VisTrails

User's Guide

Version 1.0

September 28, 2007

This is the preface.

Contents

Chapter 1

What Is VisTrails?

VisTrails is a new system that provides data and process management support for exploratory computational tasks. It combines features of workflow and visualizations systems. Similar to workflow systems, it allows the combination of loosely-coupled resources, specialized libraries, and grid and Web services; and similar to some visualization systems it provides a mechanism for parameter exploration and comparison of different results. But unlike these, VisTrails was designed to manage exploratory processes, in which, computational tasks evolve over time as a user iteratively adjusts them while formulating and testing hypotheses. A key distinguishing feature of VisTrails is a comprehensive provenance infrastructure that maintains detailed history information about the steps followed in the course of an exploratory task. VisTrails leverages this information to provide novel operations and user interfaces that streamline this process.

Important Features. One of our main uses for VisTrailshas been exploratory visualization, but the system is much more general, and contains many other features. Additional feature that might be relevant include:

- *Flexible Provenance Architecture.* VisTrails transparently tracks changes made to workflows, all the steps followed in the exploration. The system can optionally track run-time information about the execution of workflows (*e.g.,* who executed a module, on which machine, elapsed time *etc.*). VisTrails also provides a flexible annotation framework whereby users can specify application-specific provenance information.

- *Querying and Re-using History.* The provenance information is stored in a structure way. Users have a choice of using a relational database (*e.g.,* MySQL and IBM DB2) or XML files in the file system. The system provides flexible and intuitive query interfaces through which users can explore and re-use provenance information. Users can formulate simple keyword-based and selection queries (*e.g.,* find a visualization created by a given user) as well as structured queries (*e.g.,* find visualizations that apply simplification before an isosurface computation for irregular grid data sets).

- *Support for collaborative exploration.* The system can be configured with a database backend that can be used as a shared repository. It also provides a synchronization facility that allows users to collaborate asynchronously and in a disconnected fashion—users can check in and check out changes, akin to a version control system (*e.g.,* SVN–http://subversion.tigris.org).

- *Extensibility.* VisTrails provides a very simple plugin functionality that can be used to dynamically add packages and libraries. Neither changes to the user interface nor re-compilation of the system are necessary. Because VisTrails is written in Python, the integration of Python-wrapped libraries is straightforward. For example, a single line in the VisTrails start-up file is needed to import all of VTK's classes.

- *Scalable Derivation of Data Products and Parameter Exploration.* A series of operations is supported by VisTrails for the simultaneous generation of multiple data products, including an interface that allows users to specify sets of values for different parameters in a workflow. The results of a parameter exploration can be displayed side by side in the VisTrails Spreadsheet for easy comparison.

- *Task Creation by Analogy.* Analogies are supported as first-class operations to guide semi-automated changes to multiple workflows, without requiring users to directly manipulate or edit the workflow specifications.

Obtaining the Software. Visit http://www.vistrails.org to access the VisTrails community Web site. Here you will find information including instructions for obtaining the software, online documentation, video tutorial, and pointers to papers and presentations.

VisTrails is written in Python and it uses the multi-platform Qt library for its user interface. The system available as open source—it is released under the GPL 2.0 license. The pre-compiled versions for Windows, Mac OS X, and Linux, come with an installer and include a number of packages, including VTK, matplotlib, and Image Magick. Additional packages, including packages written by users, are also available (*e.g.,* ITK, Matlab, Metro). It is easy to add new packages using the VisTrails plugin infrastructure.

Chapter 2

Getting Started

The VisTrails system is distributed both as source code and pre-built binaries, and instructions for obtaining either can be found at our website: http://www.vistrails.org. Because the system is written in Python using a Qt interface, it can be run on most architectures that support these two components, even if a pre-built binary is not available for your system. Section 2.1 provides instructions to guide you through installation procedures, and Section 2.2 gives a quick orientation and serves as a jumping off point to exploring the different features of VisTrails.

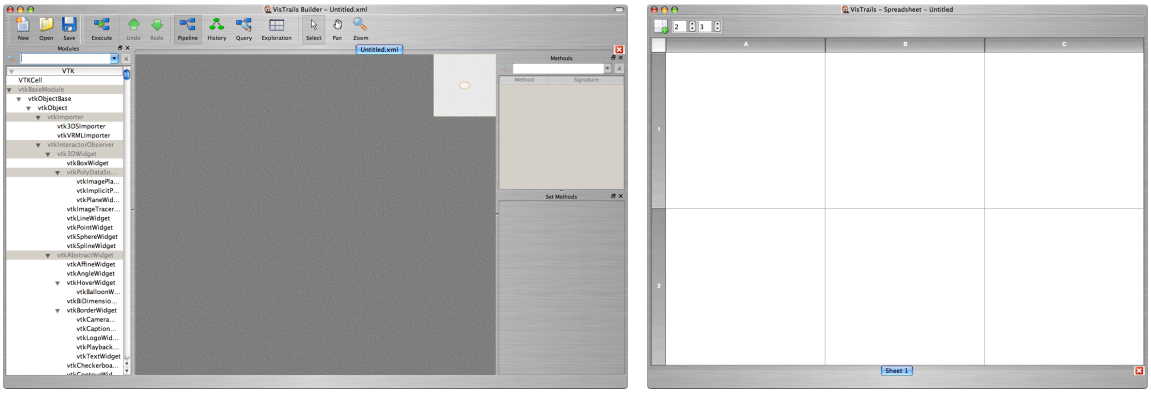
2.1 Installation

To obtain a copy of VisTrails, please see our website: http://www.vistrails.org. There, you will find binaries for Windows (tested on XP and Vista) and Mac OS X (tested on 10.4.x). In addition, there are instructions for obtaining a binary version for Ubuntu Linux. If you do not have one of these systems or would prefer to install VisTrails from source, you can also download the source. If you decide to install from source, please follow the instructions on the website as there are a few dependencies that you will need to make sure are installed. We encourage first-time users to download a binary version.

After obtaining a copy of VisTrails, installation is system-dependent, but both of the Windows and Mac OS X binaries come with installers that should be familiar to most users. In addition to the base VisTrails system, these installers also include a number of packages including VTK and matplotlib. For other versions, you should be able to install VisTrails to its own directory with the same set of packages. Again, refer to the website for specific instructions or help with installation.

2.2 Quick Start

Starting VisTrails is mildly system dependent. On Windows and Mac OS X, it requires clicking on the VisTrails application icon. In general, however, it is possible to start VisTrails on any



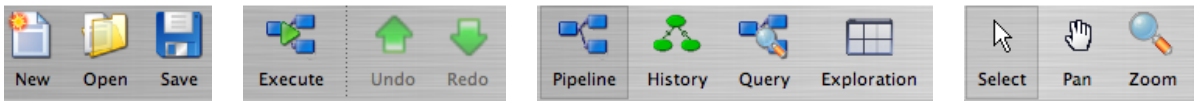
(a) VisTrails Builder Window (b) VisTrails Spreadsheet Window

Figure 2.1: Default View Upon Starting VisTrails

system by navigating to the directory where "vistrails.py" lives (usually the root directory of your installation) and executing the command:

```
python vistrails.py
```

Depending on a number of factors, it can take a few seconds for the system to start up. As VisTrails loads, you may see some messages that detail the packages being loaded and initialized. This is normal operation, but if the system fails to load, these messages will provide information that may help you understand why. After everything has loaded, you will see the VisTrails Builder window as shown in Figure 2.1(a). If you have enabled the VisTrails Spreadsheet, you will also see a second window like that in Figure 2.1(b). (Note that the spreadsheet is enabled by default.)



(see Section 2.3) (see Section 2.5) (see Chapters 3,4,6,7) (see Section 2.5)

Figure 2.2: The VisTrails toolbar

The VisTrails toolbar serves to help users navigate the various modes and functions VisTrails provides. As illustrated by Figure 2.2, the left side of the toolbar contains standard file manipulation buttons, and the next section provides buttons for execution and undo/redo functionality. The four buttons in the middle section serve to switch between different modes to manipulate, query, and explore workflows. The right-most buttons allow the user to toggle between different ways of navigating around the current canvas.

2.3 Manipulating VisTrails Files

To open a VisTrails file, or *vistrail*, you can either click the open button in the toolbar or select `Open` from the `File` menu. This brings up a standard file dialog where you can select a vistrail to open. Vistrails are identified by the ".vt" file extension. Opening a vistrail adds a tab to the builder window where each tab represents a different vistrail. Clicking a tab switches the current vistrail. Vistrails can also be stored in a database, enabling a central repository for workflows. See Chapter 9 for more details about this feature.

To close a vistrail, you can either choose the `Close` option from the `File` menu or click the red 'X' button on the upper right side of the builder window. If the vistrail has not been saved, you will asked if you wish to save your work. To save a vistrail, there is both a button and a menu item in the `File` menu. If you would like to save the vistrail with a different name or in a different location, you can use the `Save As` option.

2.4 VisTrails Basics

If you are already familiar with workflows and workflow systems, you can skip this paragraph. In general, a *workflow* is a way to structure a complex computational process that may involve a variety of different resources and services. Instead of trying to keep track of multiple programs, scripts, and their dependencies, workflows abstract the details of computations and dependencies into a graph consisting of computational *modules* and *connections* between these modules.

VisTrails exhibits an interface for building workflows that is similar to many existing workflow systems. As such, it allows users to interactively create workflows using a extensible library of modules and a connection protocol thats helps a user determine how to connect modules. Users drag modules from a list of available modules to a workflow canvas to add them to a workflow. Each module has a set of input and output ports, and outputs from one module can be connected to inputs of another module, assuming that the types match. For more information on building workflows in VisTrails, see Chapter 3.

In addition to providing an interface for manipulating individual workflows, VisTrails contains a number of features that function on a collection of workflows. A *vistrail* is a collection of related workflows. As you explore different computational approaches or visualization techniques, a workflow may evolve in a lot of directions. VisTrails captures all of these changes automatically and transparently. Thus, you can revisit a previous version of a workflow and modify it without worrying about saving intermediate versions. This history is displayed by the VisTrails Version Tree, and different ways of interacting with this tree are discussed in Chapter 4.

With a collection of workflows, one of the necessary tasks is to search for specific workflows. The criteria for these searches may vary from finding workflows modified within a specific time frame to finding workflows that contain a specific module. Because of the version history that

VisTrails captures, these tasks are natural to implement and query. VisTrails has two methods for querying workflows, a simple text-based query language and a query-by-example canvas that allows users to build exactly the workflow structure they are looking for. Both of these techniques are described in Chapter 6.

The fourth button that toggles between the different modes in VisTrails allows users to explore workflows by running the same workflow with different parameters. Parameter Exploration provides an intuitive interface for computing workflows with parameters that vary in multiple dimensions. When coupled with the VisTrails Spreadsheet, parameter exploration allows you to quickly compare results and discover optimal parameter settings. See Chapter 7 for specific information on using Parameter Exploration.

2.5 VisTrails Interaction

The `Execute` button serves as the "play" button for each of the modes describes above. In both the Builder and Version Tree, it executes the current workflow. For querying, it executes the query, and in parameter exploration, it executes the workflow for each of the possible parameter settings. The undo and redo buttons function in the standard way, but note that these actions are implicitly switching between different versions of a workflow. Thus, you will notice that as you undo or redo a change to a workflow, the selected version in the version tree changes.

For all modes except Parameter Exploration, the center pane of VisTrails is a canvas where you can manipulate the current workflow, version tree, or query workflow. The buttons on the right side of the toolbar allow you to channge the default behavior of the standard mouse button (the left button for most multiple button mice). You can choose the behavior to select items in the scene, pan around the scene, or zoom in and out of the scene by selecting the given button. In addition, if you are using a mouse with multiple buttons, the right button will zoom, and the middle button will pan. To use the zoom functionality, click and drag up to zoom out and drag down to zoom in.

Chapter 3

Creating and Modifying Workflows

3.1 Working with Modules

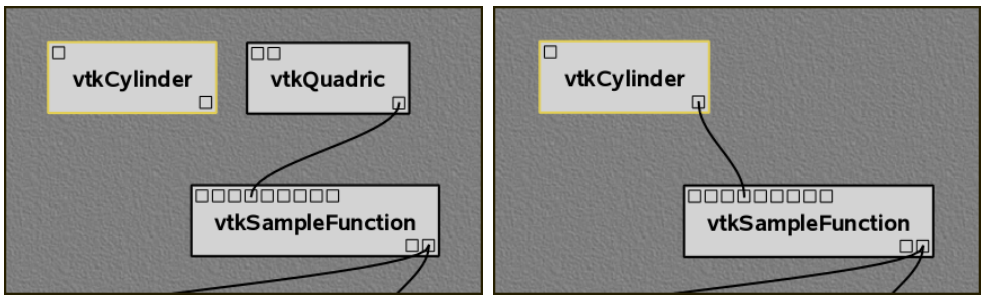
In VisTrails, modules are represented by a rectangle in the `Pipeline` view of the Builder. The name of the module is shown in bold letters in the middle of the rectangle. The input and output ports for the module are denoted by small squares on the top and bottom of the module, respectively. Modules are connected together to define the dataflow using curved black lines that go from output to input ports between modules. Each module may have also have adjustable parameters that can be viewed when a module is selected. Modules can be connected, disconnected, added, and deleted from a workflow.

As an example of building workflows, we will modify a basic VTK workflow by replacing one module with another in the `final` workflow of the "vtk_book_3rd_p189.vt" vistrail. After opening this vistrail, you will need to click on the `Pipeline` button in the toolbar to edit the workflow.

3.2 Adding and Deleting Modules

A list of available modules is displayed hierarchically in the `Modules` container on the left side of the VisTrails Builder. A core set of basic modules are always distributed with the VisTrails system. Other packages, such as VTK, are also distributed, but are not necessary for VisTrails and thus can be disabled on startup (see Chapter 11). Depending on the number of packages imported on startup, the number of modules to select from can be difficult to navigate. Thus, a simple search box is provided at the top of the container to narrow the displayed results. To add a module to the workflow, simply drag the text from the `Module` container to the workflow canvas.

Modules and connections may be selected in multiple ways and are denoted by a yellow highlight. Besides directly left clicking on the object, a box selection is available by left clicking and dragging over the modules and connections. Multiple selection can be performed with the



(a) (b)

Figure 3.1: (a) The `vtkCylinder` module is added to the canvas, (b) the `vtkQuadric` module is deleted and the connection replaced.

box selection as well as by right clicking on multiple objects with the 'Shift' key pressed.

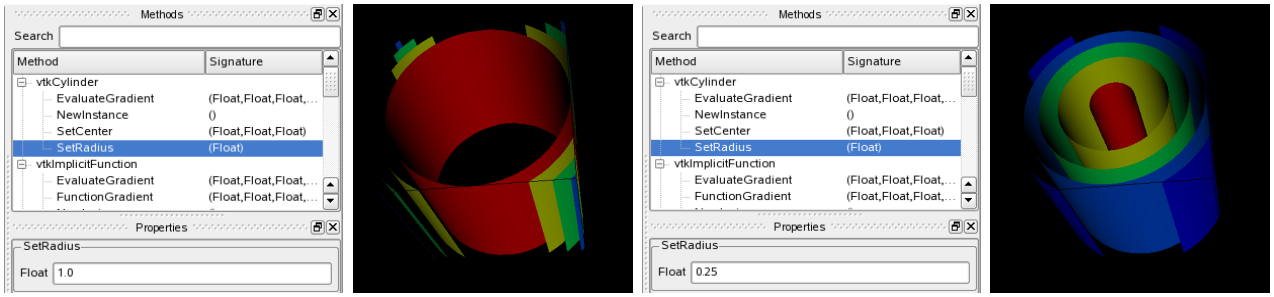
There are several ways to manipulate selected modules in the workflow canvas. Moving them is performed by dragging a selected module using the left mouse button. Deleting selected modules is performed by pressing the 'Delete' key. The modules and connections can also be copied and pasted using the `Edit` menu, or with 'Ctrl-c' and 'Ctrl-v', respectively.

In our running example, type "vtkCylinder" into the search box of the `Module` container. As the letters are typed, the list filters the available modules to match the query. Select this module and drag the text onto an empty space in the canvas. This module will replace the `vtkQuadric` module in our example. Thus, select the `vtkQuadric` module in the canvas and press the 'Delete' key. This removes the module along with any connections it has (see Figure 3.1).

3.3 Connecting Modules

Modules are connected in VisTrails through the input and output ports at the top and bottom of the module, respectively. By hovering the mouse over the box that defines a port, the name and data type are shown in a small tooltip. To connect two ports from different modules, start by left clicking inside one port, then dragging the mouse to the other. The connection line will automatically snap to the ports in a module that have a matching datatype. Since multiple ports may match, hovering the mouse over the port to confirm the desired match may be necessary. Once a suitable match is found, releasing the left mouse button will create the connection. Note, a connection will only be made if the input and output port's data types match. To disconnect a connection between modules, the line between the modules can be selected and deleted with the 'Delete' key.

