Visual Programming of VTK Pipelines in Simulink

Release 1.0

D. G. Gobbi[1], P. Mousavi[2], A. Campigotto[2], A. W. L. Dickinson[2] and P. Abolmaesumi[2,3]

October 31, 2010

[1]Atamai Inc., Calgary, Alberta, Canada
[2]School of Computing, Queen's University, Kingston, Ontario, Canada
[3]Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, Canada

Abstract

We have created a wrapper package named SimVTK that allows VTK, and third-party classes derived from VTK classes, to be seamlessly integrated with MATLAB®'s Simulink® interface. Our package generates a loadable Simulink module for each VTK class, which is then represented as a "block" on the Simulink canvas, and can be connected with other blocks to form a pipeline. Each block can be double-clicked to bring up a dialog box that allows introspection and control of the VTK class. After a VTK pipeline has been built, it can be run interactively from within Simulink. The outputs of the pipeline can be displayed in an interactive render window, written to a disk file, or exported to a MATLAB variable. Within Simulink, the VTK pipeline can also be connected to the ITK pipeline through the use of our SimITK package. We generate the VTK-Simulink wrapper automatically though the use of CMake build scripts, with XML as an intermediate representation of the VTK class interfaces.

Contents

1 Introduction

Although VTK is a collection of C++ libraries, its developers chose to provide VTK with bindings for higher-level computer languages, in order make application development with VTK more rapid and interactive [7]. These "wrappers", as the bindings are usually called, are generated automatically from the VTK class header files and allow VTK to be used from Tcl, Python, and Java. Our goal was to use the wrapping paradigm to bring VTK comprehensively into a visual programming environment, MATLAB®'s Simulink® platform. We make each VTK class available in Simulink along with its documentation and a dialog box to adjust its parameters. Unlike the Tcl, Python, and Java wrappers, our wrapper does not sit within the VTK source tree, and does not build as part of the VTK build. Instead, it builds on top of an existing, unmodified VTK build.

The advantages of a visual programming environment like Simulink over a general-purpose programming language like Python are: 1) rapid development and instrospection of applications through a simple drag-and-drop, point-and-click interface, 2) a graphical representation of the pipeline for use in documentation and other forms of communication between developers, and 3) a gentle learning curve for people with little or no programming experience.

Other visual programming environments for VTK take less direct and/or less comprehensive approaches to incorporating the VTK classes. VisTrails [1] and DeVide [3] use the Python wrapper, and utilize Python's self-introspection abilities to provide a list of classes and properties to the user. MeVisLAB [2] and XiP [8] each have their own semi-automatic method for making VTK objects available through OpenInventor. SCIRun [6] uses a manual approach that makes certain VTK classes available through its visual interface. In contrast, the wrapper that we have developed, hereafter referred to as SimVTK (http://media.cs.queensu.ca/SimITKVTK), is generated by a fully automatic process, similar the Tcl, Python, and Java wrappers. This means that SimVTK can be built for any recent (VTK 5.2 or later) source or binary distribution package for VTK. SimVTK can automatically introspect a VTK package and generate a Simulink wrapper for it, without the need for patching the VTK code.

2 Design Requirements

Simulink is the visual programming environment available in MATLAB® (MathWorks, Natick, MA, USA). We chose to use Simulink because of its maturity and its integration with MATLAB, providing a broad range of computational functionality that complements VTK. In addition, Simulink allows custom programmatic "blocks" called S-Functions to be written in either MATLAB code or C/C++, the two most common languages in the field of image analysis. The design requirements for SimVTK are as follows:

1. It must be possible to mix VTK, ITK, and native Simulink blocks in a pipeline.
2. It must be possible to introspect and set parameters of VTK objects from Simulink.
3. The VTK blocks for Simulink must be automatically generated, not written by hand.
4. The full package must be open source, apart from MATLAB and Simulink themselves.
5. The package must be multi-platform, with Linux, Windows, and Mac OS X as the targets.
6. The package must work on 32-bit and 64-bit platforms.

