

# Contents

## **Chapter 1 Introduction**

- [1.1 Igraphs](#)
- [1.2 Extensive Operator Libraries](#)
- [1.3 Generalized Data Objects](#)
- [1.4 Data-Flow Environment](#)
- [1.5 Client/Server Model](#)
- [1.6 Parallel Execution](#)
- [1.7 Data/Demand-Driven Modes](#)
- [1.8 CAD-like Environment](#)
- [1.9 User Extendable](#)

## **Chapter 2 Getting Started**

- [2.1 Installation](#)
- [2.2 Building Operators and Servers](#)
- [2.3 Using WATCOM C with WiT](#)
- [2.4 How to Avoid Reading This Manual](#)
- [2.5 The Main Panel](#)
- [2.6 Microsoft Windows Environment](#)

## **Chapter 3 Manipulating Data Objects**

- [3.1 Viewing Images and Objects](#)
- [3.2 Images and Image Properties](#)
- [3.3 General Objects](#)
- [3.4 Reading/Writing and Search Paths](#)
- [3.5 Object File Format](#)

## **Chapter 4 Data-Driven Mode**

- [4.1 Executing an Igraph](#)
- [4.2 Data-Driven Options](#)
- [4.3 Run Controls](#)
- [4.4 More Example Igraphs](#)
- [4.5 Building an Igraph](#)
- [4.6 Splitting Links](#)
- [4.7 Editing Links](#)
- [4.8 Moving Operators and Links](#)
- [4.9 Zooming and Panning the Workspace](#)
- [4.10 Saving and Deleting Igraphs](#)
- [4.11 Getting Help](#)
- [4.12 Operator Selector](#)
- [4.13 The Graph Menu](#)

## **Chapter 5 Advanced Igraph Building**

- [5.1 Operator Properties](#)
- [5.2 Link Properties](#)
- [5.3 Customizing the Grid](#)
- [5.4 Pick Tolerance](#)

5.5 Object Names

5.6 Cut and Paste Across Top Level Igraphs

5.7 Parameters as Inputs

5.8 Hierarchical Operators

## **Chapter 6 Data-Flow Model**

6.1 The Scheduler

6.2 / Extensions to Basic Data-Flow Model

6.3 Some Caveats of Data-Flow

## **Chapter 7 Demand-Driven Mode**

7.1 Reading and Displaying Images

7.2 Demand-Driven Mode Cards

7.3 Executing a Series of Operators

7.4 Demand-Driven Options

7.5 Demand-Mode Scripts

## **Chapter 8 General Tools and Techniques**

8.1 Zooming and Panning with the Mouse

8.2 Resizing Windows

8.3 Selecting Graphical Objects

8.4 Moving and Adjusting Graphical Objects

8.5 Color Panel

8.6 Zoom/pan

8.7 Printing

8.8 Icon Editor

8.9 Array Editor

## **Chapter 9 Interactive Operators**

9.1 GetData

9.2 Surface

9.3 Graph

9.4 System

## **Chapter 10 Configuration**

10.1 Command Line Options

10.2 Directories

10.3 Designing Your Own Icons

10.4 Color

10.5 Servers

10.6 Execution Environment

## **Appendix A WiT File Format**

## Chapter 1 Introduction

WiT is a powerful visual programming package for designing computer algorithms with block diagrams which can be *executed*. It is particularly suitable for complex image processing applications in areas such as machine vision, medical imaging, and imaging research. Solutions are rapidly developed and elegantly described by building imaging graphs, called *Igraphs*, using icons and links with "point-and-click" simplicity. Figure [Example WiT Session](#) shows an example of an Igraph and a typical WiT session.

Example WiT Session  
(Figure omitted on purpose on demos)

### 1.1 Igraphs

The Igraph concept is one of the most noticeable features in WiT. An Igraph is a graphical representation of an imaging algorithm. On an Igraph, each icon represents an operation, such as reading an image file, performing a convolution, or generating a lookup table. Each operation takes some number of input objects and produces some output objects, which flow along paths specified by directed links.

Operators are represented by *graphical* icons, not just a block diagram with text inside. This makes your imaging algorithm pleasant to work with and easy to understand. The properties of each operator can be interactively modified and displayed below the icon so that the Igraph is a *complete* representation of your algorithm. Flow control operators such as *if-then-else* and *for-loop* allow you to incorporate conditional logic within an Igraph.

When an Igraph is executed, WiT dynamically shows you data objects travelling along links or being processed by operators, and which inputs and outputs are being consumed and generated. Controls for pausing, continuing, or stopping Igraph execution are available.

Igraphs present a quick way of prototyping and testing image analysis procedures without committing hours of tedious programming. Igraphs also serve as concise and accurate documentation of an algorithm which is easy to read and understand.

### 1.2 Extensive Operator Libraries

WiT comes with a rich set of operators classified into iconified libraries in areas such as read/write, point, filter, transform, morphology, segmentation, and measurement. A powerful flow control library provides operators which allow Igraphs to employ conditional branches, loops, synchronization, and sequencing through arrays of objects.

Operators are available for interactive manipulation and viewing of image data. For example, you can interactively enter graphical data, such as points, rectangles, or polygons on images; plot data sets as graphs in a variety of styles; display image data as 3-D surfaces; adjust image contrast; and display image intensity profiles.

### 1.3 Generalized Data Objects

The initial stages of imaging algorithms may deal exclusively with images, but as an imaging algorithm progresses to a higher level segmentation, measurement, or recognition phase, other data types such as chain-codes and feature vectors become necessary. All data types, not just images, are treated as *objects* in **WiT**. An object is similar to the `structure` or `record` used in some programming languages. It can contain many fields. For example, an object for an employee may contain his name, address, position, and salary as fields. A `department` object may contain an array of `employee` objects. An `image` object would contain its width, height, depth, and data bits.

*Any* object type can be viewed by placing a magnifying glass (see Figure [Example WiT Session](#)) on a link. All object types can be displayed, read, saved, and printed. For example, an image and a vector of statistical data are being displayed in Figure [Example WiT Session](#).