Returning to our example, the new module `vtkCylinder` needs to be connected to the `vtkSampleFunction` module as the previous data source was. Place the cursor over the only output port on the `vtkCylinder` module, located on the bottom right. A tooltip should appear



(a) (b) (c) (d)

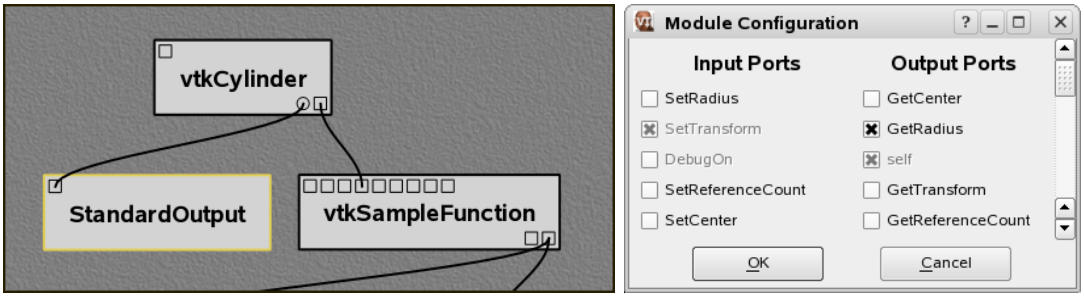
Figure 3.2: (a)(c) The module methods interface is shown with a change of the `SetRadius` parameter to 1.0 and 0.25. (b)(d) The results of the changes are displayed on execution.

that reads "Output port self (vtkCylinder)". Left click on the port and drag the mouse over the `vtkSampleFunction` module. The connection should snap to the fourth input port from the left. Hovering the mouse over this port shows a tooltip that read "Input port SetImplicitFunction (vtkImplicitFunction)". Release the mouse button to complete the connection between these two modules (see Figure 3.1). To check for a valide dataflow, execute the workflow and see if the results appear in the spreadsheet.

3.4 Changing Module Parameters

The parameters for a module can be accessed in the `Methods` container located on the right side of the Builder. When a module is selected from the canvas, the corresponding methods are displayed. As with the `Modules` container, a search box is provided to quickly find a desired method. By default, the Builder only manages methods with set parameters. To check the set parameters, a `Set Methods` container is available below the `Methods` container. Changing a parameter can be performed directly in the `Set Methods` container. To set a parameter for the first time, click on the corresponding method and drag it into the `Set Methods` container, then enter the parameters directly into the text boxes. To remove a set parameter, simply select the method in the `Set Methods` container and press the 'Delete' key.

To perform a parameter change with our example, select the `vtkCylinder` module in the canvas. The methods are shown hierarchically in the `Methods` container. Find the `SetRadius` method and select it, then drag the highlighted text from the `Methods` container into the `Set Methods` container below. The result is a `SetRadius` box with a `Float` text input. Enter 0.25 into the text box and press the 'Enter' key. By executing the workflow, the modified visualization appears in the spreadsheet. Figure 3.2 shows the interface and results of the parameter explorations.



(a) (b)

Figure 3.3: (a) The `vtkCylinder` module is configured to show an additional `GetRadius` port, which is then connected to a `StandardOutput` module. (b) The module configuration window allows the hidden ports to be displayed.

3.5 Configuring Module Ports

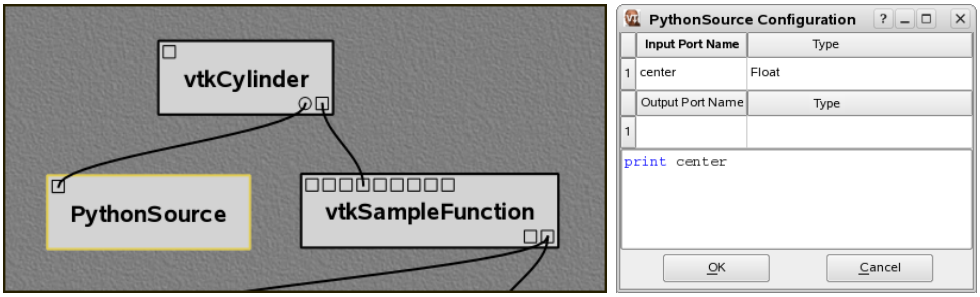
For convenience, all the inputs and outputs of a module are not always shown in the canvas as ports. The ports that are shown by default are defined using an option when defining the method signatures of a package. To access the full list of ports, the module configuration window is used. This is opened by selecting the triangle at the top right of a module to open a popup menu and selecting the `Edit Configuration` menu item, or alternatively by pressing 'Ctrl-E' when a module is selected. The window shows a list of input and output ports and allows the user to toggle any additional ports to enable. When the configuration is complete, the new ports will appear on a module with a circle icon instad of the normal square. These new ports can then be used for connections in the same way as the others.

As an example of configuring a module port, in our previous example select the `vtkCylinder` module in the canvas and press 'Ctrl-E'. In the newly opened configuration window, check the box for the `GetRadius` port, then click `OK` to close the window. A new circle port should appear on the module. Next, add a new `StandardOutput` module from the basic modules and connect the output port for `GetRadius` with the input port of `StandardOutput`. Upon execution, the value 0.25 is now output to the console. Figure 3.3 shows the new workflow together with the module configuration window.

3.6 Basic Modules

In addition to the modules provided by external libraries, VisTrails provides a few basic modules for convenience and to facilitate the coupling of multiple packages in one workflow. These modules mostly consist of basic data types in Python and some manipulators for them. In addition, file manipulation modules are provided to read files from disk and write files to disk.

Because every Python operation cannot be represented as a module, the `PythonSource`



(a) (b)

Figure 3.4: (a) A `PythonSource` module can be used to directly insert scripts into the workflow. (b) The configuration window for `PythonSource` allows multiple input and output ports to be specified along with the Python code that is to be executed.

module is provided to allow users to write Python statements to be executed as part of a workflow. By pressing 'Ctrl-E' when a `PythonSource` module is selected in the canvas, a configuration window is opened. This window allows the user to specify custom input and output ports as well as directly enter Python source to be executed in the workflow.

To demonstrate a `PythonSource` module, we return to our example. Instead of using a `StandardOutput` module as above, we will output the center of the cylinder using Python. First, add a `PythonSource` module to the canvas and remove the `StandardOutput` module. Select the `PythonSource` module and press 'Ctrl-E' to edit the configuration. In the newly opened configuration window, create a new input port named "center" of type `Float`. Next, in the source window enter:

```
print center
```

then select `OK` to close the window. Finally, connect the `GetRadius` output of the `vtkCylinder` module to the new input port of `PythonSource`. Upon execution, the radius of the cylinder is printed to the console as before. Figure 3.4 shows the new workflow together with the `PythonSource` configuration window.

Chapter 4

Interacting with the Version Tree

4.1 Version Tree View

The `History` button on the VisTrails Toolbar lets users interact with a workflow history. It consists of a tree view in the center and the `Properties` tool window on the right for querying and managing version properties. Versions are displayed as ellipses in the tree view where the root of the tree is displayed at the top of the view. The nodes of the tree correspond to a version of a workflow while an edge between two nodes indicates that one was derived from the other.

By default, only nodes that are leaves, have more than one child node, are specially tagged, and the current version, will be displayed. The nodes are displayed as colored ellipses, and are either blue or orange. A blue color denotes that the corresponding version was created by the current user while orange nodes were created by other users. The brightness of each node indicates how recently a version was created; brighter nodes were created more recently than dimmer ones. Each node may also have a *tag* that describes the version, and this tag is displayed in the center of the ellipse of the corresponding version.

All of the versions are connected to each other by either solid or broken lines. A solid line indicates that the child node is a direct descendant of the parent node, meaning the user has made only a single change from the older version to the newer version. Likewise, a broken line indicates that more than one change has been made, but the intermediate versions have not been tagged. Because most non-trivial changes to a workflow take more than action, most edges in a the version tree will be shown as these broken lines.

To see an example of a version tree, load the example vistrail "vtk_book_3rd_p189.vt". All versions will be shown in orange unless your username happens to be "emanuele". Recall that this tree displays the structure of changes to a workflow so let's make some changes to see their effect on the version history. In the `History` view, select the node tagged `Almost there`, and then click on the `Pipeline` button to switch to a view of the workflow. Select a connection and delete it. Now, switch back to the `History` view, and notice that there is a new child

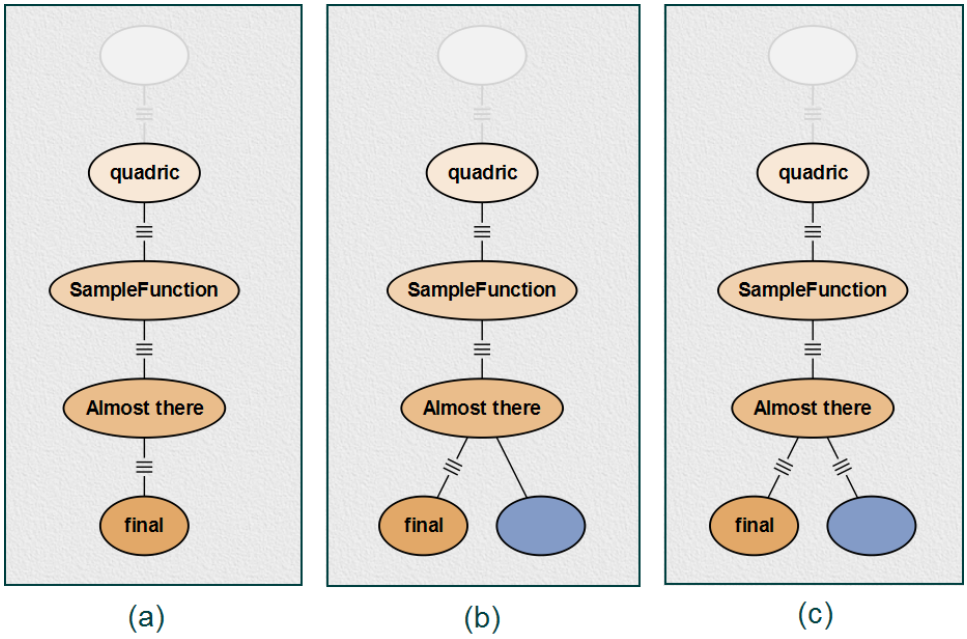


Figure 4.1: (a) All versions created by other users are shown in orange. (b) Deleting a connection results in a blue version connected by a solid line. (c) More interactions on this version will cause the solid line become a broken one.

node connected to `Almost there`. In addition, the line connecting the new node to its parent is solid, indicating that only a single change has been made. If we delete more connections, the solid line would become a broken line. See Figure 4.1.

4.2 Adding and Deleting Tags

As noted above, only certain nodes, including specially tagged ones, are shown by default in the version tree. To tag a version, simply add meaningful text to the tag text box in the `Properties` window and press 'Return'. If you would like to change the tag to a different text, click in the same text box and modify the string, again hitting 'Return' when finished. Note that deleting all of the text in the tag field effectively deletes the tag. A second way to delete a tag is to click the 'X' button to the right of the text box. Removing a tag from a node may cause it to not be displayed in the default version tree view if it doesn't satisfy any of the other criteria for display.

4.3 Adding Version Annotations

In addition to the tag field, the `Properties` window also displays information about the user who created the selected version and when that version was created. At the top of the window is a field for querying a vistrail, and this functionality is described in detail in Chapter 6. The final piece of the window is the `Notes` field which allows users to store notes or annotations related to a version. As with tags, adding notes to a version is as easy as selecting the desired version and modifying the text field. Notes are automatically saved when you save the vistrail file.

4.4 Navigating Versions

Besides clicking on nodes of a version tree, you can also use the `Undo` and `Redo` buttons to change versions. Because the version tree captures all changes to a workflow, undo and redo not only revert or reinstate changes to a workflow, but also change the currently selected version in the version tree. More precisely, undoing a change in a workflow is exactly the same as selecting the parent of the current node in the version tree. Note that because the current version is always shown in the version tree, undo and redo provide an effective way to navigate between two nodes connected with a broken line.

4.5 Comparing Versions

While selecting versions in the `History` view and using the `Pipeline` view to examine each version is extremely useful, it can be cumbersome when trying to compare two different versions. To help with such a comparison, VisTrails provides a the `Version Difference` mechanism for quickly comparing two versions. There are two ways to compute this difference. The first is to select two versions in the tree and choose `Execute Version Difference` from the `Run` menu. The easier method is to drag one version onto the other.

After either method, a `Visual Diff` window will open (see Figure 4.2). The difference is displayed in a manner that is very similar to the pipeline view, but modules and connections are colored based on similarity. Dark gray indicates those modules and connections that are shared between the two versions; orange and blue show modules and connections that exist in one workflow and not the other; and light gray modules are those where parameters between the two versions differ. Clicking the `Legend` button will bring up a window to remind you what each color corresponds to. For a module that is colored light gray, clicking on the `Parameter Changes` button will bring up a window that shows the difference in parameters for that module.

Figure 4.2 shows the result of comparing the `z-spaced` and `textureMapper` versions in the "lung.vt" example. To try out this feature, click and drag the `z-space` version to the `textureMapper` version. Note that the cursor icon will change to a green plus when the drag

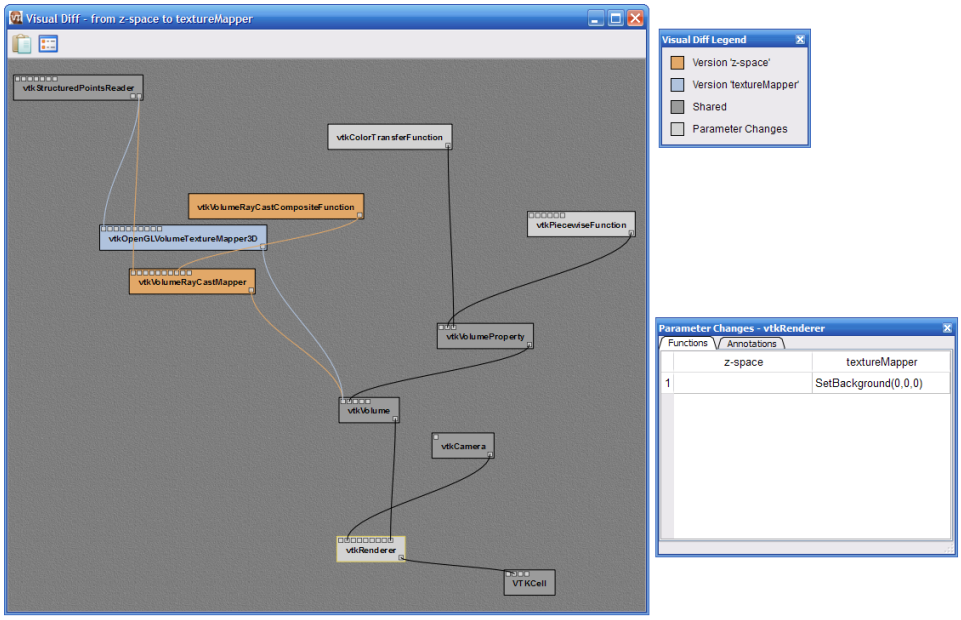


Figure 4.2: A `Visual Diff` showing the difference between version `z-space` and version `textureMapper`.

is valid. After the diff appears, click on the `Parameter Changes` button, and then click on the `vtkRenderer` module to see the parameter differences. We can see that one of the changes from `z-space` to `textureMapper` was to add a black background.

Chapter 5

The Spreadsheet

As described in Section 4.5, VisTrails has a powerful built-in mechanism to compare workflows. However, this comparison shows changes in the *design* of the workflows, and we are often also interested in differences in the *results* of workflows. The VisTrails Spreadsheet provides a simple, flexible, and extensible interface to display and compare results from workflows. Coupled with the version differences, users can explore the evolution of their workflows.

The Spreadsheet package is installed with VisTrails by default, and it can display a variety of data ranging from VTK renderings to webpages without additional configuration. In addition to the included types of viewers, users can create and register additional viewers using customized cell widgets (see Section 5.4).

5.1 The Spreadsheet Layout

As should be expected, the VisTrails Spreadsheet consists of one or more sheets, each with a customizable number of rows and columns. Users can add additional sheets either by clicking the `New Sheet` button in the `Spreadsheet` toolbar or choosing the menu item with the same name from the `Main` menu. Similarly, a sheet can be deleted by clicking the 'X' button in the lower-right corner or choosing the `Delete Sheet` menu item.

To modify the layout for the active sheet, you can both change the number of rows and columns and resize individual cells. The number of rows is controlled by the left spinner in the toolbar and the number of columns by the right one. To resize a given row or column, click and drag on one edge of the row or column header. In addition, you can resize an individual cell by moving the mouse to lower-right corner of the cell until the cursor changes and clicking and dragging to the desired size (see Figure 5.2(d)). Note that this will affect the entire layout, compressing or expanding rows and columns to generate or fill space for resized cell.