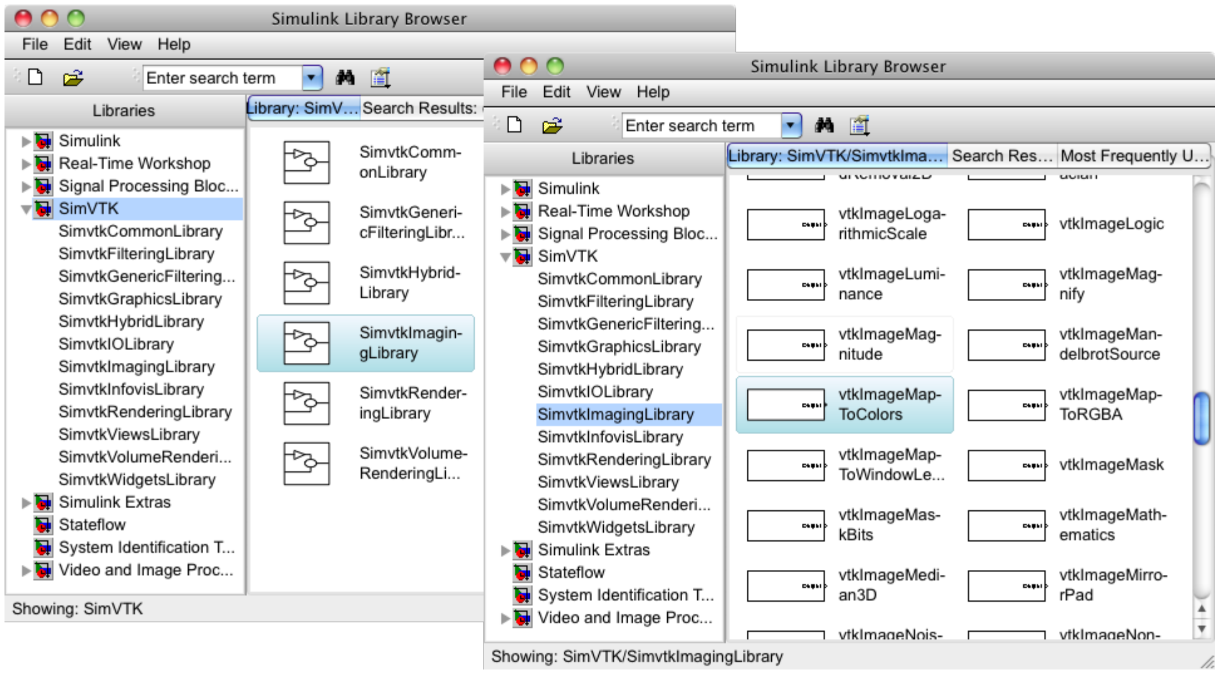


Figure 1: SimVTK library browser. Clicking on one of the libraries on the left will bring up a class browser like the one on the right.

We provide a collection of block libraries (Figure 1), where each block library consists of a text-based .mdl file that describes the block, plus a collection of loadable modules (mex files) for Simulink. To create a VTK pipeline, the user can select the SimVTK libraries in Simulink, and drag-and-drop the filter blocks in order to create a data processing pipeline (e.g. Figure 2). When a block is double-clicked with the mouse, a dialog box appears that allows the user to set all the object parameters (Figure 3). The SimVTK block descriptions and the code for the mex files are generated from XML descriptions of the inputs, outputs, and parameters of each VTK class, and the XML itself is generated by applying a custom text-processing tool to the header files of the VTK classes. The entire process is driven by a CMake [5] build script and a few small CMake macro scripts.

3 System Architecture

Simulink works by passing "signals" between the blocks in the pipeline, where each signal is a MATLAB array. Since a VTK object cannot be represented as a MATLAB array, we declare a new MATLAB data type called "vtkobject" that is an integer containing the memory address of a VTK object. From MATLAB this is an opaque data type that cannot be accidentally misinterpreted as a regular integer. To transfer data between VTK and MATLAB, we have created import and export blocks that take a vtkDataObject as input and produce MATLAB arrays as output. Currently, we only provide import/export blocks for vtkImageData, but we intend to provide blocks for vtkPolyData and vtkUnstructuredGrid in the future.