### 1.4 Data-Flow Environment

**WiT** adopts the data-flow model of program execution instead of the control-flow model used in most programming languages such as C or BASIC. In the data-flow model, an algorithm is expressed in terms of operators that process input data and produce output data (called *tokens*). An operator may be a simple procedure, such as addition or subtraction, or it may be an intricate function consisting of thousands of lines of code. An operator becomes ready for execution when all its input objects are available. With data-flow, it is not necessary to assign names to variables. The relationship of program code (operators) to data (objects that flow along links) is expressed graphically by how operators are connected. This makes the data-flow model more suitable for a graphical environment. Moreover, it is natural to express parallel program execution with data-flow. For example, suppose you want to program the equation

$C = (A + B) \times (A - B)$  If you program this in C, you might write

```
T1 = A + B;  
T2 = A - B;  
C = T1*T2;
```

Notice the use of variable names, particularly `T1` and `T2` for holding temporary results. You may argue that you can program the entire equation in a single C statement and thus avoid temporary variables, but bear in mind that this is a simplified example. The operations `+` and `-` may be arbitrarily complex expressions.

Notice also that in reality it is not necessary that the computation of `T2` should follow `T1`, or vice versa. `T1` and `T2` can be computed at the same time, since they both depend only on `A` and `B`. Most programming languages do not allow such *parallelism* to be expressed explicitly. Special languages (such as Ada and Occam) are needed. One possible implementation might look like this

```
CoBegin {
```

```

    {
      T1 = A + B;
    }
    {
      T2 = A - B;
    }
  }
  C = T1*T2;

```

where we assumed a hypothetical programming language which has the keyword `CoBegin` which allows you to explicitly specify what instructions can be computed in parallel.

In contrast, this equation can be elegantly expressed in a data-flow diagram (generally called a data dependency graph) as in Fig. [Data Dependency Graph](#).

### Data Dependency Graph (Figure omitted on purpose on demos)

The parallelism is implied by the link branches after the `READ A` and `READ B` operators. There is no longer need for intermediate variable names. Their existence is represented implicitly by the links between the `+` and `-` operations and the `x` operation. In fact, if *A* and *B* are results from some other graph, then even their names are unnecessary.

Parallel computation can greatly reduce program execution time. For example, your algorithm may require an image to be filtered with ten different filters so that the results can be compared. If you have ten computers in your local area network, **WiT** can instruct each computer to filter the image with a different filter. The overall execution speed of the algorithm can then be up to ten times faster.

**WiT** exploits these advantages of the data-flow model to offer the user *Igraphs* which are intuitive, self-descriptive, and capable of executing in parallel on many computers at the same time.

## 1.5 Client/Server Model

**WiT** is based on a client/server model and consists of two parts: a graphical user interface (GUI) and some (zero to infinitely many) computational servers, which are separate programs communicating with one another using some kind of interprocess communication (IPC) mechanism. The client/server arrangement is superior to a single program in many respects:

- The GUI and server components can run on different computers, which may even be of different makes. For example, you can have the GUI running on a PC, and the server running on a Cray supercomputer.
- You can have more than one server, which together with the data-flow model, will be able to run operators in parallel on different servers at the same time.
- New operators can be added by rewriting the server only, which is small and easy to

understand, and compiles fast.

- Because the server is small, it can even run on a diskless computer, which may need to be physically close to special purpose hardware.
- The server is often continually changed to accommodate new operators, which means it is prone to bugs. Since the server is a separate program from the GUI, any bugs are confined to the server itself, which makes it easy to detect errors and debug new operators.

## 1.6 Parallel Execution

Parallel computation is understandably highly complicated and is still a major research topic. **WiT** does not solve all the issues in parallel computation, but it offers a practical application environment which allows the user to tap into the power of networked computers which are so common nowadays.

A parallel algorithm which involves running small operations, such as the example in Figure [Data Dependency Graph](#), is termed *fine grain* parallelism. If the individual operations are large, perhaps each involving the execution of a program such as a compiler or word processor, then it is termed *coarse grain*. If it is something in between, such as when the operators are function calls, then it is termed *medium grain*. Obviously the boundaries between the different classification are rather vague. **WiT** is best suited to medium to coarse grain parallelism. It will work with fine grain operations, but the overhead involved may destroy any potential speed gains.

Parallelism is expressed in an Igraph by having link branches, or operators that produce more than one output. You can have as many branches as you like from a link. This allows you to easily represent parallelism in your algorithm. **WiT** employs a sophisticated scheduler that can analyze link connections and determine how to dispatch operations efficiently across a network of servers (if available).

An operation is dispatched by sending the necessary data objects to a server and then instructing it to carry out the operation. While this server is busy, another operation can be dispatched to another server, and so on until all available servers are busy. As soon as a server comes back with results, it can be given a new task. The **WiT** scheduler knows what servers are capable of executing which operators (if the servers are different).

Objects can be sent between heterogeneous machine architectures so that **WiT** can take advantage of different hardware platforms such as vxWorks host processors, Datacube hardware, or DSP networks. All the necessary data conversion and communication is transparent to the user.

## 1.7 Data/Demand-Driven Modes

Although Igraphs are extremely powerful in expressing complex imaging algorithms, sometimes it is more convenient to try different operators and see the results immediately. For this reason, **WiT** provides two modes of execution: *data-driven* and *demand-driven*.

Igraphs are used only in data-driven mode. In demand-driven mode, you can run operators on images or objects that you specify, and the results are displayed immediately. This is helpful in exploring and understanding the set of imaging operations **WiT** offers, as well as debugging any new operators that you have developed yourself. Actions performed in demand mode can be saved as *demand scripts* and played back later.

## 1.8 CAD-like Environment

**WiT** is designed to handle highly complex Igraphs. Hundreds of operators can be utilized in a single algorithm without any problems. High-speed zoom and pan features make it possible to view hundreds of operators on the computer screen.

**WiT** facilitates the design and layout of Igraphs by providing various features which are commonly found in CAD systems. Igraph components may be selected then copied or moved about the workspace. The workspace may be increased or reduced in size. Links are easily wired or reshaped where vertices are snapped to a grid of variable resolution. Pick accuracy can be set small so that closely spaced objects can be distinguished, or set large so that objects can be easily selected. Groups of operators can be interactively made into sub-graphs, allowing algorithm details to be hidden if the user is not interested. Named ports are used to specify connectivity among Igraphs. Sub-graph nesting level is unlimited. Multiple sub-graphs can be viewed at the same time.