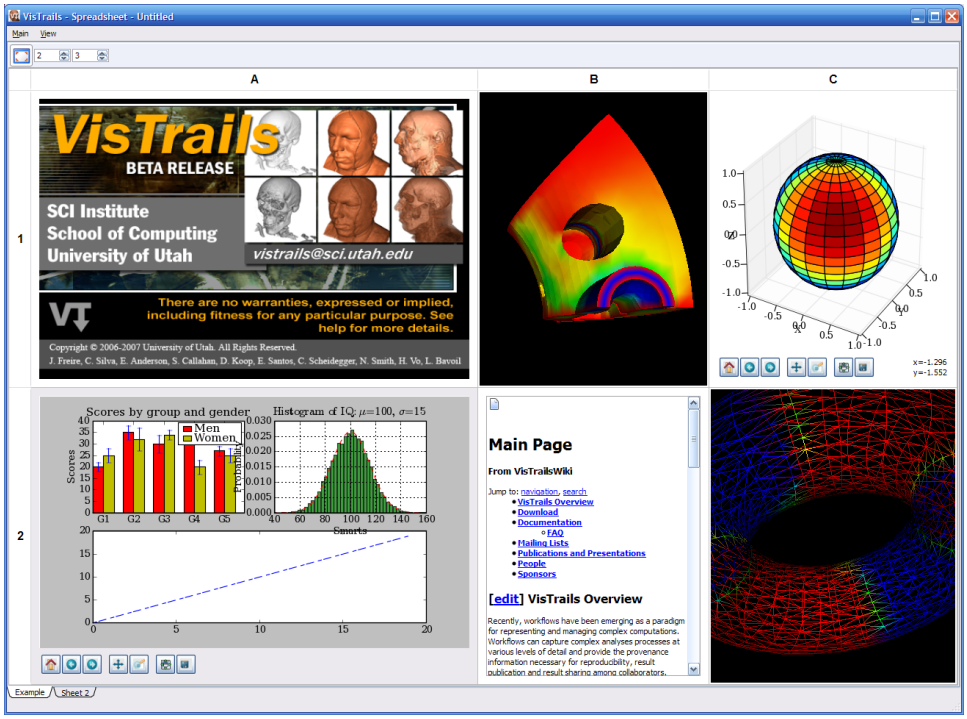


Figure 5.1: The VisTrails Spreadsheet

5.2 Using the Spreadsheet

Currently, there are two operating modes in the Spreadsheet: Interactive Mode and Editing Mode. Interactive Mode allows users to view and interact with the spreadsheet cells while Editing Mode provides operations for manipulating cells. The modes can be toggled by the `View` menu or their corresponding keyboard shortcuts ('Ctrl-Shift-I') and 'Ctrl-Shift-E').

5.2.1 Interactive Mode

In Interactive Mode, users can interact directly with the viewer for an individual cell, interact with multiple cells at once, or change the layout of the sheet. Because cells can differ in their contents, interacting with a cell changes based on the type of data displayed. For example, in a `VTKCell`, a user can rotate, pan, and zoom in or out using the mouse.

In a sheet, a cell can be both *active* and *selected*. There can only be one active cell, and that cell is highlighted by a yellow border. Clicking on any cell will make it active. This active cell will respond to keyboard shortcuts as well as mouse clicks and drags. In constrast to the active cell, one or more cells can be selected, and the active cell need not be selected. To select multiple cells, either click on a row or column heading to toggle selection or 'Ctrl'-click to add or remove a cell from the group of selected cells. The backgrounds of selected cells are

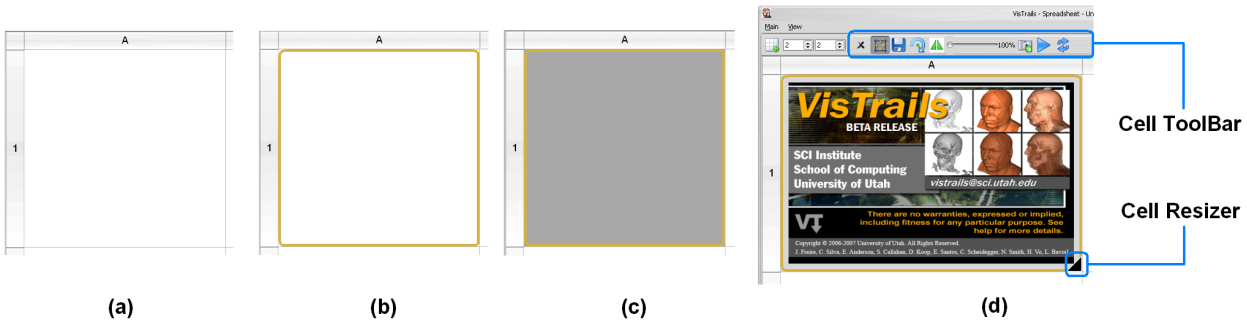


Figure 5.2: Different states of a spreadsheet cell. (a) inactive and unselected (b) active and unselected (c) active and selected (d) an active cell with its toolbar and resizer

highlighted using the system's selection color. See Figure 5.2 for examples of the different cell states.

Depending on the cell type, additional controls may appear in the toolbar when a cell is activated. These controls affect only the active cell, and change for different cell types. As shown by Figure 5.2(d), an `ImageViewerCell` adds controls for resizing, flipping, and rotating the image in the active cell.

Arranging Cells

As described in Section 5.1, cells can be resized by either resizing rows, columns, or an individual cell. In addition to resizing, a row or column can be moved by clicking on its header and dragging it along the header bar to the desired position. See Section 5.2.2 for instructions on moving a specific cell to a different location.

Synchronizing Cells

Often, when a group of cells all display results from similar workflows, it is useful to interact with all of these cells at the same time. For example, for a group `VTKCells`, it is instructive to rotate or zoom in on multiple cells at once and compare the results. For this reason, if a group of cells is selected, mouse and keyboard events for a single cell of the selection are propogated to each of the other selected cells. Currently, this feature only works for `VTKCells`, but we plan to add this to other cell types as well. An example of this functionality is shown in Figure 5.3.

5.2.2 Editing Mode

Editing Mode provides more operations to layout and organize spreadsheet cells. In this mode, the view for each cell is frozen and overlaid with additional information and controls (see Figure 5.4). The top of the overlay displays information about which vistrail, version, and type

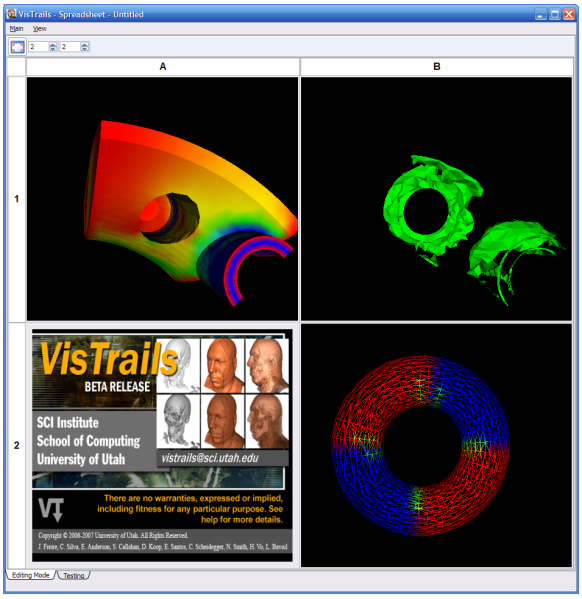


Figure 5.3: When selecting all cells, interacting with one VTK cell (A1) causes the other two VTK cells (B1 and B2) to change their camera to the same position.

of execution were used to generate the cell. The bottom piece of the overlay contains a variety of controls to manipulate the cell depending on the its state.

Cells can be moved or copied to different locations on the spreadsheet by clicking and dragging the appropriate icons (`Move` or `Copy`) for a given cell to its desired location. To move a cell to a location on a different sheet, drag the icon over the target sheet tab to bring that sheet into focus first and then drop it at the desired location. If you move a cell to an already-occupied cell, the contents of the two cells will be swapped. See Figure 5.4 for an example of swappng two cells.

If a cell was generated via parameter exploration (see Chapter 7), the `Create Version` button will be available to save the workflow that generated the result back to the vistrail. Clicking this button modifies the vistrail from which the cell was generated by adding a new version with the designated parameter settings. Thus, if go back to the `History` mode of the VisTrails Builder for that vistrail, you will find that a new version has been added to the version tree.

5.3 Saving a Spreadsheet

Warning: This is currently an experimental feature and as such is not robust. If you rename or move the vistrails used by the saved spreadsheet, the spreadsheet will not load correctly.

Because spreadsheets can include several workflow executions or parameter explorations,

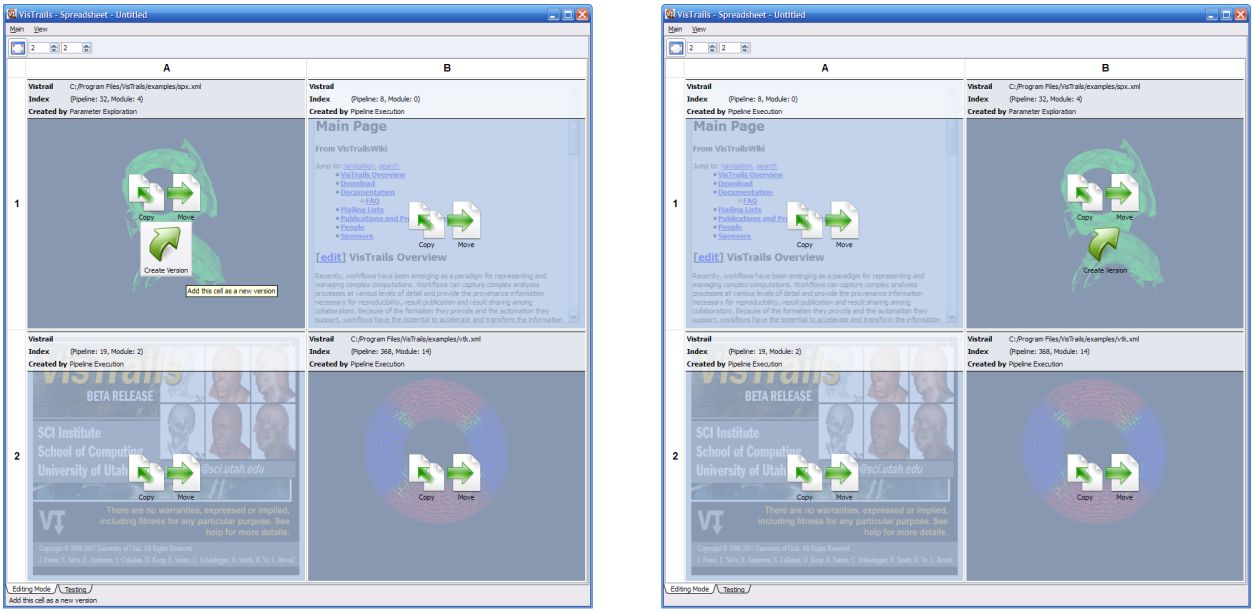


Figure 5.4: The spreadsheet in Editing Mode. (a) All cell widgets are replaced with an information widget (b) Two cells are swapped after drag and drop the 'Move' icon from A1 to B1

it is helpful to be able to save the layout of the current spreadsheet. To save a spreadsheet, simply choose the `Save` menu item from the `Main` menu, and complete the dialog. After saving a spreadsheet, you can reopen it using the `Open` menu item.

5.4 Creating a Customized Cell Widget

Incomplete: This section will be added at a later date. Check the web site for more information or contact the developers for more information.

Chapter 6

Querying the Version Tree

VisTrails is designed for manipulating collections of workflows, and an integral part of this design is the ability to quickly search through these collections. VisTrails provides two methods for querying vistrails and workflows. The first is a query-by-example interface which allows users to build query workflows and search for those with similar structures and parameters, and the second is a textual interface with a straightforward syntax. For each interface, the results are emphvisual: each matching version is highlighted in the `History` view, and if the query involves specific workflow characteristics, any matching entities are also highlighted in the `Pipeline` view for the current version.

6.1 Query By Example

One of the problems faced when trying to query a collection of workflows is the fact that structure is important. Suppose that you want to find only workflows where two modules are used in sequence. Instead of trying to translate this into a text-based syntax, it is easier to construct this relationship. VisTrails provides such an interface which mirrors the `Pipeline` view, allowing users to construct a (partial) workflow to serve as the search criteria.

To use the Query by Example interface, click on the `Query` button on the toolbar. This view is extremely similar to the `Pipeline` view and pipelines can be built in a similar manner. Just like the `Pipeline` view, modules are added by dragging them from the list on the left side of the window, connections are added by clicking and dragging from a port on one module to a corresponding port on another module, and parameters can be edited on the right-side of the window. One major difference between the `Pipeline` view and the `Query` view is that you can use comparison operations in parameter values. For example, instead of searching for a pipeline that contains a Float with a value of `4.5`, you can search for a pipeline that contains a Float with a value `'< 4.5'` or `'> 4.5'`. Figure 6.1 shows an example pipeline that has been built in the query builder.

Note that Query by Example provides the capability to iteratively refine searches by adding

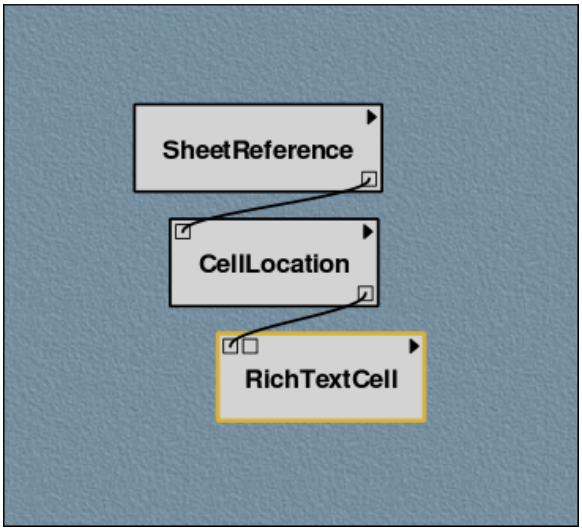


Figure 6.1: Example pipeline in query mode.

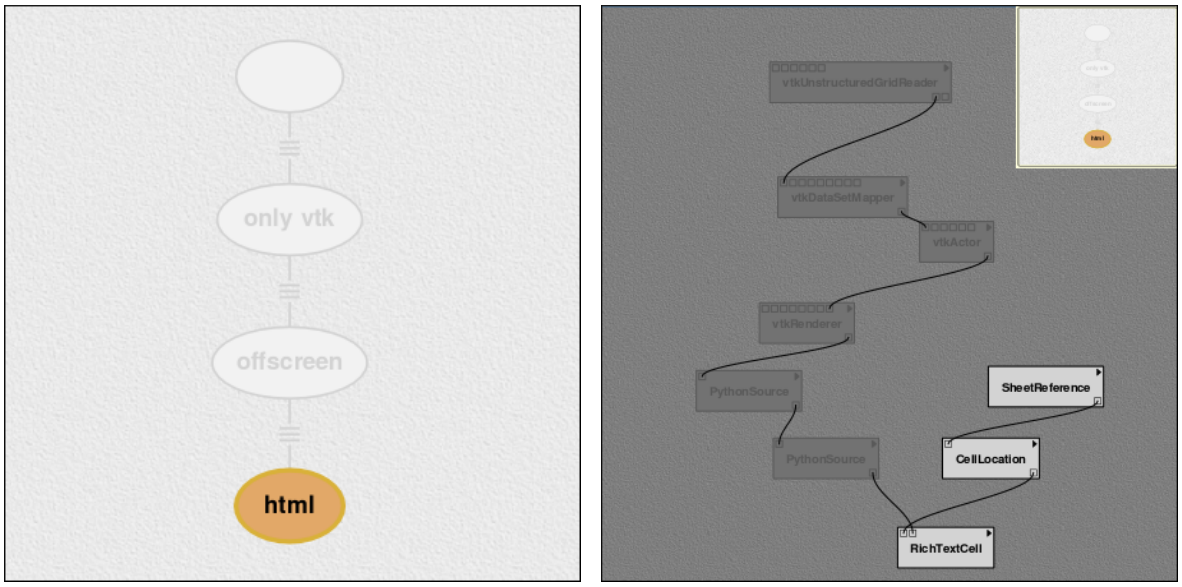
more criteria. For example, a user interested in workflows that contain a certain module may find that such a query returns too many results. That user can refine the query to only find those workflows where the given module has a parameter setting that falls in a given range.

After constructing a pipeline, click the `Execute` button to begin the query. This button will be available as long as the query window is not empty. Executing the query will bring you back to the `History` view where the matching versions are displayed. Section 6.3 provides information on interacting with query results.

6.2 Textual Queries

There are many ways to search for versions in the version tree using texutal queries, but they all rely on a simple text box for input. Begin a search by activating the `History` view. The search box is in the `Properties` subwindow, and can be identified by the magnifying glass icon next to it. If you enter query text, VisTrails will attempt to match logical categories, but if your query is more specific, VisTrails has special syntax to markup the query. Figure 6.2 shows an example query. To execute a query, simply press the 'Return' key after typing your query.

Table 6.1 lists the different ways to markup a query. Note that you can search by user name to see which changes a particular user has made and also by date to see which changes were made in a specific time frame. When searching by date, you can search for all changes before or after a given date or an amount of time relative to the present. If searching for changes before or after a specific date, the date can be entered in a variety of formats. The simplest is '*day month year*', but if the year is omitted, the current year is used. The month may be specified by either name or numerical value. For example '`before: 18 November 2004`' is a



(a) (b)

Figure 6.2: (a) Query results in history view and (b) the Results in pipeline view.

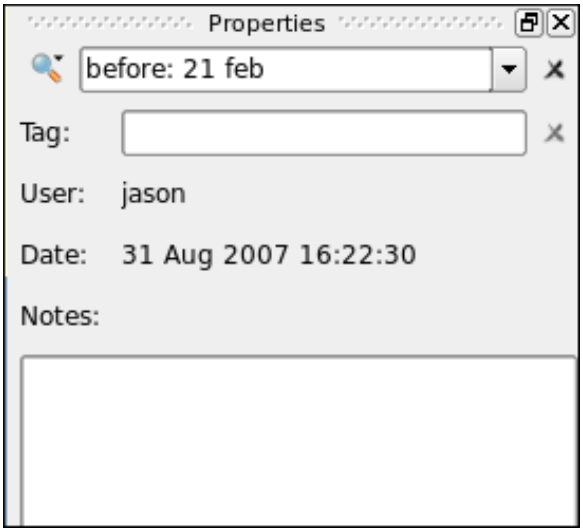


Figure 6.3: A query made to find any changes made before Feb 19.

