doi:10.1017/S143192761700191X
Microsc. Microanal. 23 (Suppl 1), 2017

The Dragonfly Macro Engine for Executing Recorded Tasks in Image Processing and Visualization

Mathieu Gendron[1], Nicolas Piche[1], Mike Marsh[1]
1. Object Research Systems. Montreal, Canada.

Automated data collection can outstrip data processing. A proven solution for dealing with routine processing is software macros that capture and reproduce sequences of user actions [1]. Since high-throughput imaging technologies are now affecting both 2D and 3D workflows, we have developed an image processing and visualization platform, called Dragonfly, that addresses data processing challenges with a user-friendly macro engine for recording, encapsulating, and executing processing pipelines in a scalable and highly flexible way. We report here on the unique architecture of the Dragonfly platform, the design of the macro engine, and macro-driven application examples.

Dragonfly is the result of a software engineering effort whereby our team refactored an existing general-purpose application, called Visual SI, with the expressed goal of tightly integrating Python for the benefit of both developers and end-users. Python is an interpreted, high-level programming language that has gained widespread adoption in scientific programming [2]. The Dragonfly architecture reproduces a common software engineering pattern where compute-intensive code is decoupled from program logic. Taking advantage of this means that image processing and visualization is implemented in compiled code (C++) and system-level libraries (e.g. OpenGL), while the performance-insensitive application logic is coded in Python. Like other coupled-approach tools that came before [3], Dragonfly enjoys fast performance of performance-critical routines but also the rapid application development and short debug cycles of interpreted languages. Because Dragonfly's API is open, end-users can also take advantage of Python for rapid development of plugins to extend Dragonfly functionality. Furthermore, the tight Python integration delivers users an embedded Python console directly in the application. This empowers users to prototype ideas directly in the rich graphical application in a way similar to how MATLAB (The Mathworks; Natick, Massachusetts, USA) developers test solutions on that platform.

This Python integration is integral to Dragonfly's macro engine. Dragonfly generally lets users execute arbitrary Python code to accomplish tasks, and the macro engine is a special case of executing Python code. The macro engine lets users automatically capture sequences of user interaction directly into a user macro, which is a special case Python script. Use of the automatic capture system is not strictly necessary as advanced users could bypass the recorder and choose to author a macro from scratch like any general purpose Python script. The macro player lets users play back those user macros with advanced playback features. Again, advanced users could bypass the player and execute the code directly from the console or from other contexts.

A critical ingredient of Dragonfly's design is the interface pattern that specifies that atomic user actions be encapsulated as single methods. In practice, this means that typical user activity (e.g. applying an image filter, adjusting a viewer camera angle, thresholding an image, etc) are defined by methods that meet a particular interface specification. The interface defines the method's input and output datatypes using the standard Python docstring convention, and any Python method can be designated as a interface

method by standard use of Python's decorator pattern. Dragonfly's architecture ensures that no matter how these user methods are actuated by the user at runtime (e.g. by mouse clicks in the GUI, invoked directly from the console, called from a script, etc.), they can be captured and logged. At any time, a Dragonfly user can review a log file that shows the history of interface methods for that session. Consequently, this design makes it possible for the macro recorder to capture an arbitrary sequence of actions and encapsulate them in a user macro.

The macro recorder interface gives the user the opportunity to start, stop, reset, and pause the recording of actions. It also lets the user name the macro, designate it for shared use, and view the Python source code in the user's preferred Python editor. The user macro, as captured by the macro recorder, is pure Python, complete with automatic documentation.

Saved user macros can be executed with the macro player. In addition to simply executing a sequence of actions, the macro player interface lets users designate steps that can be bypassed, or steps that should be paused for user inspection (much like breakpoints in modern debuggers). This functionality is useful when macro execution needs human supervision. All of these details get encoded in the user macro's comments, which means they will be ignored if the user wants to execute the macro in some context other than the macro player. The real strength of the macro player is in its dependency analysis. The macro player can introspectively interrogate the user macro to identify that results produced in early steps of the macro must be used as arguments for later steps in the macro. This dependency linkage is handled automatically for the user. Furthermore, if a user macro's early step requires a parameter that is not defined, the macro player automatically determines that parameter's data type and the interface prompts the user to find a suitable parameter. This ability to resolve unsatisfied dependencies makes the macro playback very robust across a variety of use cases.

The types of user macros that can be captured and played back are limitless. Users can design macros to capture screenshots, produce videos, apply image processing sequences, perform quantitative analysis, etc. For example, consider a popular bone segmentation routine that requires users to perform a 13-step sequence of thresholding, image filtering, morphology operations, masking, etc, in order to segment cortical bone [4]. With Dragonfly's macro engine, that entire routine is encapsulated in a user macro that can run without interruption or, optionally, with pauses so the user can review steps before progressing.

The macro engine described here is made possible by significant architecture decisions made about Python integration into Dragonfly, and it hinges on the interplay of thee components: the use of the interface pattern by developers, the macro recorder to capture sequences of actions, and the macro player to execute the saved routines. The interface pattern ensures that this solution is future-proof because new functionality from compliant developers automatically works with the macro engine.

References:

[1] CA Schneider *et al, Nature Methods* **9** (2012), p. 671.
[2] G van Rossum: Python Reference Manual. May 1995. CWI Report CS-R9525.
[3] G Tang et al. *Journal of Structural Biology* **157** (2006), p. 38.
[4] H Buie *et al. Bone* **41** (2007), p. 505.

https://doi.org/10.1017/S143192761700191X Published online by Cambridge University Press