## 1.9 User Extendable

Users can easily extend **WiT** by defining new C functions and data types, and custom servers to access specialized hardware. Although **WiT** was developed for image processing, it can serve as a general visual programming tool for developing solutions to problems interactively using a data-flow model. The object-oriented structure of **WiT** promotes the design of well structured and well behaved algorithms. Here we take a brief look at the procedures. For details, see the *Programmer's Guide*.

### 1.9.1 Adding New Operators

Adding your own operators to **WiT** is easy. All you have to do is:

1. Specify the number and type of inputs, outputs, and parameters that the new operator requires, using a special but very simple *operator definition language*.
0. Write a C function to process the inputs and generate the appropriate outputs,
1. Rebuild the server.

The new operator is then ready for use!

### 1.9.2 Adding New Object Types

Just about any data structure you can create in the C programming language can be made into a **WiT** object. Built-in object types include integers, floats, strings, images, and vectors of these basic types.

By using the **WiT** object definition language similar to the C `struct` syntax, user-defined objects are rapidly defined and built into **WiT** using a separate program called **witbuild**. **WiT** will then know how to send/receive, free, copy, print, read/write, and display the new object type.

### 1.9.3 Adding New Server Types

Due to the client/server nature of **WiT**, the server code is extremely simple. It is shipped with **WiT** in source code form, so that you can modify it to create your own server. For example, you may want to make a server which initializes you specialized hardware on start-up, or you may want to make a server that has a graphical user interface, possibly running on a different display than the **WiT** GUI.



## Chapter 2 Getting Started

### 2.1 Installation

**WiT** is a 32-bit Windows application which can be run on either Windows NT or Windows 3.1 systems. For Windows 3.1 users, the **Win32s** package produced by Microsoft must be installed *prior* to installing **WiT**. **Win32s** is a set of 32-bit DLLs which are automatically invoked when a 32-bit application is launched. It will not affect other Windows applications.

If you are using **WiT** on Windows NT, do *not* install **Win32s**. If you are using Windows 3.1 and do not already have **Win32s** installed, follow the instructions on the supplied **Win32s** diskette to install it.

To install **WiT**, follow the instructions printed on the **WiT** diskettes. The **WiT** installation uses a Windows `setup` program which will prompt you for a target drive and directory name. Setup automatically modifies your `autoexec.bat` file to define a `$WITHHOME` environment variable which is set to the name of the **WiT** directory on your hard drive. We will refer to this directory as the **WiT** directory. It will also create a **WiT** and **WiT** builder icon in a **WiT** folder. To launch **WiT**, double click on the **WiT** icon in the **WiT** folder.

The **WiT** distribution files are organized into a directory hierarchy as shown in Figure [WiT Distribution Tree](#). The contents of the directories are:

<code>bin</code>	<b>WiT</b> executables.
<code>config</code>	Default icon, <code>witrc</code> , and <code>iconcache</code> . On-line help files.
<code>demo</code>	Example <code>igraphs</code> .
<code>images</code>	Collection of 8-bit, 16-bit, float, and color images.
<code>include</code>	Header files for object/operator/server development.
<code>lib</code>	<b>WiT</b> operator development directory containing operator and object definition files, icons, help files, include files, and libraries.
<code>def</code>	Operator and object definition files.
<code>icons</code>	Operator icon files.
<code>help</code>	Operator help files.
<code>include</code>	Operator and object header files.
<code>src</code>	Operator source directory for <b>Prototype</b> library.
<code>arch</code>	Prebuilt libraries for your architecture.
<code>servers</code>	Server directory containing executables and interface source code.
<code>bin</code>	Server executables.
<code>src</code>	Server interface source code.
<code>wit arch</code>	Server build directory.
<code>samples</code>	A <b>WiT</b> development directory example setup.

## WiT Distribution Tree (Figure omitted on purpose on demos)

### 2.2 Building Operators and Servers

If you intend to build your own operators or servers, you need to have a 32-bit compiler that can link your new C code to the supplied **WiT** prebuilt libraries. Currently, **WiT** supports WATCOM C on both Windows 3.1 and NT platforms. Microsoft C is supported on NT platforms only. The list of supported compilers will grow in the future.

### 2.3 Using WATCOM C with WiT

WATCOM C 10.0 has been tested thoroughly with **WiT**. However, any earlier version may or may not work properly. Hereafter, whenever we refer to WATCOM C, it is understood that we mean WATCOM C 10.0.

WATCOM C has numerous switches some of which can be deadly if you do not select carefully. For example, some switches specify what operating system you are *supposed* to be using, but WATCOM does not check whether you are *actually* using that operating system or not.

To aid the programmer, we have summarized the procedures you should adopt when you compile new operators and servers for **WiT**. It is not absolutely necessary that you follow these steps, but if you decide not to follow these steps and you get into trouble, refer to the steps outlined here to determine where the error is.

For your convenience, all suggested settings mentioned in this section can be found in ASCII form in the directory \$WITHOME\samples\watcom. We will refer to this as the *samples* directory.

#### 2.3.1 The G Environment

The supplied **WiT** prebuilt libraries provide utilities to simplify cross-platform program development. It is a subset of the **G** program development environment that Logical Vision uses internally. **G** is a package for developing platform independent applications with or without a graphical user interface (GUI).

To ensure your programs are portable, it is *very* important that you include only the **G** header files and not any header files from the platform you are using (e.g. `windows.h`). Logical Vision has undergone extensive testing to verify that all **G** functions perform correctly under supported platforms. If you decide to interact directly with the underlying platforms, you are on your own.

The **WiT** installation procedure automatically creates a directory **G** under the **WiT** directory. It also creates an environment variable `GHOME` that points to this directory. We will refer to this as the **G** directory.

#### 2.3.2 Include File Fixes

WATCOM C has a non-standard interface for variable argument list function calls. This *has* to be fixed if you want your applications portable. The directory `watcom` under the *samples* directory contains these corrected header files:

```
stdarg.h
stdio.h
```

For safety reasons, the install procedures do *not* automatically copy these files to the standard WATCOM include directory `\WATCOM\H`. You must copy them yourself.

### 2.3.3 Compiler and Linker Switches

WATCOM can use a combination of environment variables and `makefile` flags to specify compiler and linker switches. We suggest you put these lines in your `autoexec.bat` file:

```
set wcc386=/dWINNT /w1 /e5 /zq /j /zp4 /d2 /ei
set wcl386=/dWINNT /w1 /e5 /zq /j /zp4 /d2 /ei
```

These lines can be found in file `autoexec.bat` in the *samples* directory. The `/dWINNT` switch is needed so that you can do conditional compilation when you port your files to other platforms. `WINNT` is used for both Windows 3.1 and NT platforms.