When Simulink runs, our "vtkobject" signals are passed between the blocks and used to connect the filters in the VTK pipeline. After this initial connection phase is completed, Simulink updates the pipeline at regular intervals until it is told to stop, either by the user or by an internal termination signal. For each block in the

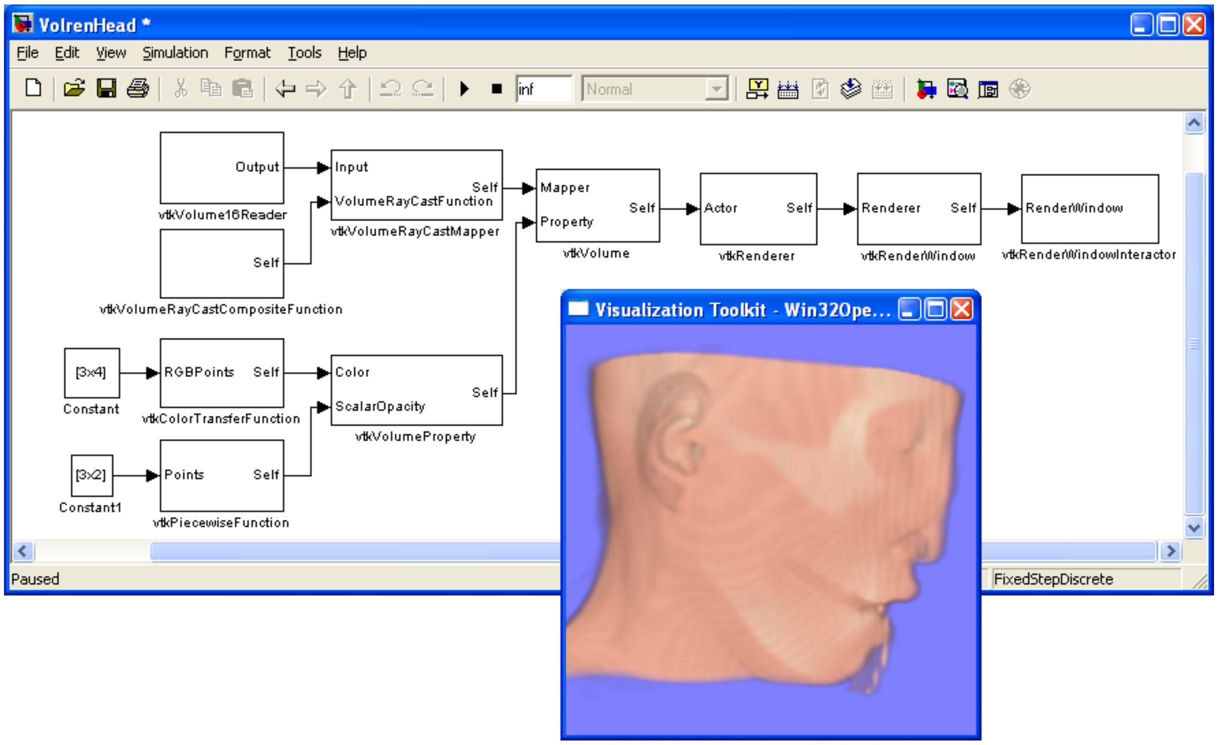


Figure 2: A SimVTK model for volume rendering. The color transfer function and opacity function for the volume are stored in MATLAB arrays.

pipeline, there is a dialog box, accessed by double-clicking the block, that allows instrospection and control of the VTK class. We have developed a set of rules that are used by our wrapper to translate the VTK class interface, as defined by the methods in the class header file, into widgets for the dialog box:

“Get” methods returning scalars, arrays, or objects. The block’s dialog box has a checkbox which, if selected, adds an output to the block for this method.