Search Type	**Syntax**
User name	`user:` *user name*
Annotation	`notes:` *phrase*
Tag	`name:` *version tag*
Date	`before:` *date*	*relative time* `after:` *date*	*relative time*

Table 6.1: Syntax for querying specific information using textual queries.

valid query. If searching by relative time, you can prepend the amount of time relative to the present including the units to 'ago'. An example of this type of query is '`after: 30 minutes ago`'. The available units are seconds, minutes, hours, days, months, or years.

You can concatenate simple search statements to create a compound search to search across different criteria or for a specific range. For example, to search for workflows whose tag includes '`brain`' and were created by the user '`johnsmith`', the query would be '`name: brain user: johnsmith`'. To search for all workflows created between April 1 and June 1, the query would be '`after: April 1 before: June 1`'.

6.3 Query Results

After executing either a query by example or a textual query, the matching versions are highlighted in the version tree. In addition, there is a button named `Reset Query` in the lower-left of the version tree that allows you to reset the query, returning the view to normal. For queries by example, if you click on a specific matching version and change to the `Pipeline` view, the matching structure will also be highlighted. Figure 6.2 shows the results of the query by example in Figure 6.1 in both the `History` and `Pipeline` views.

While in the `History` view, you can select two different ways of viewing search results. The magnifying glass icon to the left of the textual search box contains a dropdown menu with two options: "Search" and "Refine". The first displays results by simply highlighting the matching nodes while the second condenses the tree to show only the versions that match. For large vistrails, this second method can help you determine relationships between the matching versions more easily.

In addition, VisTrails keeps track of the most recent textual queries, and repeating these queries can be accomplished by selecting the recent query from the dropdown menu attached to the search box. You can also clear recent searches using this menu. Finally, the 'X' button next to the search box will reset the query and restore the normal view of the version tree.

Chapter 7

Parameter Exploration

While exploring workflows, one critical task is tweaking parameter values to improve simulations or visualizations. VisTrails contains an integrated parameter exploration interface that allow users to thoroughly explore the parameter space and quickly identify their desired settings. By binding parameters to a range of values, users can generate a collection of results without having to tediously edit the workflow.

VisTrails Parameter Exploration is Spreadsheet-aware so users can map the intermediate results from explorations to cells of the Spreadsheet. Because the Spreadsheet provides a multi-view, gridded, interface that makes efficient use of screen space, users can quickly compare the results of different parameter settings. The changes in parameters can be displayed across rows, columns, and sheets. In addition, parameters can be explored across timesteps, and displayed in the Spreadsheet as animatations. This could be used, for example, to show how pathological tissues and tumors are affected by radiation treatment in a series of scans.

7.1 Creating a Parameter Exploration

Before beginning to explore parameters, make sure that the workflow that you wish to explore is active. See Chapter 4 for information on selecting a specific workflow. To access VisTrails Parameter Exploration, click on the `Exploration` button in the VisTrails toolbar.

The `Parameter Exploration` window (shown in Figure 7.1) is centered around a tabular environment where the exploration parameters can be setup. On the right side of the window, there are a variety of panels that control aspects of the exploration (see Figure 7.4: the `Set Methods` panel contains the list parameters that can be be explored; the `Annotated Pipeline` panel displays the workflow to be explored an helps resolve ambiguities for parameter settings, and the `Spreadsheet Virtual Cell` aids users in laying out exploration results in the spreadsheet.

The columns headings of the main exploration window control how parameter values are interpolated. The five controls on the to right side determine, from left to right, exploration in

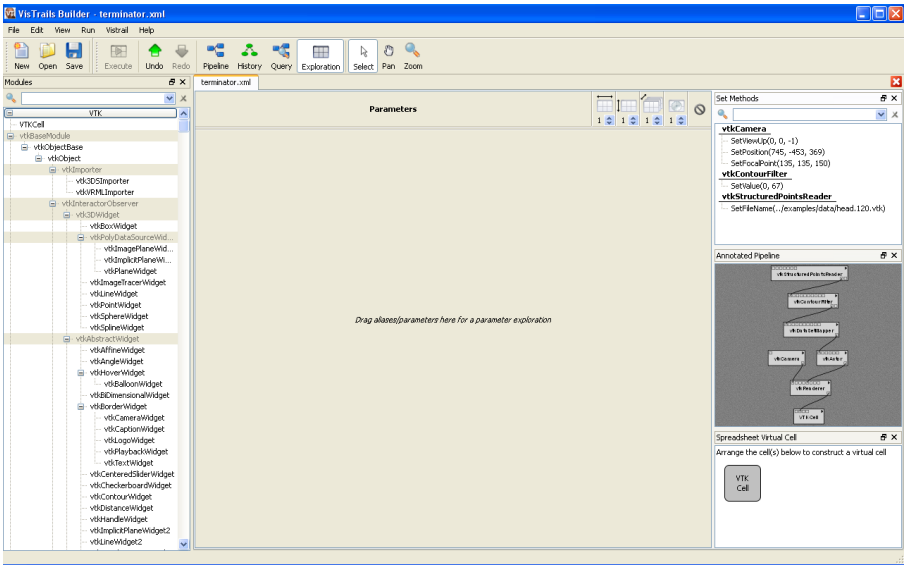


Figure 7.1: Parameter Exploration Window

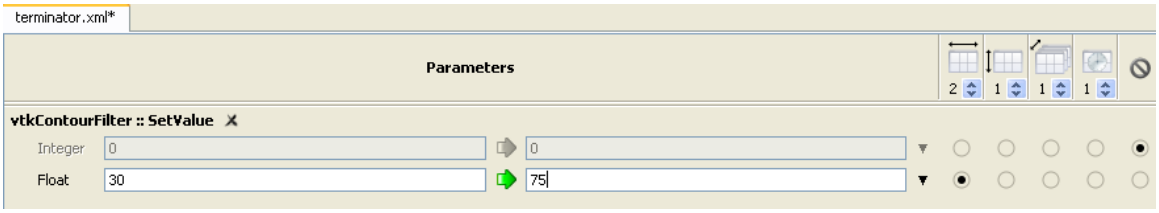
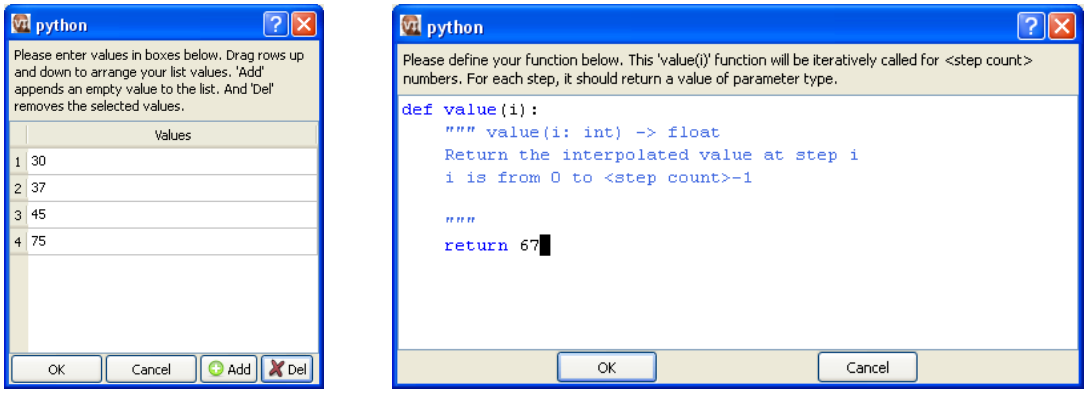


Figure 7.2: Setting values for parameter exploration.

the 'x' direction, the 'y' direction, the 'z' direction, time, and no direction. The spinner next to each of these controls the number of parameter values to be explored in that direction, and for each parameter, you can select *one* of the directons to explore that parameter's values.

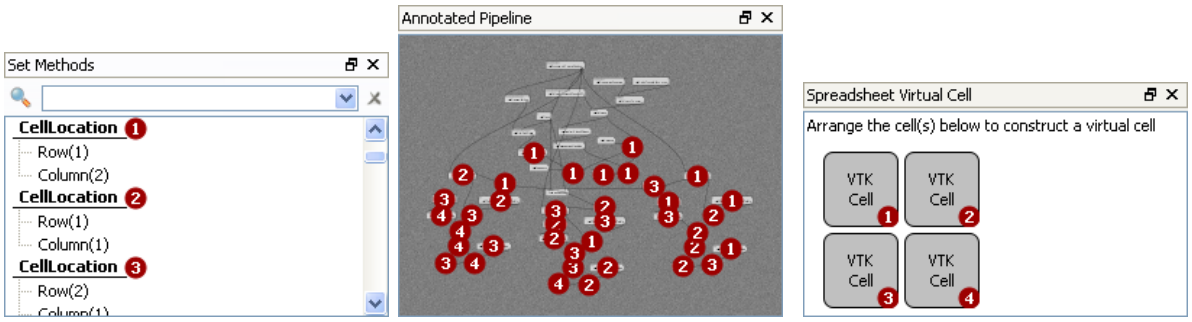
To add parameters to an exploration, simply drag the corresponding method from the `Set Methods` panel to the center canvas. To reduce clutter, this panel only shows the methods for which parameters were assigned values in the `Pipeline` view. See Chapter 3 for instructions on adding methods and parameters to a module.

After dragging a method to the exploration canvas, you can, for each parameter, set the collection of values to be explored and the direction in which to explore. See Figure 7.2 for an example. The collection of values can be set by linear interpolation, a list of values, or a user-defined function. You can choose the desired method from the drop-down menu on the right side of the parameter heading. For linear interpolation, the starting and ending values must be specified; for a list, the entire comma-separated list must be specified, and for a user-defined function, a Python function must be specified. For the list and user-defined functions, you can access an editor via the '...' button. See Figure 7.3 for an example. In addition, you



(a) (b)

Figure 7.3: Editors for (a) lists of values, and (b) user-defined functions.



(a) (b) (c)

Figure 7.4: The right panels of the Parameter Exploration window. The numbered red circles distinguish duplicate modules, and the cells in (c) determine the layout for spreadsheet results.

can manually enter a list using Python notation; for example, `[30, 36, 45, 75]`. To set the direction in which to explore a given parameter, simply select the radio button in the column for the specified direction. Note that choosing the final column disables exploration for that parameters.

In both the `Set Methods` and `Annotated Pipeline` panels, you may see numbered red circles. See Figure 7.4 for an example of this behavior. These circles appear when there is more than module of a given type in a workflow. For each type satisfying this criteria, the instances are numbered and displayed so that you can identify which part of the pipeline a module in the `Set Methods` panel corresponds to.

To run a parameter exploration, click the `Execute` button in the VisTrails toolbar or select `Execute Parameter Exploration` from `Run` menu.

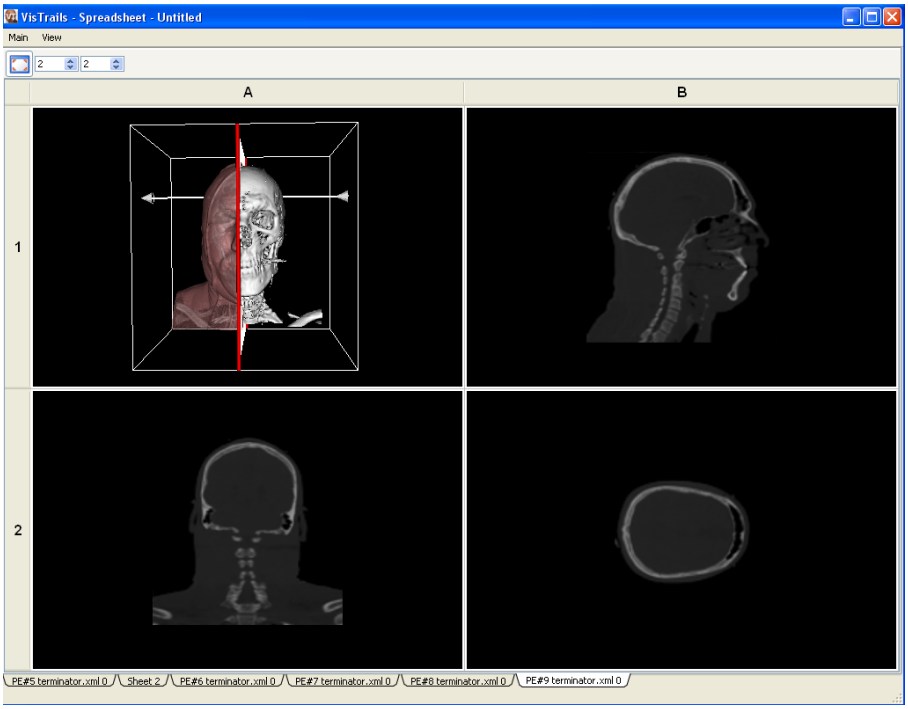


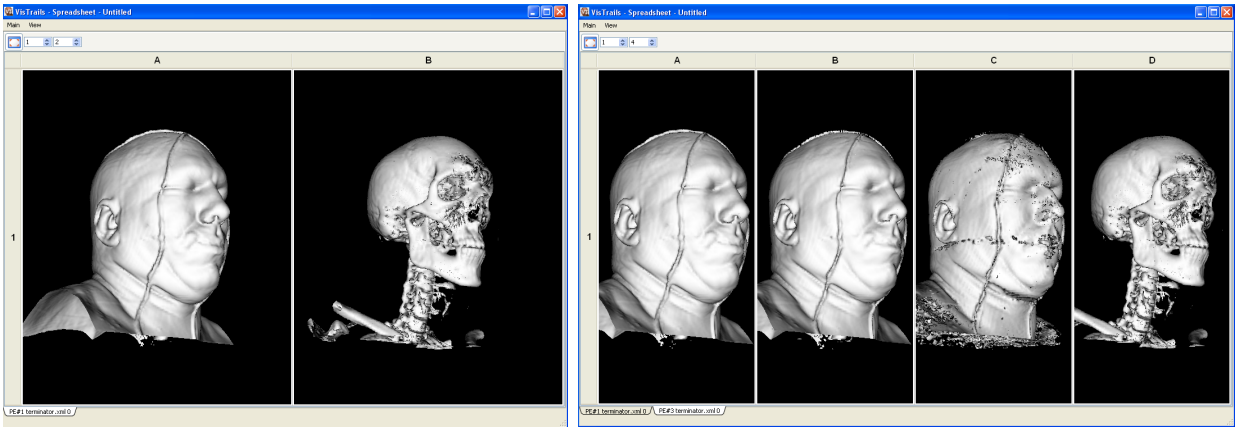
Figure 7.5: Results of the Virtual Cell arrangement.

7.2 Spreadsheet Integration

As stated earlier, the Spreadsheet provides integrated support for parameter explorations. Each of the directions of exploration corresponds to a visual dimension in the spreadsheet: the 'x' direction corresponds to columns; the 'y' direction to rows; the 'z' direction to sheets; and time to animations. However, when a workflow already outputs to more than one cell, you can layout the group of cells as it will be replicated during the exploration. For example, given a workflow with 2 output cells and an exploration for three parameter values in the 'x' direction, the resulting spreadsheet could be 1×6 or 2×3. The `Spreadsheet Virtual Cell` panel controls the layout of the pattern. Drag and drop cells to position them. See Figures 7.4(c) and 7.5 for an example.

7.3 Examples

To demonstrate the power of parameter exploration, we conclude this chapter with a couple of detailed examples. For these examples, make sure that the Spreadsheet is installed and enabled (see Chapter 5).



(a) (b)

Figure 7.6: Parameter Exploration of (a) two and (b) four isovalues as displayed in the Spreadsheet

7.3.1 Isosurfaces

One important task in visualization is finding isosurfaces that capture interesting features. In the this example, we'll look at determining the interfaces between different types of tissue captured by CT scans. To begin, load the "terminator.vt" vistrail, select the "isosurface" node in the version tree, and switch to parameter exploration. From the `Set Methods` panel, click and drag the `SetValue` method of the `vtkContourFilter` module to the center view.

We'd like to compare different values for the isosurfaces so change the start and end values to "30" and "75". Since side-by-side visualization will look better on most monitors, select the radio button below the 'x' dimension control, and increase the value of the control to 2 (see Figure 7.2). Execute the exploration and switch to the Spreadsheet to view the results. They should match Figure 7.6(a).

While these two isovalues show interesting features, we may wish to examine other intermediate isosurfaces. To do so, switch back to the main VisTrails window and increase the number of results to generate in the 'x' direction to four. VisTrails will calculate the intermediate values via linear interpolation, and your execution of this new exploration should match Figure 7.6(b).