### 2.3.4 Makefiles

To simplify user makefiles, we provide master makefiles stored in the **G** directory which should be included in user makefiles.

There are two master makefiles:

1. `lib.mk` is for static libraries.
2. `exe.mk` is for executables.

Sample user makefiles can be found in the *samples* directory:

1. `libsam.mk` is a sample makefile for making static libraries. It includes `lib.mk` from the **G** directory.
3. `exesam.mk` is a sample makefile for making executables. It includes `exe.mk` from the **G** directory.

To make a new library, copy `libsam.mk` to `makefile`, and modify the `OBJS` list in the `makefile`. Then simply type `wmake/u`.

Making executables is similar. Just substitute `libsam.mk` with `exesam.mk`.

## 2.4 How to Avoid Reading This Manual

As with many complex tools, most people prefer to learn by experimentation rather than going through thick manuals. This manual has been written as a tutorial with a step-by-step presentation which should be easy to follow and not take long to go through. However, if you would rather start experimenting right away, please read this section at least! If you are not familiar with Microsoft Windows, you should read Section [Microsoft Windows Environment](#) first.

Like all graphical programs, most of your interaction with **WiT** is done via your mouse. **WiT** requires a three button mouse to operate. If you have a two button mouse, you can simulate the middle button by holding down the `control` key and use the left button. Specifically, **WiT** recognizes these mouse actions for various functions:

Button	Down	Up	Drag
Left	Select. Start area select. Set position.	Done area select.	Move and modify objects.
Middle	Add to or remove objects from selected list. Backup when entering polyline.	No function	No function
Right	Display pop- up menus.	No function	No function

With shift key held down:

Button	Down	Up	Drag
Left	Start area zoom.	Done area zoom.	Update area zoom.

<b>Middle</b>	Start pan windows.	Done pan windows.	Pan windows.
<b>Right</b>	No function	No function	No function

Context sensitive on-line help is available for most objects, such as icons, links, and Igraph. Point the mouse at the object and hit the **F1** key. The complete User's Guide (what you are reading now) is available on-line from the **Help** pull-down menu.

With this knowledge, you are ready to explore **WiT**. But if you would rather get a complete feel of what **WiT** offers before starting to experiment, read on.

## 2.5 The Main Panel










The main panel consists of a menu bar and a tool bar below it. Each item on the menu bar produces a pull-down menu when pressed. Within a menu, items that are followed by an ellipsis (...) bring up a sub-panel when selected.

<b>File</b>	<b>Load...</b>	Load a new Igraph or demand script. The current Igraph or demand script is cleared.
	<b>Save</b>	Save the current Igraph or demand script.
	<b>Save as...</b>	Save the current Igraph under a different name.
	<b>Delete...</b>	Delete an Igraph or object file.
	<b>New</b>	Clear current Igraph.
	<b>View objects...</b>	Display an object on a detached window.
	<b>Print...</b>	Make a hardcopy of the workspace and/or displayed objects.
<b>Edit</b>	<b>Quit</b>	Exit <b>WiT</b> . A warning message will be issued to confirm this.
	<b>Cut</b>	Delete all selected objects.
	<b>Copy</b>	Copy selected objects to buffer.
	<b>Duplicate</b>	Duplicate selected objects (copy <i>and</i> paste).
	<b>Paste</b>	Paste objects in buffer to Igraph.
<b>Graph</b>	<b>Undo</b>	Undo the last <i>edit</i> change. Other changes such as erase or running an <i>igraph</i> cannot be undone.
	<b>Select all</b>	Select all objects in Igraph workspace.
	<b>Node group properties...</b>	Set properties for all selected nodes.
	<b>Link group properties...</b>	Set properties for all selected links.
	<b>Trace all</b>	Trace data-flow path for selected objects.
	<b>Trace inputs</b>	Trace data-flow path for inputs of selected objects.















	<b>Trace outputs</b>	Trace data-flow path for outputs of selected objects.	
<b>Operators</b>	<b>Make operator... Selector</b>	Make Igraph into an operator. Text-based operator selector with wild-card capability and access to operators from all libraries.	
	<b>Dataflow</b>	Flow control operators (must be present).	
	<b>Interactive</b>	Operators that run on the GUI (must be present).	
	<b>Read/Write</b>	Operators for reading and writing objects.	
	<b>Point</b>	Operators that operate on each pixel individually, e.g. invert, subtract.	
	<b>Filter</b>	Digital filters, e.g. low-pass, high-pass, 1 and 2-D convolution.	
	<b>Morphology</b>	Operators that operate on shapes, e.g. dilate, erode.	
	<b>Transform</b>	Transform operators, e.g. FFT, discrete cosine.	
	<b>Measurement</b>	Operators that measure various aspects of an image, e.g. statistics, spectrum.	
	<b>Segmentation</b>	Operators that classify portions of an image into distinct segments, e.g. collect blobs, local thresholding.	
	<b>Pyramids</b>	Multi-resolution imaging, e.g. decimate, burt.	
	<b>Options</b>	<b>Setup...</b>	General setup options, such as execution mode and heap size tracking.
		<b>Data driven...</b>	Data-driven options, such as grid spacing and scheduling mode.
<b>Demand driven...</b>		Demand-driven options, such as number of objects cards to be maintained in the workspace.	
<b>Run</b>	<b>Start</b>	Start Igraph or demand script execution, clearing all previous states.	
	<b>Pause</b>	Pause Igraph execution. Can resume later with <b>continue</b> .	
	<b>Continue</b>	Continue after a previous pause.	
	<b>Stop</b>	Stop Igraph or demand script execution, cannot continue afterwards.	
	<b>Reset tiles</b>	Reset object frame tiling to begin at upper left corner of screen.	
<b>Help</b>	<b>Contents</b>	Complete user's manual.	
	<b>About...</b>	General information about <b>WiT</b> .	










The tool bar provides a set of push buttons to perform many of the commonly used operations.