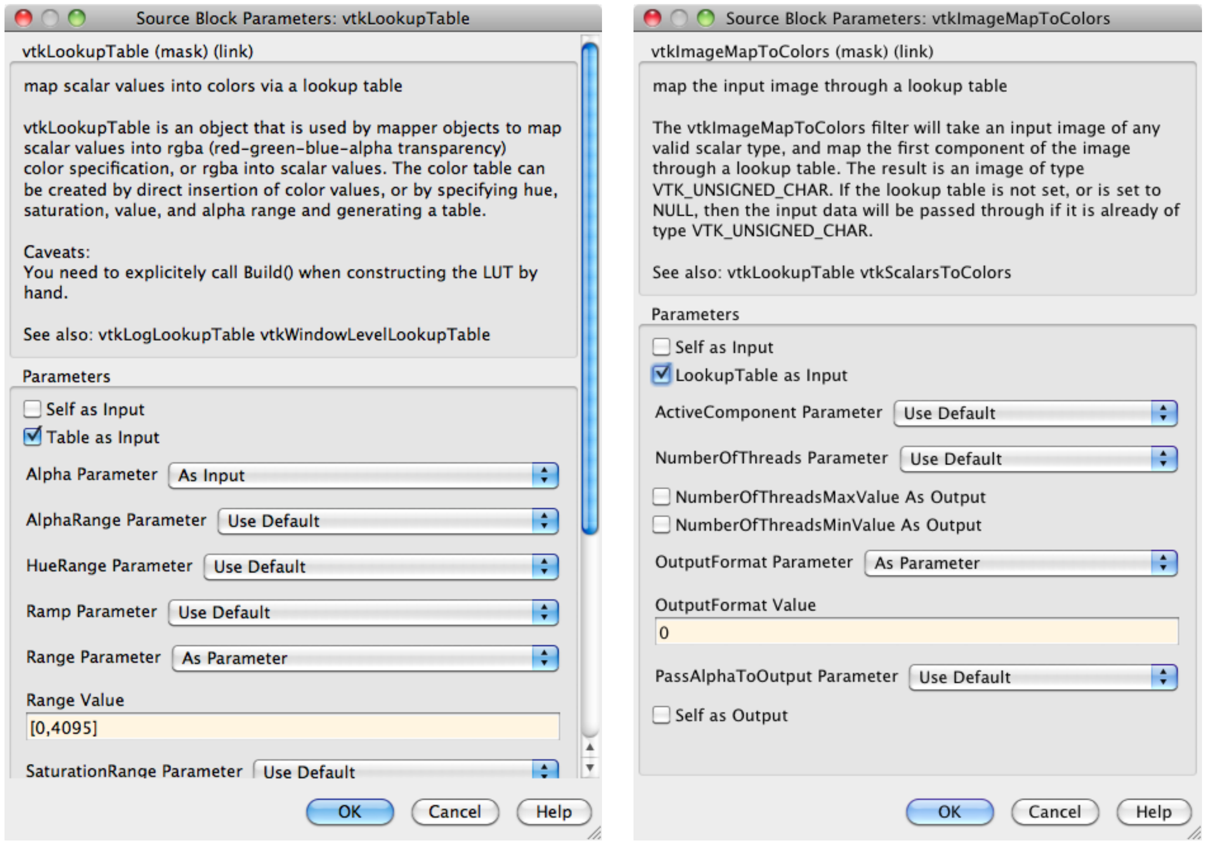
“Set” methods with a scalar or array parameter, like SetPoint(double p[3]). The dialog box allows the user to select from three options for these methods: *‘Ignore’* - use the default parameter value, *‘As Parameter’* - use an entry box to set the parameter, *‘As Input’* - add an input to the block for this parameter, so that it can be connected to a “Get” method from another block.

“Set” methods with an object parameter, like SetMapper(vtkMapper *). The dialog box has a checkbox which, if selected, adds an input to the block for this object.

“Add” methods with an object parameter, like AddActor(vtkActor *). The dialog has an entry box to select how many inputs the block should have corresponding to this method.

Outputs for algorithms, like GetOutputPort(). All algorithm outputs are always included as block outputs.

Inputs for algorithms, like Set/AddInputConnection(). For mandatory algorithm inputs, there is always a block input. For optional inputs, there is a checkbox, and for repeatable inputs, an entry box to select the number of inputs to add to the block.



Source Block Parameters: vtkLookupTable

vtkLookupTable (mask) (link)

map scalar values into colors via a lookup table

vtkLookupTable is an object that is used by mapper objects to map scalar values into rgba (red-green-blue-alpha transparency) color specification, or rgba into scalar values. The color table can be created by direct insertion of color values, or by specifying hue, saturation, value, and alpha range and generating a table.

Caveats:
You need to explicitely call Build() when constructing the LUT by hand.

See also: vtkLogLookupTable vtkWindowLevelLookupTable

Parameters

☐ Self as Input
☑ Table as Input

Alpha Parameter	As Input
AlphaRange Parameter	Use Default
HueRange Parameter	Use Default
Ramp Parameter	Use Default
Range Parameter	As Parameter
Range Value	[0,4095]
SaturationRange Parameter	Use Default

Source Block Parameters: vtkImageMapToColors

vtkImageMapToColors (mask) (link)

map the input image through a lookup table

The vtkImageMapToColors filter will take an input image of any valid scalar type, and map the first component of the image through a lookup table. The result is an image of type VTK_UNSIGNED_CHAR. If the lookup table is not set, or is set to NULL, then the input data will be passed through if it is already of type VTK_UNSIGNED_CHAR.

