Fig 3: Manual Segmentation Plug-in

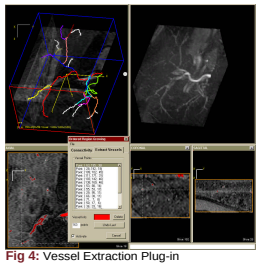

Fig 4: Vessel Extraction Plug-in

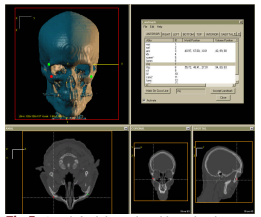

Fig 5: Craniofacial Landmarking Plug-in

- Passes all window messages received by specific windows to the plug-in. The plug-in gets these window messages before the application acts on them. If the plug-in consumes these messages the application ignores them. The plug-in can also alter the message such that the application will act on the altered message rather than the original message.
- Allows the plug-in to draw onto specific windows. The drawing sequence for each window is as follows:
  - Application OpenGL drawing
  - Plug-in OpenGL drawing
  - Application overlay drawing
  - Plug-in overlay drawing
- Provides a mechanism for the plug-in to request the application to allocate a fixed quantum of memory for each dataset loaded into memory. This frees the plug-in from having to deal with complex issues of multiple volume management.

RESULTS:

Operational Plug-in prototypes have been developed for (i) general purpose 3D manual segmentation, with application to visualization and quantification of specific regions of interest (*Figure 3*); (ii) vascular structure extraction from MR Angiographic data (*Figure 4*) using ordered region growing [1] with seed selection from Maximum Intensity Projection (MIP) view (iii) 2D/3D landmark placement (*Figure 5*) on craniofacial datasets [2].

The core application and the plug-ins were developed using Visual C++, OpenGL, and Win32 SDK and are functional on any Windows (NT, 9x) platform. Other plug-ins for image/volume registration, multi-modal (MR/CT/PET) rendering, functional MR analysis, brain MR segmentation, and virtual endoscopy navigation are being designed.

REFERENCES:

[1] Yim PJ, Summers RM, Mullick R, and Choyke PL, "Detection of the Small Vessels in Magnetic Resonance Angiograms by Grey-Scale Skeletonization," 1999 Biomedical Imaging Symposium, National Institutes of Health.

[2] Valeri CJ, Cole TM 3rd, Lele S, Richtsmeier JT, "Capturing data from three-dimensional surfaces using fuzzy landmarks," Am J Phys Anthropol 1998 Sep;107(1):113-24.

MORE INFORMATION:

`http://developer.netscape.com:80/support/faqs/plugins/general.html`
`http://developer.netscape.com:80/docs/manuals/communicator/plugin/basic.htm`
`http://partners.adobe.com/asn/developer/gapsdk/PhotoshopSDK.html`

CONTACT:

EMail: rmullick@nih.gov

Address: Clinical Center, NIH
Bldg. 10, Room 1C660
10 Center Dr., MSC 1182
Bethesda, MD 20892

Phone: +1-301-496-7700