On the left hand side of the tool bar, a status indicator informs you about the current status of **WiT**. The meanings of the indicators are:

-  Execution stopped, waiting for user action: edit Igraph, start run, etc.
-  Executing at **walk** speed.
-  Executing at **run** speed.
-  Executing at **sprint** speed.
-  Executing at **warp** speed.
-  Waiting for user input for an operator.
-  User has requested execution to be stopped.
-  User has requested execution to be paused.
-  Execution has been successfully paused.

The meanings of the tool bar buttons are:

-  Erase current Igraph or demand script.
-  Read Igraph or demand script.
-  Save Igraph or demand script.
-  Switch to data-driven mode.
-  Switch to demand-driven mode.
-  Run at **walk** speed.
-  Run at **run** speed.
-  Run at **sprint** speed.
-  Run at **warp** speed.
-  Pause execution.
-  Continue execution after pause.
-  Stop execution.
-  Reset window tiling to start at upper left of screen.
-  Make hierarchical operator for current Igraph.

-  Home workspace to upper left corner.
-  Pan workspace left.
-  Pan workspace right.
-  Pan workspace up.
-  Pan workspace down.
-  Cut selected Igraph objects or first demand card.
-  Copy selected Igraph objects.
-  Paste selected Igraph objects.
-  Undo last Igraph editing action.

### 2.5.1 Setup Panel

The **Setup** panel provides general setup options (Fig. [Setup Panel for General Options](#)). The entries are:

#### Setup Panel for General Options (Figure omitted on purpose on demos)

<b>Run mode</b>	Select between demand or data-driven mode.
<b>Object placement</b>	<p>This option controls how new object (images or text) windows are to be placed:</p> <ul style="list-style-type: none"> <li>• Random --- random locations on the screen.</li> <li>• Mouse --- current mouse position.</li> <li>• Tile --- objects are tiled (i.e. without overlap) across the screen.</li> </ul>
<b>Status</b>	You can show or hide the <b>WiT</b> status panel with this control. If the status window is hidden, all status messages will be discarded. This allows you to eliminate the delay caused by printing these messages. When <b>WiT</b> issues a warning (which goes to the status window), it will <i>always</i> bring up the status window.
<b>Workspace Colors</b>	This option controls colors used operators, links, etc:



- Color --- multi-colored.
- Light --- dark objects on a bright background.
- Dark --- bright objects on a dark background.
- Custom --- (color displays only) user defined color scheme.

## Speed

The speed control determines whether how much feedback **WiT** provides when an Igraph is run or when a demand script is played back. **Walk** is slowest and **warp** is fastest.

At **walk** speed, operators turn green when they execute, and their names appear on the status window. The scheduler waits one second after each operator execution before proceeding to the next operator. Token travel on links is animated slowly so that it is easy to follow the progress of the Igraph.

At **run** speed, the scheduler pause duration is shorter, and token travel is animated at higher speed than **walk** speed.

At **sprint** speed, token animation is turned off completely.

At **warp** speed, all feedback is disabled, the status window is taken down, and even the running boy icon becomes a static lightning bolt display, thus allowing Igraphs to run absolutely unimpeded.

With demand-driven mode, the speed selection only affects the delay between each step when a script is playback. Speed selection does not affect manual execution in demand-driven mode.

## Adjust...

This will bring up a new panel for you to adjust the delay associated with each execution speed. The delay affects minimum operator execution time and token animation time in data mode and script playback speed in demand mode. Since some computers are faster than others, the numbers should be treated as relative rather than absolute (such as number of seconds). In other words, simply adjust the numbers until the delay is satisfactory, without paying much attention to what the delay value actually is.

## Heap

This gauge reports the heapsize (in Kbytes) currently in use by **WiT**. It can be enabled/disabled with the **On/Off** selector.

## 2.6 Microsoft Windows Environment

Before you start experimenting with **WiT**, it is important that you know the basic techniques to interact with a Microsoft Windows application. If you are already familiar with the techniques, then you can skip this section.

Only the GUI techniques relevant to **WiT** are discussed here, so that you can quickly learn the basics and proceed to explore **WiT**. Refer to the Windows User's Manual for more information.

### 2.6.1 Mouse Buttons and Movement

**WIT** works with a three-button mouse. In general, the left button is used for selecting things and most other actions, and to display pull-down menus from the menu bar. The middle button is used for adjusting things, such as augmenting a previous selection set. The right button, is used for invoking a menu on a canvas (graphics window), or bringing up properties of an object.

Different actions are sometimes invoked by pressing a mouse button when the shift or control keys are held down.

When you move the mouse with none of the mouse buttons held down, you are *moving* the mouse. When you move the mouse with any one of the mouse buttons held down, you are *dragging* it.

### 2.6.2 Menu Bar

Some windows (most importantly the main application window) contain a menu bar at the top. Each item on the menu bar contains a menu which you can pull down by clicking with the left mouse button. You can then move the mouse to the item you want and then click the left mouse button there to select it.

### 2.6.3 Buttons

Buttons should be pressed by using the left mouse button. Actions associated with a button are normally executed when a button is *released*.

### 2.6.4 Resize Corners

Windows that are decorated with double line borders can be resized. Resize corners are used in **WIT** to permit the user to alter the size of graphs, images, the Igraph workspace, etc.

### 2.6.5 Disabled Items

When an item is not applicable in the current context (such as a delete action when there is nothing to delete), it is disabled so that the user will not accidentally try to execute an invalid action. Disabled text items are shown with a gray dithered appearance. Disabled iconic items are shown with a stippled screen.

### 2.6.6 On-line Help

On-line help is available by hitting the **F1** key on the keyboard with the mouse cursor pointing to the object for which help is required. This is called context sensitive help. Of course, the window must have the current keyboard focus when you do this. (The window with the keyboard focus has a highlighted title bar.)

At this moment, context sensitive help is not supported for control widgets (buttons, sliders, etc). It is only supported for Igraphs, Igraph operators, and Igraph link.

The entire User's Guide (what you are reading now) is available from the **Help** menu bar item.

## Chapter 3 Manipulating Data Objects

Data objects play a vital role in **WiT**. They can be images, simple scalar values such as integers or strings, or complex data structures which consist of many different fields. **WiT** can read, write, display, print, and send data objects across networks, and **WiT** performs its work by processing data objects with its operators. So before we get into the details of how to design algorithms with **WiT**, let us first take a moment to look at the rich set of utilities that **WiT** provides for manipulating images and general objects.