7.3.2 Resampling

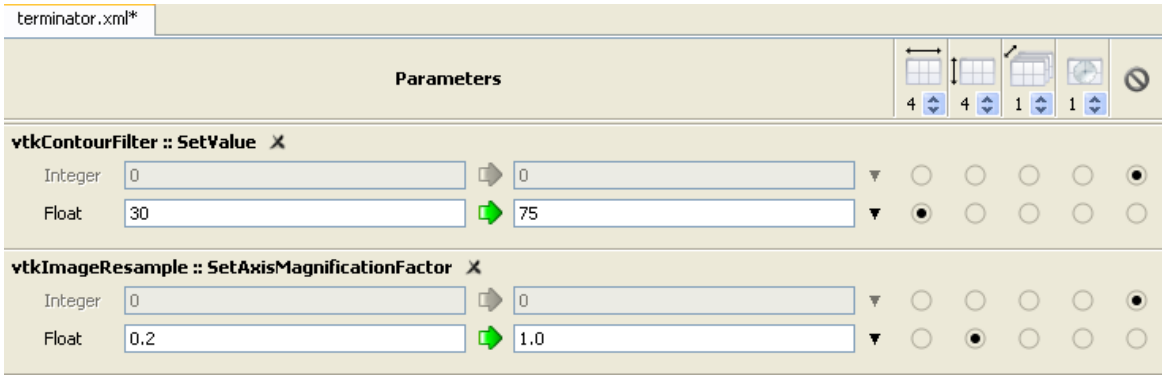
The next example uses both `X` and `Y dimension` combo boxes to change the values of two parameters at the same time in the same spreadsheet. For this we will add the module `vtkImageResample` to the pipeline insert it between `vtkStructuredPointsReader` and `vtkContourFilter` and connect the output of the reader to input of the resampler and the output of the resampler to the input of the contour filter. Finally, select the `vtkImageResample` module and add the `SetAxisMagnificationFactor` method with parameter values 0 and 0.2. See Chapter 3 for

reminders on how to accomplish these tasks.

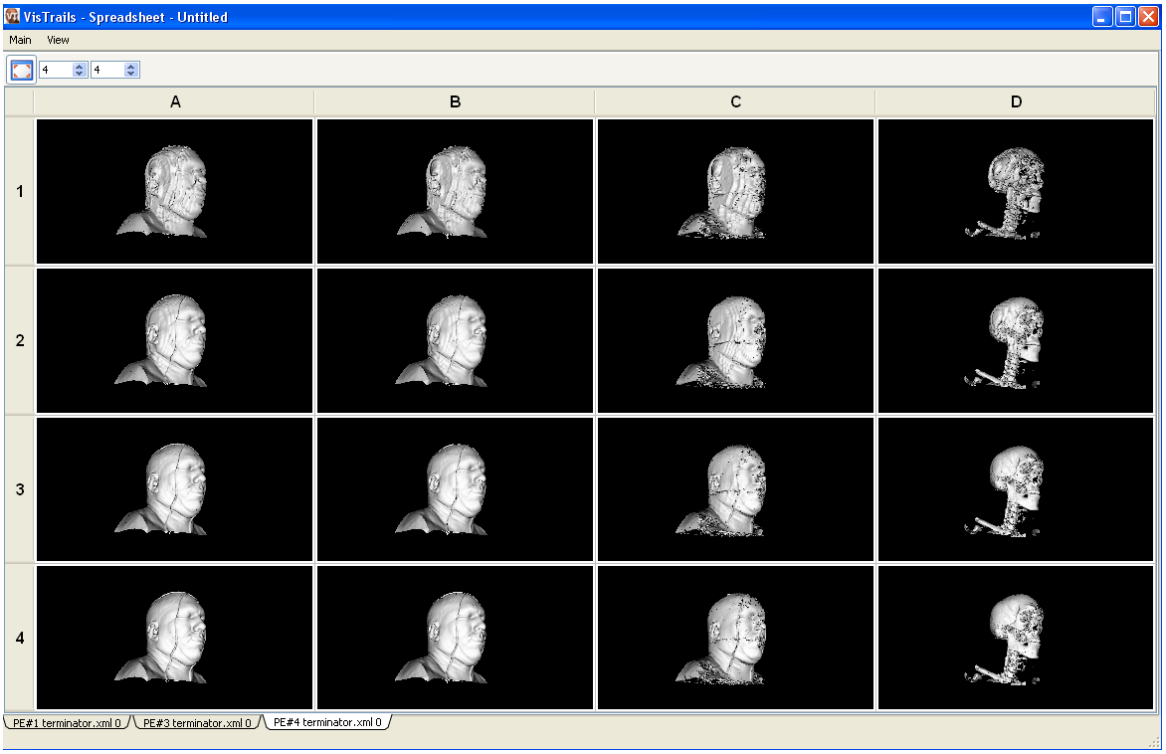
After modifying the workflow, switch back to the `Exploration` view, and add drag the `SetValue` and `SetAxisMagnificationFactor` methods to the exploration table. Set the isovalues as in the previous example, but set the range of the second parameter of the magnification factor to start at at 0.2 and end at 1.0. Also, set the magnification factor to vary over the 'y' direction. Finally, set the exploration to generate 16 results, four in the 'x' direction, and four in the 'y' direction. Your exploration setup should match Figure 7.7(a), and after executing, you should see a result that resembles Figure 7.7(b). Notice that the isosurface changes from left to right while the images have less artifacts as the magnification factor approaches 1.0 from top to bottom.

7.3.3 Animation

To create an animation, we'll use the same "terminator.vt" example (make sure that you have the "Isosurface" version selected). Follow the same steps as in the Isosurface example, but this time, use the range from 30 to 80 (again using linear interpolation) and select time as the dimension to explore, setting the number of results to generate to 7. See Figure 7.8(a) to check your settings. After executing, the Spreadsheet will show a *single* cell, but if you select that cell, you will be able to click the `Play` button in the toolbar. You should see an animation where each frame is the result of choosing a different isovalue. A sample frame is displayed in Figure 7.8(b).

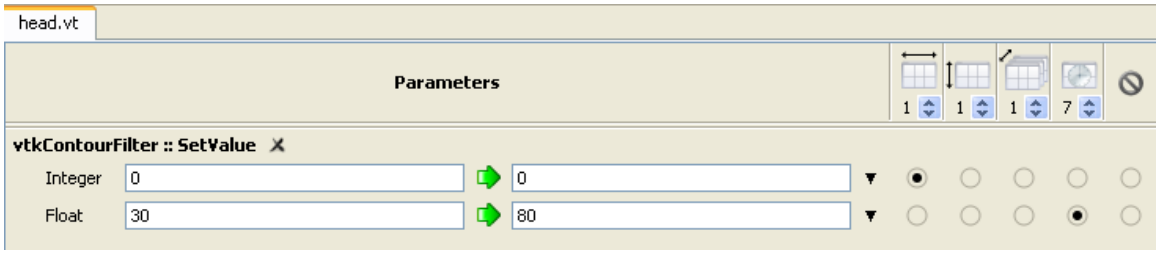


(a) Setting up parameter exploration

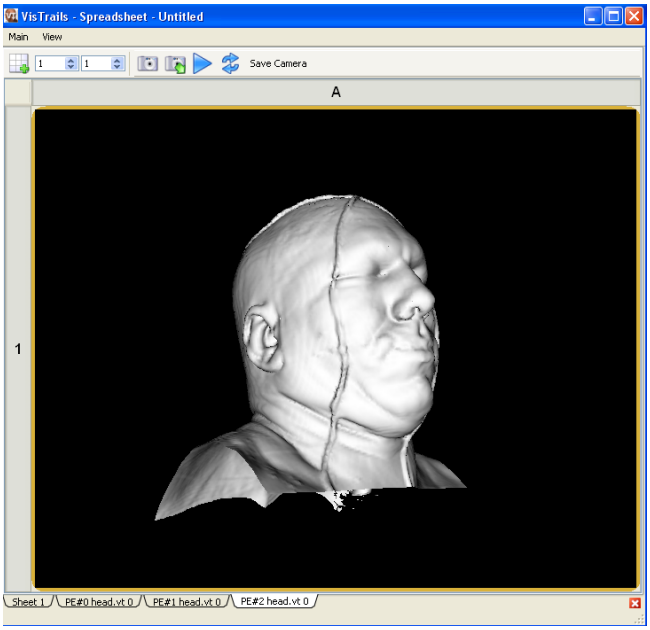


(b) Resulting spreadsheet

Figure 7.7: Using parameter exploration with two parameters



(a) Setting up parameter exploration



(b) One frame from the resulting animation

Figure 7.8: Animations with parameter exploration

Chapter 8

Using Bookmarks

Chapter 9

Connecting to a Database

As an environment for collaborative scientific exploration, VisTrails supports both stand-alone, file-based storage and relational storage of vistrails. With a relational database supporting VisTrails, users can easily collaborate on projects without copying files back and forth. At the same time, a user who wishes to work without being connected to a database can save their work locally to files. Finally, VisTrails can import and export to both types of storage so a user can import a vistrail from the database, save it locally as a file on their laptop, make and save changes, and export those changes back to the database. Remember that because the complete workflow evolution is always saved, users will never step on each others' feet.

By default, VisTrails works with local files stored in the ".vt" format (essentially compressed XML). You can save the uncompressed XML by saving the file with a ".xml" extension. When saving a vistrail, the system displays a standard save dialog. These files have a version associated with them so when the schema for these files may change, VisTrails will be able import older versions. The current version of the XML schema can be found in the distribution at:

```
db/versions/v0_8_0/schemas/xml/vistrail.xsd
```

where `v_8_0` is the current version.

9.1 Setup

As described earlier, VisTrails supports relational storage as well as file-based storage. Currently, VisTrails has been tested with the MySQL system, but we plan to support most standard relational systems.

9.1.1 Setting up the database

Before using VisTrails with a relational system, you must have a database system installed and have access to create, access, and modify that database. If you are planning to deploy for

institution-wide access, you should consult your system administrator to determine the correct configuration. The schema for relational VisTrails system can be found in the distribution at

`db/versions/v0_8_0/schemas/sql/vistrails.sql`

where `v0_8_0` is the current version. This schema contains a sequence of SQL commands that define the tables needed for storing vistrails.

After you or someone else has created the database for the vistrails, you will need the following information:

1. *hostname*: the name or IP of the machine that stores the database (`localhost` if it is your own machine)
2. *port*: the port that you connect to the database on
3. *user*: the username that should be used to access and modify the vistrails database
4. *password*: the password for the account corresponding to the given user
5. *database name*: the name of the database where the vistrails are to be stored.

9.1.2 Setting up VisTrails

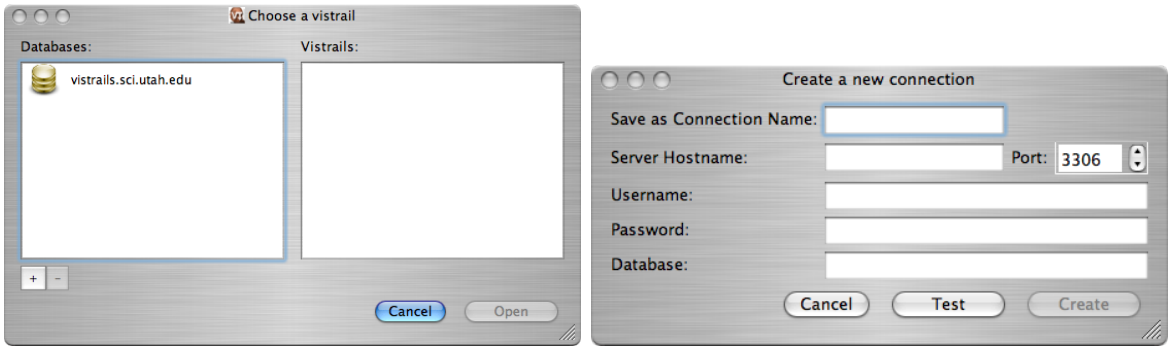
If you would are planning to use the database for most of your work, you can switch VisTrails to open vistrails from the database by default. To do so, open the `Preferences` window, and check the "Read/Write to database by default" box in the `General Configuration` tab. You can switch the default back to a file-based interaction by unchecking this box.

9.2 Opening from a database

If you have set VisTrails to use a relational database by default (see Section 9.1.2), you will be able to open a vistrail by either selecting `Open` from the `File` menu or clicking the button with the same name on the toolbar. Otherwise, you should choose the `Import` item from the `File` menu. You should see a dialog like the one pictured in Figure 9.1(a).

If you have already connected to databases using VisTrails, you should see a list of them in the left column of the dialog. If not, you will need to add one. To do so, click the plus icon in the lower-left corner. This will bring up a dialog like that shown in Figure 9.1(b), and to setup a connection, you will need the database connection information outlined in Section 9.1.1. After filling in that information, you can test the connection by clicking the `Test` button. If the test succeeds, click the `Create` button to add the database to the available sources for vistrails.

The database you wish to use should now be listed in the left column. Clicking on that row will query the database for a list of vistrails available from the database and display them in



(a) (b)

Figure 9.1: (a) Opening a vistrail from the database, (b) Creating a new database connection

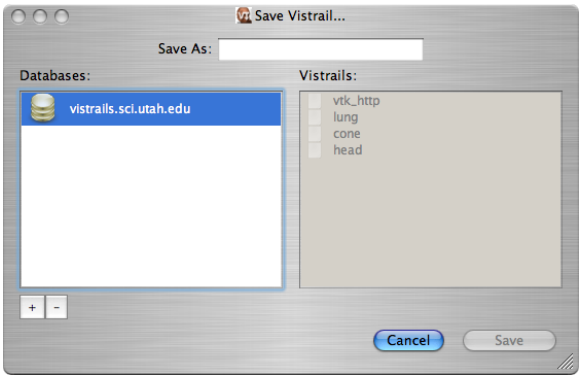


Figure 9.2: Saving a vistrail to the databse

the right column. To open a vistrail, select the desired vistrail and click the `Open` button or simply double-click the vistrail. When the vistrail has loaded, you will be able to interact with it in exactly the same way as a vistrail loaded from an XML file.

9.3 Saving to a database

If you opened a vistrail from the database, the default save action will be to save that vistrail back to the database. There will be no dialogs displayed—the database the vistrail was loaded from will be automatically updated.

If you opened the vistrail from a file, you will need to select either `Save As..` or `Export` from the `File` menu, depending on whether VisTrails uses the database by default (see Section 9.1.2. You will be shown a dialog similar to the one in Figure 9.2. As discussed in Secion 9.2, you can create a new connection to the database or use an existing one. Note that the name of the vistrail must differ from those already stored on the databse, and clicking the `Save` button will

persist the changes to the database.

9.4 Known Issues

Currently, saving a vistrail to the database will *overwrite* the vistrail currently stored on the database. However, we plan to add synchronization soon so that all explorations are captured. Thus, be aware that if two users have the same vistrail loaded from the database at the same time, and both users save their changes, only the second user's changes will be captured.

Chapter 10

Using Analogies to Update Workflows

Chapter 11

Writing VisTrails Packages

11.1 Introduction

VisTrails provides a plugin infrastructure to integrate user-defined functions and libraries. Specifically, users can incorporate their own visualization and simulation codes into pipelines by defining custom modules. These modules are bundled in what we call *packages*. A VisTrails package is simply a collection of Python classes stored in one or more files, respecting some conventions that will be described shortly. Each of these classes will represent a new module. In this chapter, we will build progressively more complicated modules. Note that even though each section introduces a specific large feature of the VisTrails package mechanism, new small features are highlighted and explained as we go along. Because of this, we recommend at least skimming through the entire chapter at least once.

Let us start with a minimal complete example of a very simple calculator:

```
1  import core.modules.module_registry
2  from core.modules.vistrails_module import Module, ModuleError

3  version = "0.9.0"
4  name = "PythonCalc"
5  identifier = "edu.utah.sci.vistrails.pythoncalc"

6  class PythonCalc(Module):
7      """PythonCalc is a module that performs simple arithmetic operations on
8  its inputs."""

9      def compute(self):
10         v1 = self.getInputFromPort("value1")
11         v2 = self.getInputFromPort("value2")
12         result = self.op(v1, v2)
13         self.setResult("value", result)
```

```
    def op(self, v1, v2):
        op = self.getInputFromPort("op")
        if op == '+': return v1 + v2
        elif op == '-': return v1 - v2
        elif op == '*': return v1 * v2
        elif op == '/': return v1 / v2
        else: raise ModuleError(self, "unrecognized operation: '%s'" % op)

##############################################################################

def initialize(*args, **keywords):

    # We'll first create a local alias for the module registry so that
    # we can refer to it in a shorter way.
    reg = core.modules.module_registry.registry

    reg.addModule(PythonCalc)
    reg.addInputPort(PythonCalc, "value1",
                     (core.modules.basic_modules.Float, 'the first argument'))
    reg.addInputPort(PythonCalc, "value2",
                     (core.modules.basic_modules.Float, 'the second argument'))
    reg.addInputPort(PythonCalc, "op",
                     (core.modules.basic_modules.String, 'the operation'))
    reg.addOutputPort(PythonCalc, "value",
                      (core.modules.basic_modules.Float, 'the result'))
```

To try this out in VisTrails, save the file above in your packages directory as `pythoncalc.py`. Then, click on `Edit` and then `Preferences`. A dialog similar to what is shown in Figure 11.1 should appear. Select the `pythonCalc` package, then click on `Enable...` This should move the package to the `Enabled packages` list. Close the dialog. The package and module should now be visible in the VisTrails builder.