See also: vtkLookupTable vtkScalarsToColors

Parameters

☐ Self as Input
☑ LookupTable as Input

ActiveComponent Parameter	Use Default
NumberOfThreads Parameter	Use Default

☐ NumberOfThreadsMaxValue As Output
☐ NumberOfThreadsMinValue As Output

OutputFormat Parameter	As Parameter
OutputFormat Value	0
PassAlphaToOutput Parameter	Use Default

☐ Self as Output

Figure 3: VTK parameter dialog box. The checkboxes and selection boxes indicate parameters that are to appear as inputs or outputs of the block.

"Self" as an output. All blocks can have an optional "Self" pointer as an output. This allows any block to be used as an input to another block that accepts that block type as an input.

"Self" as an input. All blocks can take an optional "Self" pointer as an input. This allows us to take a vtk object as an output from one block, and then associate a block with that object. For example, if a vtkActor has a Property output, that output can be fed into the Self input of a vtkProperty block, allowing us to introspect that Property.

While Simulink is running the pipeline, all RenderWindows and Interactors are updated at each time step, so user interactions with the RenderWindow work as usual. Furthermore, if input values for any VTK "Set" method changes, then the result of the change is displayed at the next time step.

4 Code Generation

The wrapping process for SimVTK uses XML as an intermediate step, i.e. an XML description of all of the VTK classes is generated by parsing the VTK header files; the XML is then read by tools that generate the wrapper source code. Separating the header parsing and the code generation into two steps was invaluable to our development process, since it provided a clear and human-readable description of all the data that was supplied to the code-generation tools. The overall code generation process occurs in the following steps:

1. CMake scripts read the VTK “export” scripts, which list all the VTK classes;
2. our vtkXML tool generates an XML description of each class;
3. perl scripts read the XML and generate the C++ code and .m files;
4. perl scripts read the XML and generate Simulink .mdl library files;
5. the code is compiled into loadable Simulink modules (mex files).

The entire process, from start to finish, is driven by CMake scripts and can be done on any of the three supported platforms. The scripts define CMake macros that take the following actions:

SimvtkGenerateXMLFile generate XML for a class

SimvtkGenerateSFunctionFile generate the C++ code for Simulink blocks

SimvtkGenerateMEXFile compile the C++ code into loadable modules

SimvtkGenerateLibraryFile generate a Simulink library file

Since SimVTK builds on top of VTK and MATLAB, it is necessary to set the CMake variables VTK_DIR and the MATLAB_ROOT when CMake is first run. It is also necessary to use an installation of VTK that was built with shared libraries and with a compiler that is supported by the installed version of MATLAB. We have done builds with gcc 3, gcc 4, MSVC 8.0, and MSVC 9.

Step 2, which creates XML descriptions of all the VTK classes, is done by a C executable called vtkXML that we created by modifying the vtkWrapPython executable that VTK uses to generate its Python wrapper. The vtkXML executable is called on each of the VTK header files by our CMake scripts

The next two steps, which generate the code and libraries, are performed by perl scripts. Perl has excellent text processing facilities, and we found it to be an ideal tool for reading the XML and using it to fill in the blanks in our “boilerplate” Simulink/C++ source files, in order to create new Simulink/C++ files that provide Simulink bindings for each VTK class.

The final step, compiling the code with either gcc or Microsoft Visual C++, is also driven by the CMake scripts. The result of this step is a collection of platform-specific binary “mex” files (either .mexmaci, .mexw32, .mexglx, etcetera) that can be loaded by Simulink. To improve the experience for the user, we also generate a “browser” library block that contains a list of all the VTK kits, as shown in Figure 1.

5 Results

We built several VTK pipelines as Simulink models, two of which are shown in Figures 2 and 4. When the “play” button on the model is pressed, the render window appears with the initial model view, and the view updates when the user clicks or drags the mouse in the window, or when a pipeline parameter is modified. SimVTK provides this interactivity by polling the pipeline at short, regular intervals to determine whether the pipeline should be re-executed. Since the VTK pipeline natively supports lazy execution, we achieve this polling by regularly calling the Update() method on all terminal filters in the pipeline, and the Render() method on all terminal render windows and interactors.