### 3.1 Viewing Images and Objects

Let us start by displaying one of the example images in **WiT**. Select the **View objects...** item from the **File** menu on the top menu bar. This will bring up the familiar file dialog.

If you have followed the instructions so far, you should now be in the `demo` directory (check the directory shown on the **File** dialog). To access the sample images, you need to go up one directory level and then go down to the `images` directory. You should now see the sample image files. Select a file (say `saturn`). The image will appear in a detached window. Try to display *all* the images at the same time!

### 3.2 Images and Image Properties

**WiT** provides many utilities for studying image data. Assuming you have brought up the `saturn` image, move the cursor inside the image and bring up the pop-up menu by pressing and holding the right mouse button. Select the **Properties...** item. You should get a panel like [Figure Image Properties Panel](#). The functions of the various image property controls are listed below. Different image types may have specific properties shown only with those types, e.g. profiles are not applicable to color images, and image index is only applicable to an image vector. What follows is a complete list of all the possible properties.

#### Image Properties Panel (Figure omitted on purpose on demos)

<b>Size/Type</b>	Original width and height of image, and the type of each pixel.
<b>Position</b>	Position of mouse in image coordinates. Upper left corner is (0,0).
<b>Value</b>	Value of pixel at which mouse is pointing at.
<b>Pixel scale</b>	Screen size of each image pixel.
<b>Frame size</b>	The size of the image window relative to the original size of the image. If <b>Custom</b> is selected, the sizes specified in the <b>Custom frame</b> explained below is applicable.
<b>Custom frame</b>	The size of the image <i>window</i> , or frame, expressed as a floating-point number for the X (horizontal) and Y (vertical) directions, relative to the original size of the <i>image</i> . These numbers track changes caused by the built-in choices from the <b>Frame size</b> item above, the resize corners on the image window, or

zooming in or out with the rubber band area select box (see below).

### Show profile

Show a plot of pixel values along a line on the image in the X or Y directions. The plot will be shown in a separate window, initially placed beneath (X) or to the left (Y) of the image. You can move and resize the profile window once it has come up. As you drag (move with left button held down) the mouse up and down on the image, the profile plot will be updated accordingly.

Not available for color images.

### Image index

Applicable to vector of images only. Select which image in the vector to display.

### Load all

Applicable to vector of images only. A vector of images is cached on the display server so that the scrollbar can be used effectively to quickly select different images within the vector. Normally this cache operate like most caches: an image is placed in the cache only when it has been accessed, i.e., when it is displayed. This can create some annoying initial delay when using the scrollbar. The **Load all** button allows you to force all the images in the vector to be loaded in the cache, provided that the cache size is not exceeded. No harm is done if the cache is not large enough. In that case the cache will be filled to capacity.

Consideration must be given to memory consumption. Images typically take up a large amount of memory. Loading a vector of images can cause severe performance degradation to **WiT** and other programs, due to excessive swapping of RAM data to and from disk. If you do not need to access all components of the image vector, then it may be better to let the cache load the images on demand.

The cache is flushed whenever the image scale has changed, such as when you scale an image using the resize corners. If you wish to pre-load all images in the cache, then you will have to click the **Load all** button every time you resize the image window. An example of image vector will be presented in Section [Collect](#).

### Colormap

Behavior of this item is different for grayscale and color images.

For grayscale images, three colormap types are available:

- Grayscale: Consists of 48 shades of gray. The number of shades can be user configured (see Chapter [Configuration](#), *Configuration*).
- Pseudo: Grayscale values in an image are mapped to a set of colors representing the visible light spectrum from blue to yellow to red. Pseudo-color is useful for enhancing the contrast between different parts of a grayscale image.
- Custom: Bring up a **Custom Color** panel which allows you to specify colors for specific pixel intensity ranges. See Section [User Defined Color Mapping](#) for details.

For color images, two colormap types are available:

- Shared: A low quality map shared by all displayed color images. The low quality allows all color images to be displayed reasonably at the same time.
- Render: A special colormap is computed for the current image. This map is the best colormap that can be used given the total number of colors

available to **WiT**, but may cause other color images to be displayed with incorrect colors.

- Refresh** Redisplay the image. Useful when parts of the image are not displayed properly due to actions from other windows.
- Save** Save the current image as a **WiT** object to a file. The name of the file will be the name shown on the top name stripe of the image. The saved object can be read by the **rdObj** operator. See Section Directories for more information about which directory the image will be written to.

Experiment with these controls, particularly the X and Y profiles.

In addition to the control gadgets, you will also notice a grayscale ramp on the right hand side of the properties panel. The use of the ramp will be discussed in detail in Section Color Control.

### 3.2.1 Resizing Images

When images are first displayed, they are displayed in their original size. You can change the size of the image by changing the size of the window. **WiT** handles all window size changes in a consistent way, see Section Resizing Windows for details.

Sometimes you want to magnify an image without increasing the window size, i.e., make the window size smaller than the image size, and the window displays only a portion of the image. You will also want to pan the window to look at different portions of the image while maintaining a constant scale. Zoom and pan are handled in a consistent manner across all windows in **WiT**. Details can be found in Section Zooming and Panning with the Mouse.

### 3.2.2 Image Types

**WiT** supports many different image types. Samples of all supported image types are provided in the `images` directory:

Image Type	Example Image
8-bit unsigned	saturn
8-bit signed	mri
16-bit unsigned	fighter
16-bit signed	brain
32-bit grayscale floating-point	moon
24-bit color	frog

Thanks to the carefully planned colormap manipulation procedures, you can display all these

image types correctly at the same time within **WiT** (unless you modified some of the image properties, see Section [Color Control](#)). The `saturn` image that we used is an 8-bit grayscale unsigned image. The type of an image is shown on the image properties panel.

With the exception of color images, all built-in **WiT** operators can deal with all image types. Only a subset of **WiT** operators can deal with color images directly. Typically, color images are processed by first splitting them into their RGB or HSV components (these operators are provided), processing the components, and then merging them back to form a color image again.

### 3.2.3 Color Control

Sometimes it is convenient to change the contrast on images to enhance certain features for viewing purposes. More technically speaking, we would like to change the size and position of the pixel intensity window through which we look at an image. This is what the ramp on the right of the image properties panel is for.