Now create a workflow similar to what is shown in Figure 11.2. When executed, this workflow will print the following on your terminal:

```
7.0
```

Let's now examine how this works. The first two lines simply import required components. Then, we have three lines that give VisTrails meta-information about the package. `version` is simply information about the package version. This might be tied to the underlying library or not. The only recommended guideline is that compatibility is not broken across minor releases, but this is not enforced in any way. `name` is a human-readable name for the package.

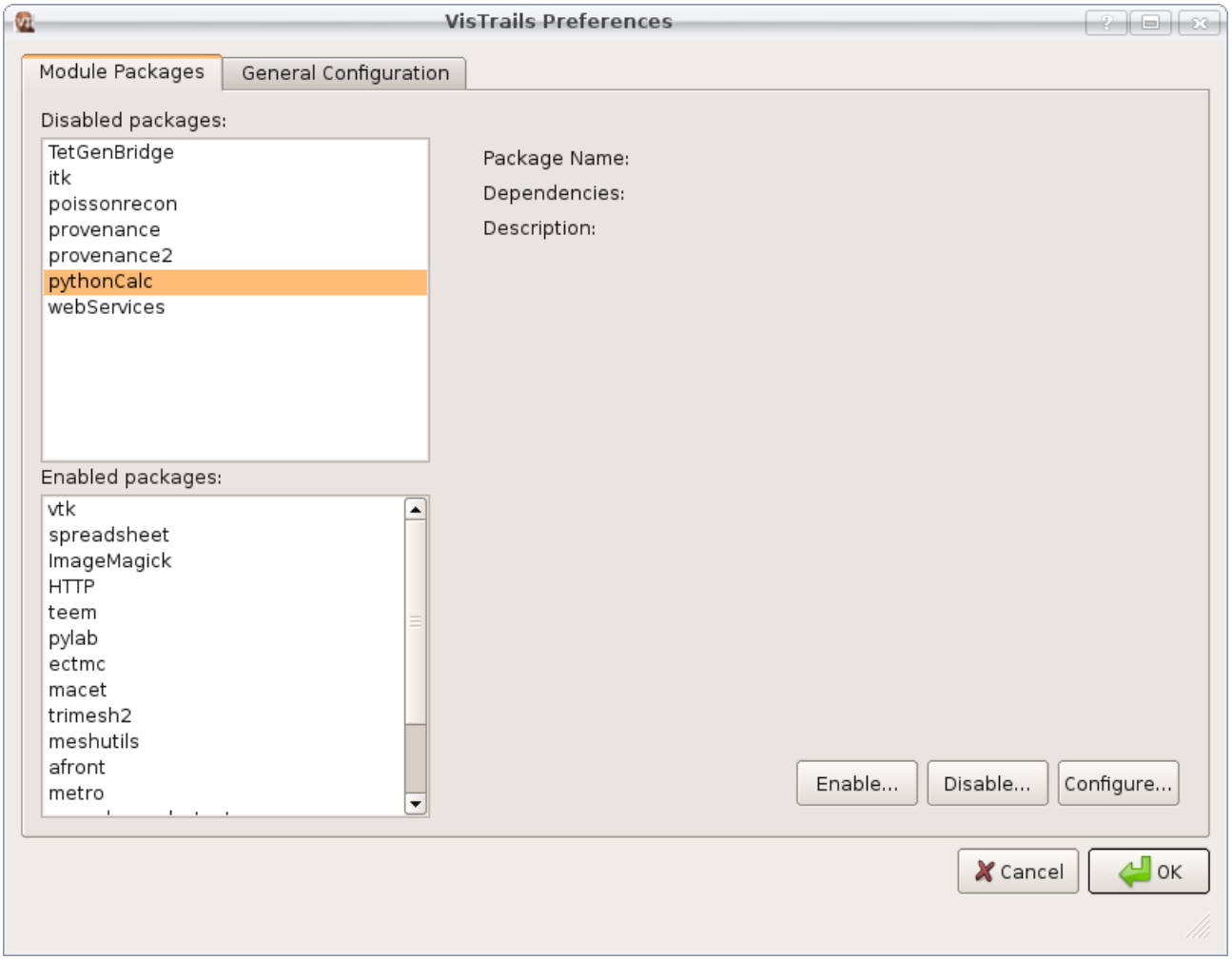


Figure 11.1: All available packages can be enabled and disabled with the VisTrails preferences dialog.

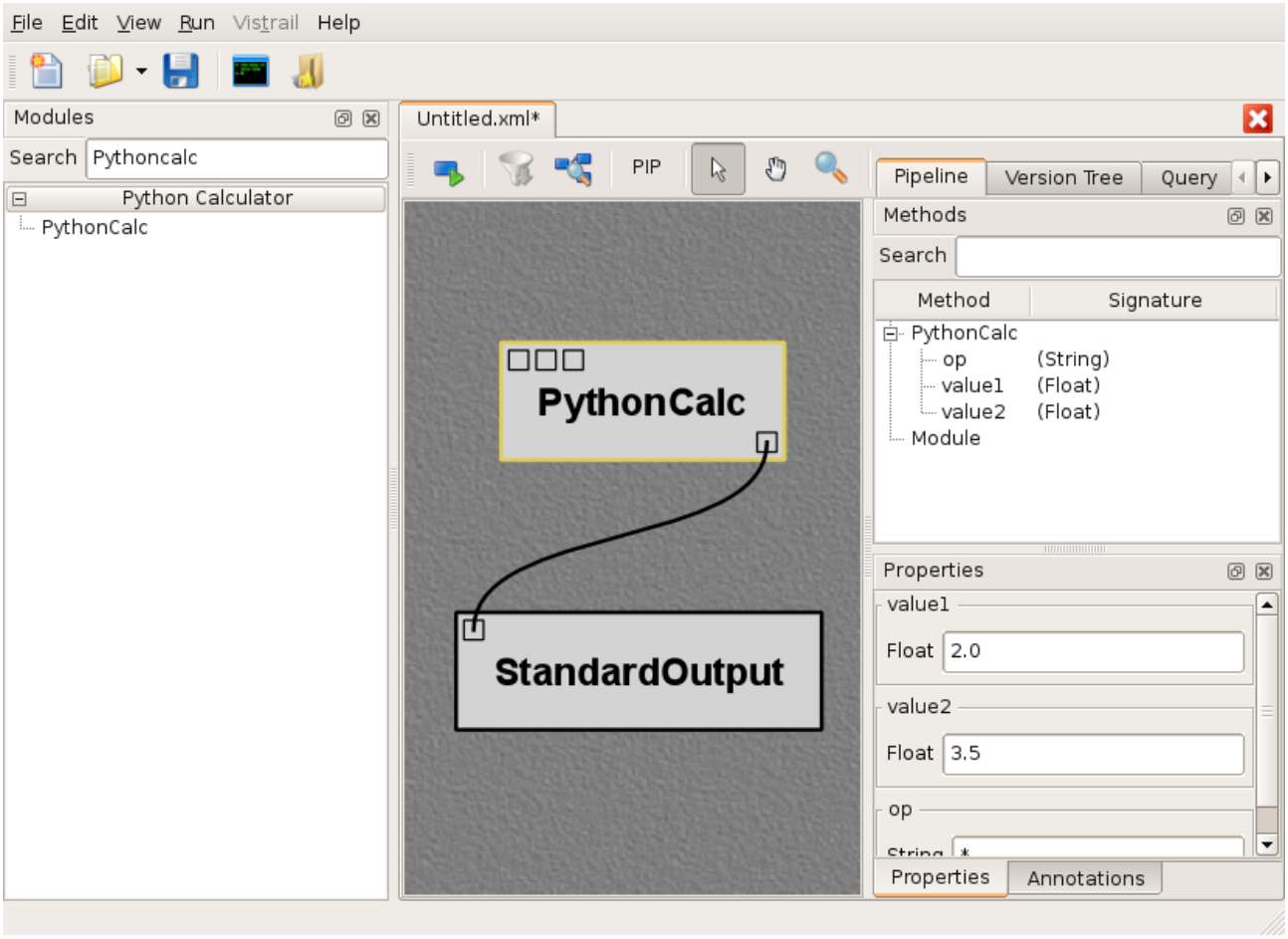


Figure 11.2: A simple workflow that uses `PythonCalc`, a user-defined module.

Choosing a good identifier The most important meta-data, however, is the package *identifier*, stored in `identifier`. This is a string that must be globally unique across all packages, not only in your system, but in any possible system. We recommend using an identifier similar to Java's package identifiers. These look essentially like regular DNS names, but the word order is reversed. This makes sorting on the strings a lot more meaningful. You should generally go for `institution.project.creatorusername` for a package related to a certain project from some institution, and `institution.creatorname` for a personally developed package. If you are wrapping third-party functionality, *do not* use their institution's DNS, use your own. The rationale for this is that the third party itself might decide to create their VisTrails package, and you do not want to introduce conflicts.

Line 6 is where we actually start defining a new module. Every VisTrails module corresponds to a Python class that ultimately derives from `Module`, a class defined in `core.modules.vistrails_module`. A new module must implement a `compute` method that takes no extra parameters, such as on Line 9. This method represents the actual computation that happens in a module. This computation typically involves getting the necessary input and generating the output. We will now see how that works.

Line 10 shows how to extract input from a port. Specifically, we're getting the values passed to input ports `value1` and `value2`. We then perform some operation with these values, and need to report the output on an output port, so that it is available for downstream modules. This is done on Line 13, where the result is set to port `value`.

Let us now look more carefully at the remainder of the class definition. Notice that developers are allowed to define extra helper methods (Line 14). These methods can naturally use the ports API. The other important feature of `op(self, v1, v2)` is *error checking*. `PythonCalc` requires a string that represents the operation to be performed with the two numbers. If the string is invalid, it signals an error, by simply raising a Python exception `ModuleError` that is provided in `core.modules.vistrails_module`. This exception expects two parameters: the module that generated the exception (typically `self`) and a string describing the error, which will be presented to the user.

That is all that it takes in terms of module behavior. The rest of the code is meant to interact with VisTrails, and let the system know about the modules and ports being exposed. To do that, users must provide an function in the main body of the package file (the function starting on Line 22). The first thing is usually to register the module itself, such as on Line 26. Then, we need to tell VisTrails about the input and output ports we want to expose. Input ports are set with the `addInputPort` method in the registry, and output ports, with `addOutputPort`. These calls take three parameters. The first parameter is the module you're adding a new port to. The second one is simply the name of the port, and the third one is a description of the parameter. In simple cases, this is just a pair, where the first element is a VisTrails module representing the module type being passed, and the second element is a string describing it. Later, we will see how to pass more complicated data types. Notice that the types being used are VisTrails modules (Line 30), and not Python types.

This is it — you have successfully created a new package and modules. From now on, we will look at more complicated examples, and more advanced features of the package mechanism.

11.2 Wrapping Command-line tools

Many existing programs are readily available through a command-line interface. Also, many existing workflows are usually first implemented through scripts, which work primarily with command-line tools. This section describes how to wrap command-line applications so they can be used with VisTrails. We will use as a running example the `afront` package, which wraps `afront`, a program to generate 3D triangle meshes[1]. This package is available by default on a VisTrails install. We will wrap the basic functionality in three different modules: `Afront`, `MeshQualityHistogram` and `AfrontIso`.

11.2.1 Class Mixins

Each of these modules will be implemented by a Python class, and they will all invoke the `afront` binary. `Afront` is the base execution module, and `AfrontIso` requires extra parameters on top of the original ones. Because of this, we will implement `AfrontIso` as a subclass of `Afront`. `MeshQualityHistogram`, however, requires entirely different parameters, and so should not be a subclass of `Afront`. Our package will look something like this, then:

```
1  from core.modules.vistrails_module import Module
2  ... # other import statements

3  name = "Afront"
4  version = "0.1.0"
5  identifier = "edu.utah.sci.cscheid"

6  class Afront(Module):
7      def compute(self):
8          ... # invokes afront

9  class AfrontIso(Afront):
10     def compute(self):
11         ... # invokes afront with additional parameters

12 class MeshQualityHistogram(Module):
13     def compute(self):
14         ... # invokes afront with completely different parameters
```

[1] `Afront` is available at http://afront.sourceforge.net

```
15 def initialize():
16     ...
```

It should be clear that all three modules share some functionality (invoking `afront`), but not all. We would like to avoid duplicate code, but there is not a single class where we can implement the base code. The solution is to create a *mixin class*, where we implement the necessary functionality, and then inherit from both classes. In the following snippets, we will highlight the changes in the code.

```
1  from core.modules.vistrails_module import Module, ModuleError
2  from core.system import list2cmdline
3  import os

4  class AfrontRun(object):
5      _debug = False
6      def run(self, args):
7          cmd = ['afront', '-nogui'] + args
8          cmdline = list2cmdline(cmd)
9          if self._debug:
10             print cmdline
11         os.system(cmdline)
12         if result != 0:
13             raise ModuleError(self, "Execution failed")

14 class Afront(Module, AfrontRun):
15     ...

16 class MeshQualityHistogram(Module, AfrontRun):
17     ...
```

Now every module in the `afront` package has access to `run()`. The other new feature in this snippet is `list2cmdline`, which turns a list of strings into a command line. It does this in a careful way (protecting arguments with spaces, for example). Notice that we use a call to a shell (`os.system()`) to invoke `afront`. This is frequently the easiest way to get third-party functionality into VisTrails.

11.2.2 Package Configuration

There are two obvious shortcomings to the way `run()` is implemented. First, the code assumes `afront` is available in the system path, which might not be true in practice. Second, the debugging variable is inaccessible to the interface, where it would be really handy. VisTrails

provides a way to *configure* a package through a dialog. It is very simple to provide your own configuration: just add a `configuration` attribute to your package, as follows:

```
1  from core.configuration import ConfigurationObject
2  from core.modules.vistrails_module import Module, ModuleError
3  from core.system import list2cmdline
4  import os

5  configuration = ConfigurationObject(path=(None, str),
6                                      debug=False)

7  class AfrontRun(object):

8      def run(self, args):
9          if configuration.check('path'): # there's a set directory
10             afront_cmd = configuration.path + '/afront'
11         else: # Assume afront is on path
12             afront_cmd = 'afront'
13         cmd = [afront_cmd, '-nogui'] + args
14         cmdline = list2cmdline(cmd)
15         if configuration.debug:
16             print cmdline
17         ...
18 ...
```

Let us first look at how to specify configuration options. Named arguments to the `ConfigurationObject` constructor become attributes in the object. If the attribute has a default value, simply pass it to the constructor. If the attribute should by default be unset, pass the constructor a pair whose first element is `None`, and second element is the *type* of the expected parameter. Currently, the valid types are `bool`, `int`, `float` and `str`.

To use the configuration object in your code, you can simply access the attributes. (as on line 15). This is fine when there is a default value set for the attribute. In the case of `path`, however, the absence of a value is encoded by a tuple `(None, str)`, so using it directly is inconvenient. That is where the `check()` method comes in (line 9). It returns `False` if there is no set value, and returns the value otherwise.

The real advantage of using a configuration object is that the values can be changed through a GUI, and they are persistent across VisTrails sessions. To configure a package, open the `Preferences` menu (click on `Edit` and `Preferences`). Then, select the package you want to configure by clicking on it (a package must be enabled to be configurable). If the `Configure` button is disabled, it means the package does not have a configuration object. When you do click on it, a dialog like the one in Figure 11.3 will appear.

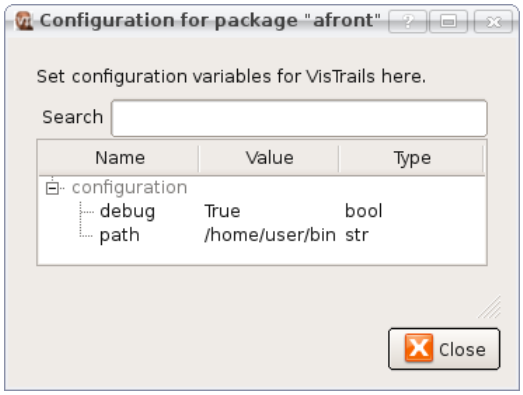


Figure 11.3: Configuration window for a package that provides a configuration object.

To edit a particular field, double-click on it, and change the value. The values set on that window are persistent across VisTrails sessions, being saved on a per-user basis.

11.2.3 Temporary File Management

Command-line programs typically generate files as outputs. On complicated pipelines, many files get created and passed to other modules. To facilitate the use of files as communication objects, VisTrails provides basic infrastructure for temporary file management. This way, package developers do not have to worry about file ownership and lifetimes.

To use this infrastructure, it must be possible to tell the program being called which filename to use as output. VisTrails can accomodate particular some filename requirements (in particular, specific filename extensions might be important in Windows environments, and these can be set), but it must be possible to direct the output to a certain filename.

We will use `Afront`'s `execute()` method to illustrate the feature.

```
...
class Afront(Module, AfrontRun):

    def compute(self):
        o = self.interpreter.filePool.create_file(suffix='.m')
        args = []
        if not self.hasInputFromPort("file"):
            raise ModuleError(self, "Needs input file")
        args.append(self.getInputFromPort("file").name)
        if self.hasInputFromPort("rho"):
            args.append("-rho")
            args.append(str(self.getInputFromPort("rho")))
        if self.hasInputFromPort("eta"):
```

```
13              args.append("-reduction")
14              args.append(str(self.getInputFromPort("eta")))
15          args.append("-outname")
16          args.append(o.name)
17          args.append("-tri")
18          self.run(args)
19          self.setResult("output", o)
20  ...
```

Line 4 shows how to create a temporary file during the execution of a pipeline. There are a few new things happening, so let us look at them one at a time. Every module holds a reference to the current *interpreter*, the object responsible for orchestrating the execution of a pipeline. This object has a `filePool`, which is what we will use to create a pipeline, through the `create_file` method. This method takes optionally a named parameter `suffix`, which forces the temporary file that will be created to have the right extension.