Figure 2 shows a model for volume rendering, where every block has been named after a VTK class except for the two “Constant” blocks on the far left, which are MATLAB arrays containing tables for the color

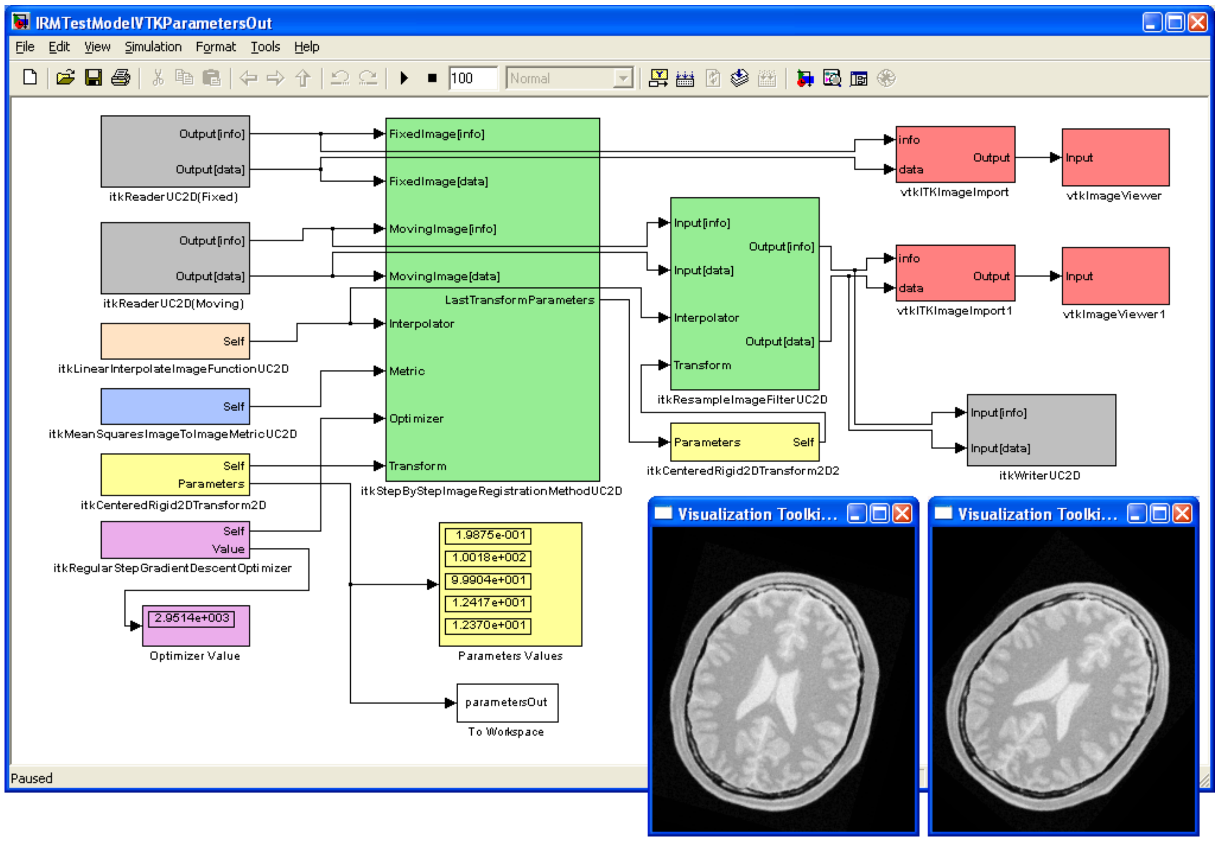


Figure 4: A Simulink model that combines SimVTK and SimITK. This model does a registration with ITK and displays the results in real-time using VTK as the registration process iterates.

transfer function and the opacity function used to color the volume. Note that the block name is the text shown below each block, while any text inside the block is the name of a block input or output. The flow of data in the pipeline is from left to right: the vtkVolume16Reader block in the top left is responsible for reading a medical data set from disk, while the vtkRenderWindowInteractor block on the far right is responsible for checking the display window to see if the user clicks on or drags the mouse within the window. By dragging the mouse in the window, the user is able to rotate the view of the data set.