The current color map of the image is represented by the ramp. The arrows with numbers on the right indicate the current intensity window settings. You can drag these arrows with the left mouse button to define a smaller or larger intensity window. When the window size changes, the grayscale ramp is remapped to the new range. You can keep the intensity window size fixed but move its position on the gray scale by dragging the middle arrow. The ramp is applicable to both grayscale and pseudo-color colormaps.

For color images, you can display them using a shared color map, which uses a stipple pattern to approximate colors. This allows multiple color images to be displayed correctly simultaneously. However, resolution is reduced because of the stipple effect. Alternatively, you can choose to use a private color map (using the image properties panel), which means the color map is set up so that only colors relevant to this particular image are stored. The effect is that the image is no longer stippled, so resolution is increased. But the side effect is that if there are other color images being displayed, their colors will not be correctly displayed. Switching between private and shared color maps on color images does not affect the display of grayscale images.

When images are displayed with the default settings, they can be all displayed correctly at the same time. For example, you can display any number of 8-bit, 16-bit, and color images simultaneously. However, if you have adjusted the intensity window differently on different images, or selected a private color map for a color image, then **WiT** can no longer correctly display all the images. You can load the correct colors for a particular image by clicking the left mouse button at the image.

### 3.2.4 User Defined Color Mapping

**WiT** allows you to selectively map a range of grayscales to a specified color. When you select **Custom** option for the **Colormap** item, **WiT** brings up a **Color Editor** window (Fig. [User Defined Color Mapping](#)).

Custom Color Panel  
(Figure omitted on purpose on demos)

If we take the `saturn` image as an example again, suppose we want to map the background (grayscale range 0-31) to a deep blue, and the range 104-143 to a bright yellow color. On the grayscale ramp on the left side of the **Color Editor**, move the window to (0, 31), then select **Add** from the **Edit** menu. A new entry will appear on the box on the right side of **Color Editor**. The entry will show the mapping **(000, 031) (000, 000, 192)**. Repeat the process for the range (104-143) on the grayscale ramp. Now click **Apply**. The `saturn` image will now be shown with a blue background with some yellow patches. This custom color map will be used for this image for as long as it is displayed. You can delete a particular color mapping by selecting (with left mouse button) an entry in the mapping window and then selecting **Delete** from the **Edit** menu. You can restore the original grayscale color map by clicking the **Remove** button.

The **File** menu allows you to save the mapping to a file and to read in a previously saved mapping file to apply to the current image.

### 3.3 General Objects

**WiT** can process objects other than images. The `images` directory contain an example named `general`. Use **View objects...** to bring up this object. The data will come up in a detached window, as when an image is displayed, but this time the window contains the content of the object as formatted text.

Many of the controls which apply to image windows also apply to general object windows. For example, you can change the size of the window, and the text will be rescaled to fit the window. It is rescaled such that horizontally, the number of columns displayed per line is constant regardless of how large the window is, but vertically, as many lines as can fit in the window are displayed. So if you want to make the characters larger, increase the window width but not the height. If you want to see more of the object while maintaining the character size, keep the width of the window constant and increase its height.

You can also pan the object vertically. Horizontal panning is not supported since general object data is formatted as a long column, so it is unlikely anyone would want to view only a portion of a line, when the entire width can be comfortably displayed. Likewise, area zooming is not supported because it is unlikely that anyone will want to magnify a portion of a line of text, or to shrink the text so that part of the window is unused. If you have a large (long) object and you want to see more of it, you can reduce the width of the window (but not the height) so that the text size becomes smaller.

### 3.4 Reading/Writing and Search Paths

When **WiT** reads an object, it uses a search path to look for it. Search paths allow frequently used objects to be found regardless of where your working directory is, and without you having to specify the complete path. Search paths will be discussed in detail in Section [Execution Environment](#).

Writing objects is handled differently. Objects and print files are always written to the `object` directory, which can be specified with the **Object dir...** item from the main panel **File** menu.

Igraphs are written to the `igraph` directory, which is set when **Save as...** is used from the **File** pull-down menu. See Section [Directories](#) for more information.

### 3.5 Object File Format

WiT objects are stored in an architecture independent format. This means that, for example, if you have WiT running on a SUN SPARCStation, someone who has WiT running on an i486 can give you some WiT object files, and you will be able to read them without any problems. Provided, of course, that the file is transmitted in a medium (tape, floppy disk, etc.) and medium format (tar, bar, FAT, etc.) that is supported by both systems. See Appendix [WiT File Format](#) for a description of the WiT object file format.



## Chapter 4 Data-Driven Mode

When you explain an algorithm to other people, or prepare documentation for it, you would probably draw a block diagram so that the algorithm can be *visualized*. It would be nice if such a diagram can actually be used for the *execution* of the algorithm. This is precisely what the data-driven mode of **WiT** provides. It allows you to specify an algorithm with a graph (block diagram), called an *Igraph*. Igraphs are made up of icons connected with links. Icons represent operators. Links specify the direction of data-flow through the operators, i.e. the order in which the operations are performed.

This data-flow environment is suitable for a variety of applications, the most notable of which is probably image processing. A typical image processing algorithm involves running images through a series of operations in a particular order. For example, you may want to smooth an image with a low-pass filter, threshold it at a certain grayscale, then extract the skeletal representation of the resulting blobs. The order of execution is as important as the operations themselves. This suits the data-flow environment perfectly.

In **WiT**, operators process data objects which are transmitted along links. An operator may modify an incoming object and then send it out, or simply consume it. The behavior of operators can be controlled with interactively modifiable parameters. Output objects in turn travel along links to be processed by other operators. Links can branch off into an unlimited number of directions. When an object reaches a branch junction, it multiplies itself and the individual branches can be executed *in parallel*, provided that you have more than one processor available. In addition, **WiT** provides flow control operators such as if-then-else and for-loop to allow even greater flexibility in the construction of large complicated graphs.

With meaningful graphical icons for operators and parameter values displayed directly below each operator, an Igraph is not only the means of executing an algorithm, it also serves as precise and accurate documentation for it.

### 4.1 Executing an Igraph

Let us start by reading and executing a simple Igraph. Select the **Load...** item from the **File** menu on the main panel. For this exercise, choose `simple` from the list. **WiT** will load the simple Igraph shown in Figure [A Simple Igraph](#).