The file pool returns an instance of `basic_modules.File`, a module that is provided by the basic VisTrails packages. There are two important things you should know about `File`. First, it has a `name` attribute that stores the name of the file it represents. In this case, it is the name of the recently-created temporary file. This allows you to safely use this file when calling a shell, as can be seen on Line 16. The other important feature is that it can be passed directly to an output port, so that this file can be used by subsequent modules. This is shown on Line 19.

Accomodating badly-designed programs Even though it is considered bad design to not allow the specification of output filename, there exist programs that do so. In this case, a possible workaround is to execute the command-line tool, and move the generated file to the name given by VisTrails.

11.3 Interpackage Dependencies

When creating more sophisticated VisTrails packages, you might want to create a new module that requires a module *from another package*. For example, using modules from different packages as input ports, or even subclassing modules from other packages require managing of interpackage dependencies. VisTrails needs to know about these, so packages can be initialized in the correct order. To specify these dependencies, you should add a python callable named `dependencies` to your package. We will start with a simple situation where new input and output ports are added to `PythonCalc`, one that uses modules in a different package:

```
1  ...
```

```
2       class PythonCalc(Module):

3           def compute(self):
4               ...

5               # Computes values for SciPy matrices as well
6               if self.hasInputFromPort("mat1"):
7                   m1 = self.getInputFromPort("mat1")
8                   m2 = self.getInputFromPort("mat2")
9                   self.setResult("mat_value", self.mat_op(m1, m2))

10          def mat_op(self, v1, v2):
11              reg = core.modules_module_registry.registry
12              Matrix = reg.getDescriptorByName('Matrix').module
13              result = Matrix()
14              result.matrix = self.op(m1.matrix, m2.matrix)
15              return result

16  ...

17  def dependencies():
18      return ["SciPy"]

19  def initialize():
20      ...
21      Matrix = reg.getDescriptorByName('Matrix').module

22      reg.addInputPort(PythonCalc, "mat1", (Matrix, 'first matrix'))
23      reg.addInputPort(PythonCalc, "mat2", (Matrix, second matrix'))
24      reg.addOutputPort(PythonCalc, "value", (Matrix, matrix result))
```

Let us look at the first new call on Line 6. `hasInputFromPort` can be used for *optional ports*: ports that only used when there are connections attached to it. If there is one such connection, we get the inputs from both ports, and call `mat_op`. This method will perform the necessary conversion to native `SciPy` matrices before calling `op`.

11.4 Requirements

11.5 Interaction with Caching

11.6 Advanced: Wrapping a big API

In the future, there will be a walkthrough of the VTK package wrapping mechanism.

Chapter 12

Advanced Topics: Module Execution and Caching

Chapter 13

Example: Web Services

In this chapter, you will learn how to invoke web services from within VisTrails workflows. We will build a simple workflow that invokes a few web services and publishes a web page with the results. The web services we will use are provided by The Chemical Informatics and Cyberinfrastructure Collaboratory (CICC) at Indiana University and can be found at http://www.chembiogrid.org/products/index.html.

13.1 Enabling the webServices Package

The first thing we need to do after starting VisTrails is to enable the "webServices" package on the `Preferences` pane.

Open the `Preferences` panel (On Windows and Linux, it is located under the `Edit` menu, and on the Mac, it is under the `VisTrails` menu), and select the tab named `Module Packages` (see Figure 13.1). On the Disabled packages list, select `webServices` and click on `Enable`.

Now select `webServices` on the Enabled packages list and click on `Configure`. A new window will appear and you will be able to add a ';'-separated list of web services urls. Select `wsdlList` and click on the `Value` field. You can type the web services urls you want. For our example, we need the following two urls:

```
http://rguha.ath.cx:8080/pws/services/Structure?wsdl;
http://rguha.ath.cx:8080/cdkws/services/StructureDiagram?wsdl
```

Click on `Close`. Then you are required to disable and enable the package again so the urls can be loaded. After that, close the `Preferences` window.

13.2 Creating a new vistrail

After configuring the `webServices` package properly, you will see that there will be a tab `webServices` in your `Modules` panel (see Figure 13.2(a)). The *webService* package will generate

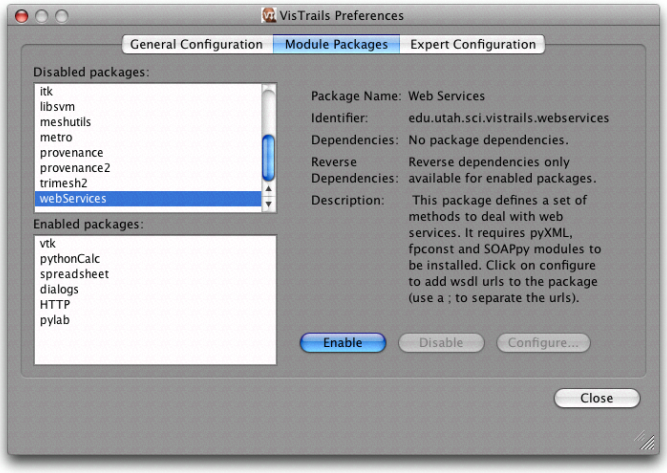


Figure 13.1: `webServices` package information shown on VisTrails Preferences Pane.

a module for each published method in a web service. If you do not already have a new vistrails open, that is the right time to create a new one.

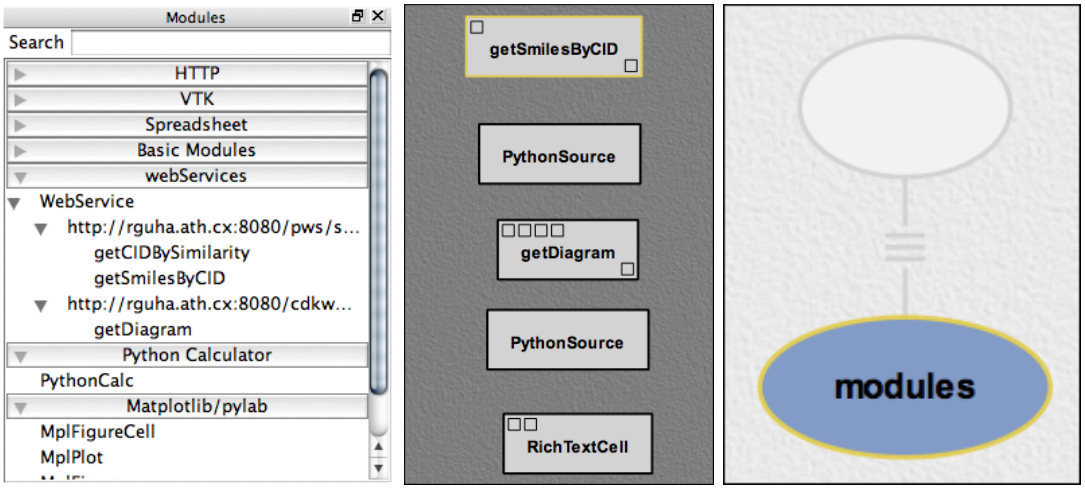
13.3 Adding modules to the workflow

At this point, we should start adding modules to our workflow.

If you have the web services list visible in your Modules panel, click on "getSmilesByCID" and drag it to the `Pipeline` view area. Otherwise, use the search capability: in the `Search` field of the `Modules` panel (the leftmost pane of the Builder Window), type in "getSmilesByCID". You will notice that a module under the webServices branch will be selected. Now you can add it by clicking-and-dragging it over the `Pipeline` view area represented by the darker grey canvas on the Builder Window. This module gets the SMILES [1] corresponding to a compound ID. We need to add more modules that will process the output provided by the `getSmilesByCID` module, including another web service module that will obtain the 2D diagram of the compound. In the same way described above, add the following modules (the number in parenthesis represents the number of modules you should add):

- `PythonSource`(2)
- `getDiagram`(1)
- `RichTextCell`(1)

[1] Simplified Molecular Input Line Entry System. Specification for unambiguously describing the structure of chemical molecules using short ASCII strings.



(a) (b) (c)

Figure 13.2: (a) The `webServices` tab is added to the `Modules` panel after modifying vistrais startup.py file (b) All the modules necessary for building the workflow were added to the workflow (c) the version tree after tagging the current version

After adding these modules, our workflow should be similar to the one shown in Figure 13.2(b).

The modules were added to the pipeline view, but remain unconnected. This is a good point for us to save our work. First we will name this pipeline as modules. To do that, we need to switch to the version tree view by selecting the `Version Tree` tab in the top pane on the right of the buider window. In the vtinterfaceVersion Tag field type in "modules" and click on `change`. Your version tree should now look similar to the one in Figure 13.2(c).

Now we save our work by clicking on the `Save` button (the third from left to right on the Builder Toolbar) or pressing 'Ctrl+S' ('Command+S' on Mac). Give a name to your file, such as "chembiogrid_webservice.xml". Then we can go back to the pipeline view by selecting the `Pipeline` tab.

13.4 Module customization and parameterization

Each module box has a set of input ports, located in the upper-left hand corner of the box, and a set of output ports, located in its lower-right hand corner. They will be used to *pipe* data between modules. You may have noticed that the `PythonSource` module does not contain any input or output port. This module is designed to contain any piece of Python code. So we, as pipeline builders, must define the input and the output ports and include the piece of code to manipulate the inputs and generate outputs. We are going to use this module to process data between the web services and to create our html page.

First, open the configuration window of the top most `PythonSource` module by clicking on the arrow on the top-right corner of the module box and select `Edit Configuration` (please refer to Chapter 3 for more information on `PythonSource`).

Add one input port, "data" of type `String` (the same output type of `getSmilesByCID` module) and add an output port called "smiles" also of type `String` (the same input type of `getDiagram`).

Now type the following code in the text area and click "OK" when you are done:

```
smiles = data[0]
```

The `getSmilesByCID` module returns an array of strings encoded in a String object. The `getDiagram` module, on the other hand, expects a single "smiles". So we need to extract only one element of the array and pipe it through the `getDiagram` module (later you can add another output port and pipe the other smiles to another `getDiagram` module.

Now we will customize the other `PythonSource`. Open its `Configuration Window` and add an input port, "diagram" of type `String` (the same output type of the `getDiagram` module). Also, add an output port, htmlFile of type `File` and type the following piece of code in the text area:

```
import base64
image = base64.decodestring(diagram)
f = self.interpreter.filePool.create_file(".jpg")
my_file = open(str(f.name), 'wb')
my_file.write(image)
my_file.close()
text = '<HTML><TITLE>Compound Summary</TITLE><BODY BGCOLOR="#FFFFFF">'
text += '<TABLE WIDTH="100%" BORDER="1" BGCOLOR="#FFFFFF" %CELLPADDING="4"> '
text += '<TR><TD VALIGN="TOP"><P><IMG SRC="'
text += f.name + '"></TD>'
text += '<TD>Name:<B>: Caffeine </B><BR>'
text += "A methylxanthine naturally occurring in some beverages and "
text += "also used as a pharmacological agent. Caffeine's most notable "
text += "pharmacological effect is as a central nervous system stimulant,"
text += " increasing alertness and producing agitation.</TD></TR></TABLE>"
output = self.interpreter.filePool.create_file()
my_file = open(str(output.name), 'w')
my_file.write(text)
my_file.close()
self.setResult("htmlFile",output)
```

The `PythonSource Configuration Window` should look similar to the one shown on Figure 13.3.

We also need to set a few parameters in order to `getSmilesByCID` to work properly. `getSmilesByCID` receives a compound ID. Caffeine's CID is 2519.

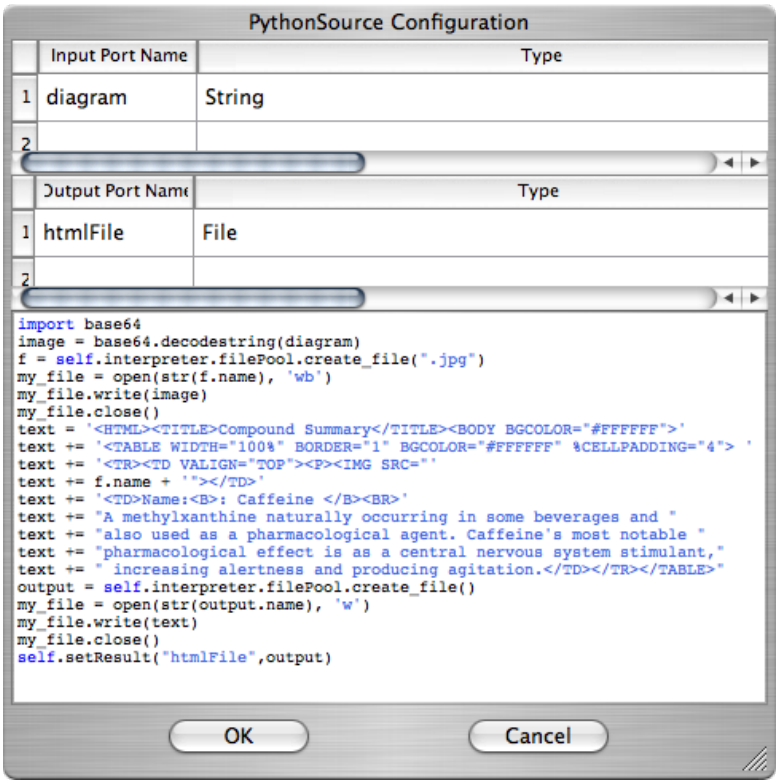


Figure 13.3: The code for decoding the image and generating the html file.

The `getDiagram` also needs parameters to be set. Provide the following values: "height": 250, "width": 250, and "scale": 1.0. Please refer to Chapter 3 for details on how to set parameters.

Let's give the name "parameters set" to this pipeline. Repeat the steps we performed above to change a version tag and save your pipeline.

13.5 Connecting modules

Now that we have all the modules necessary to process our data, we must connect them properly to fully form our processing pipeline. Each module box has a set of input ports, located in the upper-left hand corner of the box, and a set of output ports, located in its lower-right hand corner. In order to connect two modules together, click-and-drag the appropriate output box contained in the module to the module using it as its input. For modules that have more than one input/output, you can see the type of each individual port when hovering the mouse (as a tooltip). VisTrails will also snap a connection to matching ports.

So, for example, by clicking and dragging the output port of the `getSmilesByCID` module to the input port of `PythonSource` module, a connection will be made between the two modules.

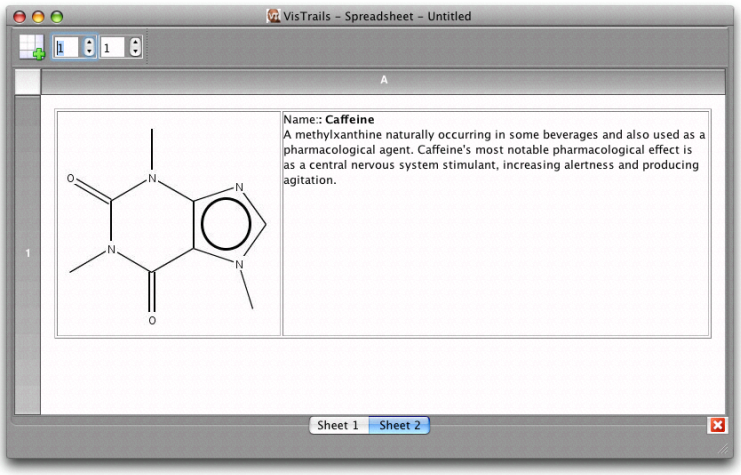


Figure 13.4: The html report generated by our pipeline.

This connection is indicated by a solid black line. Now we must continue connecting our pipeline. Add the following connections in the same way described above:

- The output port of `PythonSource` to the input port "smiles" of `getDiagram`
- The output port of `getDiagram` to the input port of `PythonSource`
- The output port of `PythonSource` to the input port "File" of `RichTextCell`

Name this version "connections" and you are ready to execute this pipeline.

13.6 Executing the workflow

The workflow is now ready to be visualized. As we have a `RichTextCell` module, pressing the `Execute current pipeline` button will send the current pipeline with the current parameters to the VisTrails Spreadsheet, resulting on an image similar to Figure 13.4.

Chapter 14

Example: ITK

14.1 Introduction to ITK

The Insight Toolkit, or ITK [1], is an open-source software system designed to support the Visible Human Project [2]. ITK is under continual development, being updated to employ cutting-edge segmentation and registration algorithms for multiple dimensions.

In order to fascilitate the implementation of processing mechanisms specific to the medical imaging community, ITK provides a robust set of general purpose image processing tools. These image processing tools are available to users through the standard ITK Filter interface [3]. Although ITK is implemented in C++, through the use of CMake [1] and CableSwig [2], the functionality of ITK is made available to languages such as TCL, Java, and Python.

14.2 Preparing ITK

At the time of this writing, the latest stable release of ITK is 3.2.0

In order to incorporate the functionality of ITK into the VisTrails system, it first must be built and installed. In the following sections, we will describe in detail the process of downloading, building, and installing ITK and all the required components needed to use it.