SimVTK is not always used in isolation, Figure 4 demonstrates the use of SimVTK together with our SimITK package [4]. SimITK moves image data between blocks by using two MATLAB arrays as signals; one of the arrays holds the image itself, while a second array holds the pixel or voxel size, usually measured in millimetres, and the image position within a predefined cartesian coordinate system, e.g. the CT scanner coordinate system. In order to connect SimITK blocks to SimVTK blocks, we wrote a vtkITKImageImport block that takes the aforementioned SimITK arrays as input, and produces a vtkImageData object as output. When the ITK/VTK model shown in the figure is run, VTK displays a side-by-side view of the source and target images of an ITK image registration, and updates the view after each optimization step. This is a useful image registration tool, not only because it allows the user to clearly see each part of the registration pipeline and adjust various parameters, but also because it allows the user to see the evolution of the registration.

6 Future Work

We believe that SimVTK represents a practical approach to visual programming with VTK, since it combines a popular open-source visualization library with a ubiquitous commercial research package. All of the VTK classes that are available from the VTK wrapper languages are available in SimVTK, with the following exceptions: the SQL database classes, the vtkWidgetEvent class, and the vtkTextMapper class.

This project is still in the prototype stage, hence the labeling of our current release as SimVTK-0.2.4. Although nearly every VTK class is wrapped, the only VTK class methods are wrapped are the simple Set/Get/Add/Remove methods that are used to change parameter values. The VTK event-handling mechanism is also hidden from Simulink, so event handling cannot be customized in SimVTK (however, VTK classes like the vtkRenderWindowInteractor that use event handling internally are fully operational). As well, we do not yet provide a "make install" rule in our CMake scripts, hence the mex files must be manually copied from the build tree to create a binary package. Our main items for future work, in addition to fixing the above issues, are to provide a means of adding a Qt GUI on top of a SimVTK pipeline, and to export the SimVTK models as XML pipeline descriptions, similar to those used by VisTrails, and use those XML descriptions to generate pure C++ VTK/Qt visualization applications.

References

[1] L. Bavoil, S.P. Callahan, P.J. Crossno, J. Freire, C.E. Scheidegger, C.T. Silva, and H.T. Vo. VisTrails: Enabling interactive multiple-view visualizations. In *Proc. IEEE Visualization*, pages 135–142, 2005. http://www.vistrails.org. 1

[2] I. Bitter, R. Van Uitert, I. Wolf, L. Ibáñez, and J.-M. Kuhnigk. Comparison of four freely available frameworks for image processing and visualization that use ITK. *IEEE Trans. Vis. Comp. Graph.*, 13:483–493, 2007. 1

[3] C.P. Botha and F.H. Post. Hybrid scheduling in the DeVIDE dataflow visualisation environment. In H. Hauser, S. Strassburger, and H. Theisel, editors, *Proc. Simulation and Visualization*, pages 309–322. SCS Publishing House Erlangen, 2008. 1

[4] D.G. Gobbi, P. Mousavi, K.M. Li, J. Xiang, A. Campigotto, A. LaPointe, G. Fichtinger, and P. Abolmaesumi. Simulink libraries for visual programming of VTK and ITK. In *Systems and Architectures for Computer Assisted Interventions*. MICCAI Workshop, 2008. http://hdl.handle.net/10380/1461. 5

[5] K. Martin and B. Hoffman. *Mastering CMake*. Kitware, Inc., Clifton Park, New York, 5th edition, 2010. http://www.cmake.org. 2

[6] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Proc. ACM/IEEE Conference on Supercomputing (CD-ROM)*, page 52, 1995. http://www.scirun.org/. 1

[7] W. Schroeder, K.W. Martin, and W. Lorensen. *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Kitware, Inc., Clifton Park, New York, 4th edition, 2006. http://www.vtk.org/. 1

[8] I. Wolf, M. Nolden, T. Schwarz, and H.-P. Meinzer. Integrating the visualization concept of the medical imaging interaction toolkit (MITK) into the XIP-Builder visual programming environment. In K.H. Wong and M.I. Miga, editors, *Medical Imaging 2010: Visualization, Image-Guided Procedures, and Modeling*, volume 7625, page 762504. SPIE, 2010. 1