14.2.1 Downloading ITK

ITK can be downloaded in either source tarballs or via public CVS access to the ITK source repository. The following instructions take advantage of the CVS source repository; however, source tarballs can be downloaded from:

- http://www.itk.org/HTML/Download.php

[1] CMake cross-platform make system. http://www.cmake.org.

[2] CableSwig Interface generator. http://www.itk.org/HTML/CableSwig.html

These instructions can be found, in part, at the ITK website [3]. To use CVS, you must have a CVS client installed on your system. To download the ITK library, issue the following commands:

cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight login
password: insight

cvs -d :pserver:anonymous@www.itk.org:/cvsroot/Insight co Insight

Change directory into the newly created **Insight/Utilities** directory and issue the following command:

cvs -d :pserver:anonymous@public.kitware.com:/cvsroot/CableSwig co CableSwig

This checkout includes CableSwig in the ITK system allowing it to be built automatically during compilation of ITK itself.

14.2.2 Building the ITK Libraries

ITK requires CMake to be installed and available on your system. CMake can be found at:

- http://www.cmake.org

As of ITK version 3.2.0, CMake version 2.4.6 or greater must be used to prevent compilation errors. In order to simplify updating ITK to later versions of the software, we perform an out-of-source build. To do this, we first create a directory outside the **Insight** directory created for us during the CVS checkout process.

mkdir itk
cd itk

We now run cmake, or the GUI-based version ccmake, in this directory.

ccmake ../Insight

Note: The above command assumes that the **Insight** directory exists at the same level as the **itk** directory that we just created.

The following advanced CMake variables must be set to the appropriate values:

[3] ITK website. http://www.itk.org

CMake Variable	Value
BUILD_SHARED_LIBS	ON
INSTALL_WRAP_ITK_COMPATIBILITY	ON
ITK_CSWIG_PYTHON	OFF
ITK_USE_REVIEW	ON
USE_WRAP_ITK	ON
WRAP_ITK_PYTHON	ON

Note: Some CMake variables are only available based on the state of others. If a variable is missing from the list, set what is visible and re-configure, this will often allow you to see and set additional parameters.

After generating the appropriate files and exiting ccmake, the standard build process can be completed. In Linux variants and Mac:

make
sudo make install

On Windows, the build process is governed by the type of project or Makefile CMake generated.

Note: It is possible to use ITK without installing it. To do this, the environment variables LD_LIBRARY_PATH and PYTHONPATH must be set to the appropriate build directories:

LD_LIBRARY_PATH=/Path_To_itk/bin
PYTHONPATH=/Path_To_itk/Wrapping/WrapITK/Python

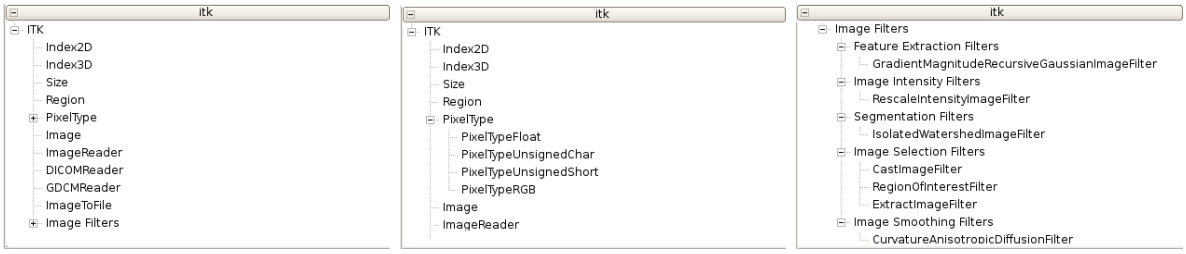
At this point, ITK is build and installed. To validate this, open a Python shell and issue the following commands:

>>> import itk
>>> itk.Image[itk.US,2]

The above commands should both complete without error. The WrapITK implementation used to wrap ITK for use in Python lazily instantiates required classes. This means that even if the import succeeds, the instantiation of the above itk.Image class may fail. This is particularly common if the envornment LD_LIBRARY_PATH is incorrectly set.

14.3 ITK and VisTrails

When built and installed with the appropriate Python bindings included, ITK can be used from VisTrails through the ITK package. ITK is a third-party package and is not included in the general VisTrails distribution. However, like many third-party packages, it is accessible from the VisTrails homepage:



(a) (b) (c)

Figure 14.1: (a) The VisTrails ITK Package Structure Overview (b) The ITK Package Supported PixelTypes (c) The ITK Package Filter Structure

http://www.vistrails.org

Please Note: The VisTrails ITK package is not a complete wrapping of all ITK functionality at the time of this writing. If you would like to contact the author regarding the wrapped functionality, please do so through the e-mail address on the VisTrails homepage.

The VisTrails ITK package is under continual development with the latest versions being announced on the vistrails homepage. After downloading the package and extracting it in accordance with the posted instructions, the following line should be added to the startup.py file:

addPackage('itk')

Upon starting VisTrails, the ITK package modules will be made available to the Builder Window.

14.3.1 ITK Package Organization

The ITK VisTrails package loosely mimics the ITK functionality hierarchy. The package's top level consists of base classes, containers, and file readers as shown in Figure 14.1. Also available at the top level is the PixelType module and the specific types used to create and execute ITK-based pipelines.

Currently, the ITK Image Filters are organized into functional groups. The five filter types, as show in Figure 14.1, are:

- Feature Extraction Filters
- Image Intensity Filters
- Segmentation Filters
- Image Selection Filters

- Image Smoothing Filters

All filter types currently have at least one representative ITK filter wrapped and usable from within the VisTrails environment.

14.3.2 Reading DICOM Volumes

ITK includes DICOM support through the GDCM libraries [4]. It is worthwhile to note at this time that these libraries are currently not a complete implementation of the DICOM standard.

DICOM volumes can be integrated into VisTrails through the use of either the GDCMReader or DICOMReader modules in the ITK package. For the rest of this example, we will use the GDCMReader module as it's performace is slightly higher than the DICOMReader implementation.

Figure 14.2 shows the use of the GDCMReader module. In order to properly read a DICOM volume, the GDCMReader must be supplied with the dimension of the volume to be read and the directory containing the series to read. By default, WrapITK supports two- and three-dimensional volumes. In order to include support for higher dimensions, the appropriate WrapITK variable must be set using cmake.

14.3.3 Volume Processing With ITK and VisTrails

Typically, DICOM volumes are written with no 16-bit unsigned shorts. Unfortunately, most systems allow the display of only 8-bit values. Because of the higher precision inherent in DICOM data, it is often preferable to perform any computation, segmentation, or processing on the data prior to rescaling in order to utilize as much information as possible.

14.3.4 Volume Processing With ITK and VisTrails

Those familiar with the ITK libraries know that ITK image filters are typically templated based on the dimensionality of the data being processed as well as the data type being processed. In VisTrails, these parameters are handled through the use of PixelType Modules. Although any ITK Filter wrapped in VisTrails can accept any of these PixelTypes, the underlying implementation may not be compatible with the input PixelType. Using PixelTypes incompatible with the underlying filter implementations is the most frequent cause of error when executing otherwise functional pipelines in VisTrails.

When processing volumes, it is often necessary to extract a single slice from the volume at different stages of the processing pipeline. This is possible in VisTrails through the use of the ExtractImageFilter. Given a volume, a Region, and Dimensionality information, the ExtractImageFilter can extract a single slice from the data volume that can be used in further

[4]Grass roots DiCoM Project. http://www.creatis.insa-lyon.fr/Public/Gdcm/

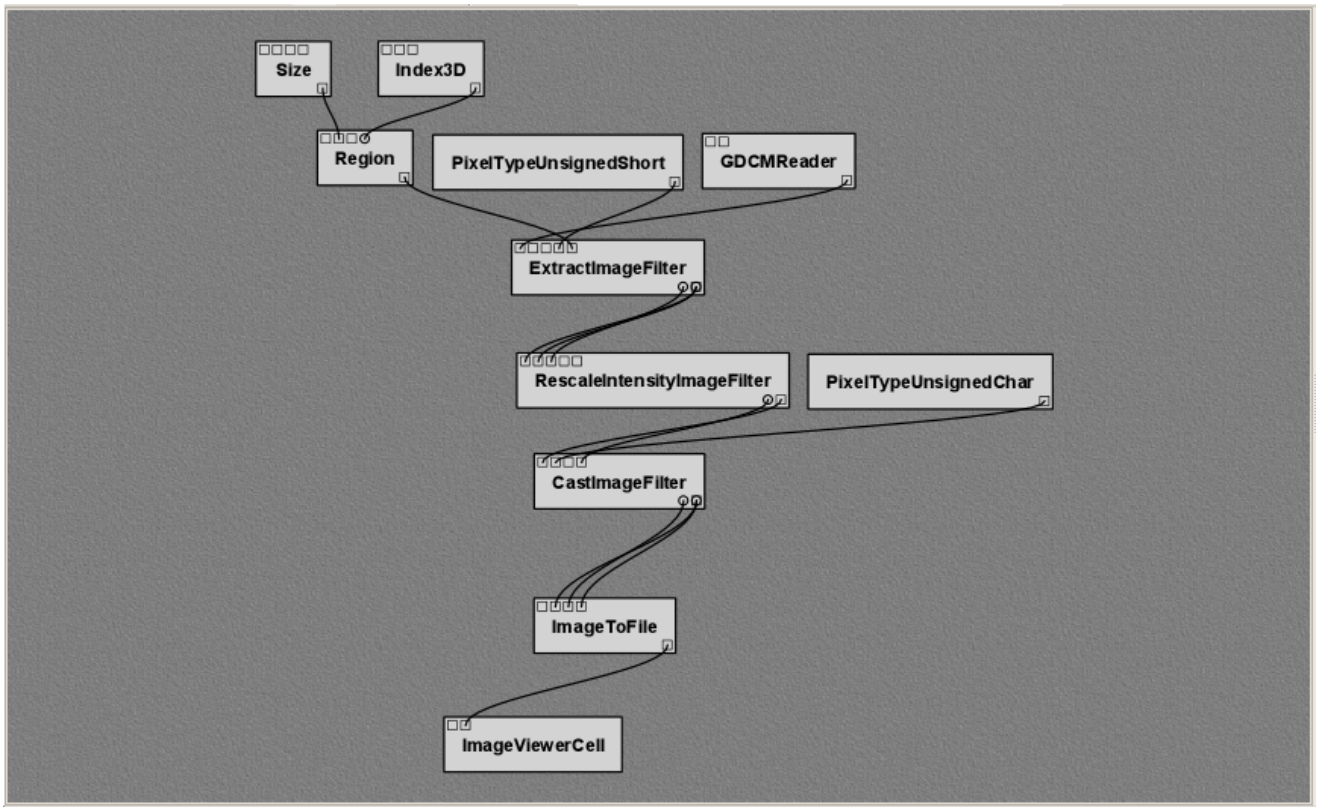


Figure 14.2: VisTrails workflow utilizing ITK to extract a single slice from a DICOM volume. The slice is chosen by first forming a Region to extract. The result is viewed through the use of standard VisTrails Spreadsheet modules.

processing, previewing the results, or writing to disk. An example workflow that extract a slice from a DICOM volume can be seen in Figure 14.2.

14.3.5 Visualizing the results

Although ITK's processing filters and the DICOM standard both support 16-bit processing and storage, many image viewers are capable of displaying in only 8-bit resolution using the UnsignedChar PixelType. Since the output of an ITK processing workflow is an image, it makes sense to view it as such. This means that we are required to both remap the data values in the image to 8-bits as well as perform a casting operation to change the data type from unsigned shorts to unsigned chars. These operations are performed through the use of the RescaleIntensityImageFilter and the CastImageFilter. Figure 14.2 demonstrates the use of the RescaleIntensityImageFilter and the CastImageFilter in conjunction with the ImageToFile and ImageViewerCell Modules to view the resulting slice in the VisTrails Spreadsheet.

Chapter 15

Frequently Asked Questions

15.1 Running workflows

How can I run a workflow using the command line? Call vistrails using the following options:

```
python vistrails.py -l -b path_to_vistrails_file -w pipeline
```

Using the command line, we'd like to execute a workflow multiple times, with slightly different parameters, and create a series of output files. Is this possible? Starting in rev 444, we can change parameters that have an alias through the command line.

For example, offscreen pipeline in offscreen.xml always creates the file called image.png. If you want generate it with a different filename:

```
python vistrails.py -l -b ../examples/offscreen.xml \
  -w offscreen -a"filename=other.png"
```

filename in the example above is the alias name assigned to the parameter in the value method inside the String module. When running a pipeline from the command line, VisTrails will try to start the spreadsheet automatically if the pipeline requires it. For example, this other execution will also start the spreadsheet:

```
python vistrails.py -l -b ../examples/head.xml -w aliases \
  -a"isovalue=30,Diffuse_Color_R=0.8,Diffuse_Color_G=0.4,Diffuse_Color_B=0.2"
```

I can load a vistrail, and the version tree shows up fine. However, no pipelines appear when I click on a version. What gives? THIS ANSWER IS OUT OF DATE. PLEASE REFER TO CHAPTER 11. The most likely reason is that the vistrail uses a package that is not registered with VisTrails. You need to identify the needed package and add it to your .vistrails/startup.py. A single line like the following should be enough:

```
addPackage('enter_package_name_here')
```

Some packages might need more information. For example:

```
addPackage('afront',executable_path='/path/to/afront')
```

Refer to the package documentation for details. The one inconvenient step is that currently there's no automated way to describe what is the missing package. We're working on this feature for future releases.

15.2 Building workflows

Is there a way to give each widget a "display name" in addition to the module name at the center of the widget? Yes, but it is not easily accessible from the GUI and it definitely needs to be more intuitive. For now, we use the annotation value of key "__desc__" as a module label. If you want to set a PythonSource label, you have to select the module. Then click on the Annotation tab, and add a key named "__desc__", whatever value you set to this key will be the label. We are currently working on a new interface for this functionality.

15.3 Spreadsheet

Below, "pipeline" is a version number or a tag.

How can I save an image from the spreadsheet? While having the focus on a spreadsheet cell press "Ctrl" (on Windows) and select the camera to take a snapshot. The system will prompt you for the location and file name where it should be saved. The other icons can be used for saving multiple images that can be used for generating an animation on demand.

Is it possible to save the complete state of the spreadsheet? Yes, the spreadsheet has a "Save As" functionality.

Can I view multiple sheets at the same time? Yes. Each sheet on the spreadsheet can be displayed as a dock widget separated from the main spreadsheet window by dragging its tab name out of the tab bar at the bottom of the spreadsheet.

Then, how can I put back a separated sheet? A sheet can be docked back to the main window by dragging it back to the tab bar or double-click on its title bar.

How can I order sheets on the spreadsheet? This can be done by dragging the sheet name on the bottom top bar and drop it to the right place.

Can I control where a cell will be placed on the spreadsheet window? By default, an unoccupied cell on the active sheet will be chosen to display the result. However, you can specify exactly in the pipeline where a spreadsheet cell will be placed by using CellLocation and SheetReference. CellLocation specifies the location (row and column) of a cell when connecting to a spreadsheet cell (VTKCell, ImageViewerCell, ...). Similarly, a SheetReference module (when connecting to a CellLocation) will specify which sheet the cell will be put on given its name, minimum row size and minimum column size. There is an example of this in examples/vtk.xml (select the version below Double Renderer).

15.4 Integrating your software into VisTrails

How can I integrate my own program into VisTrails? The easiest way is to create a package. Writing a package is often very simple, see Chapter 11 for detailed instructions.

How do modules deal with multiple inputs in a same port? For compatibility reasons, we do need to allow multiple connections to an input port. However, most package developers should never have to use this, and so we do our best to hide it. The default behavior for getting inputs from a port, then, is to always return a single input. If on your module you need multiple inputs connected to a single port, use the 'forceGetInputListFromPort' method. It will return a list of all the data items coming through the port. The VTK package uses this feature, so look there for usage examples (packages/vtk/base_widget.py)

Are there mechanisms for attaching widgets to different modules/parameters? Right now, we have a mechanism for putting a specific widget for an input port. For example, if a port is SetColor(red, green, blue), we can put a color wheel widget there. Or we can also replace the SetFileName port with a File Widget. However, this is not per parameter (only per port). We are currently working on this problem.

15.5 VTK

Given a VTK visualization, how can I generate a webpage from it? Check out the html pipeline in offscreen.xml.

I'm trying to use VTK, but there doesn't seem to be any output. What is wrong? To use VTK on VisTrails, you need a slightly different way of connecting the renderer modules. Instead of using the standard RenderWindow/RenderWindowInteractor infrastructure, you simply connect the renderer to a VTKCell. The examples directory in the distribution has several VTK examples that illustrate.

Bibliography

[1] T. S. Yoo, M. J. Ackerman, W. E. Lorensen, W. Schroeder, V. Chalana, S. Aylward, D. Metaxes, and R. Whitaker, "Engineering and algorithm design for an image processing API: A technical report on ITK - The Insight Toolkit," *Proceedings of Medicine Meets Virtual Reality*, pp. 586–592, 2002.

[2] R. A. Banvard, "The visible human project image data set from inception to completion and beyond," *Proceedings of CODATA*, 2002.

[3] L. Ibanez, W. Schroeder, L. Ng, and J. Cates, *The ITK Software Guide*, 2nd ed., Kitware, Inc. ISBN 1-930934-15-7, http://www.itk.org/ItkSoftwareGuide.pdf, 2005.

Index