A Simple Igraph  
(Figure omitted on purpose on demos)

This Igraph has two operators connected by a single link. Normally, an operator is shown as an icon with the operator name at the top, parameters at the bottom, input ports on the left, and output ports on the right. Links are always uni-directional, the direction of data-flow indicated by the arrow on the link. In this Igraph, the **rdObj** operator will read the object **saturn** and send it to the **display** operator for viewing. The **display** operator will use the name 'saturn' on its name stripe.

Bring up the **Run** menu from the top menu bar and select the **Start** item to execute the Igraph.

You will notice the status indicator below the **File** menu item changes from a mouse icon to a running-boy icon, indicating that the Igraph is now running. The **rdObj** operator will turn green momentarily, indicating that it is processing (on a monochrome screen, icons are highlighted with a stipple pattern). When the file is read, a data object is generated by the **rdObj** operator. This object is then sent to the **display** operator. You can actually see this object travel along the link! When **display** obtains its data, it begins execution and becomes highlighted in green. **Display** will bring up the saturn image in a detached window.

#### 4.1.1 Color Indicators

On a color display, in addition to the green color which indicates an operator is busy, other colors are used to indicate various states:

Color	Meaning
Red	selected
Green	busy
Blue	output blocked
Brown	waiting for user input

All colors are user configurable, see Section [Color](#) for more details. On displays which do not have enough colors (or monochrome), **WiT** will try to indicate different states as well as it can. For example, on a monochrome display, **WiT** uses different stipple patterns to indicate selected and busy operators.

#### 4.2 Data-Driven Options

You can configure the data-driven environment using the **Data-Driven Options** panel (Fig. [Data-Driven Options Panel](#)) from the **Options** menu item. The available controls are:

##### Data-Driven Options Panel (Figure omitted on purpose on demos)

###### **Scheduler** **Auto clear**

This is a rather advanced control. See Section [The Scheduler](#) for details.  
Controls whether object data is automatically cleared when the Igraph is started or when a new graph is loaded.  
When off, object data remain displayed until the window is closed by the user. This is useful when you want to compare results from different Igraphs.

<b>Snap</b>	<p>When snap is on, Igraph objects (icons or links) placed on the workspace will be snapped to the nearest grid point.</p> <p>Turning snap off will generally make it extremely difficult to create a neat looking graph. It is probably better to reduce the grid size rather than turn off snap when you are trying to fine position objects.</p>
<b>Grid</b>	<p>Set the desired grid size in pixels. In addition to improving readability, the grid is also used for snapping icons and links.</p> <p>Sometimes the visual presence of a grid can be annoying, particularly when the grid size is small. Hiding (choosing the <b>Hide</b> option) the grid maintains its snap function, but does not show it.</p>
<b>Pick tolerance</b>	<p>Specify the accuracy (in pixels) by which the user may pick an Igraph object such as a link or icon. The tolerance relates to how close in screen pixels the mouse arrow must be to an object before it is picked. Closely spaced icons, for example, require a smaller pick tolerance.</p> <p>The Igraph cursor will change to reflect the pick tolerance. Pick tolerance is <i>not</i> affected by settings of <b>Snap</b> or <b>Grid</b>.</p>
<b>Show Names</b>	<p>Controls how Igraph object instance names are displayed. See Section <a href="#">Object Names</a>.</p>

## 4.3 Run Controls

You can pause Igraph execution by bringing down the **Run** pull-down menu and selecting **Pause**. **WiT** may take a while to respond to your request, because it has to wait till all currently active operators have finished before it can pause. The run mode indicator will change from the running-boy to a small 'z' (sleep) immediately after you select the **Pause** item, then when all the active operators are finished, the run mode indicator changes to 'zZ' to indicate that the Igraph is now paused. You can resume execution by selecting **Continue** from the run menu. You can also stop execution altogether by selecting **Stop** while an Igraph is running. Again, **WiT** can stop only after all currently active operators are finished. So like pausing, the run mode indicator first changes to a stop sign, then back to the mouse sign when the Igraph is completely halted. You cannot **Continue** after a **Stop**.

## 4.4 More Example Igraphs

Besides the **simple** Igraph, there are many example Igraphs in the demo directory that will give you a good understanding of the kind of algorithms that you can construct with **WiT**. Some of these examples incorporate flow control operators such as if-then-else, for-loop, and the sequencer, which allow you to build much more complicated algorithms than the linear graph in **simple**. Load them like you loaded **simple** and see what they do. Try running with different speed settings (from **Setup** panel).

Following are brief explanations of what some of the example Igraphs do, and points of interest that they introduce.

### 4.4.1 If

Igraph `if`  
(Figure omitted on purpose on demos)

This Igraph (Figure [lgraph `if`](#)) is a good demonstration of how the **if** operator works, and how a data probe (the magnifying glass icon) can be used to display data. In this example, we read an image, threshold it, and send it to a **skeleton** operator. We continue to apply the **skeleton** operator until the number of pixels removed by **skeleton** becomes zero, at which point we display the final image. We have a probe on the **N** output of **skeleton** so that we can keep track of how many pixels are actually removed after each iteration. The probe inside the **if** loop allows us to see the image gradually reduced to its skeleton.

We could have used the **display** operator instead of the probe, but the probe is more compact in appearance, and, as we shall see in Section [Creating and Editing Links](#), it is also much easier to invoke.

This example also illustrates a convention which **WiT** uses when displaying detached object windows: objects with the same name are displayed in the same window. If it were not for this feature, we would have many windows each with a different count of pixels removed and numerous images of the various stages of the skeletonization.

#### 4.4.2 Sequence

Igraph `sequence`  
(Figure omitted on purpose on demos)

The `sequence` Igraph (Figure [lgraph `sequence`](#)) demonstrates the use of the **sequencer** and **collector** operators, as well as the use of parameters as inputs and the **graph** operator. The **sequencer** operator takes a vector of objects as input and sends the components out one by one to its output port. In this example, we first use the **ui** operator **dir** to read all files in the working object directory that match the specified pattern (`s*.wit`). This produces a vector of strings. The names are sequenced one at a time to the **rdObj** operator, which reads the image file, and sends it to the **zoomSize** operator to scale the image size to 64x64. The scaled images are then sent to the **stats** operator to compute an image variance. Notice that only one output of the **stats** operator is connected. This is allowed. Unconnected outputs are simply discarded. Finally, the individual variance (which is a single floating-point number) of each image is collected by the **collector** operator, which takes a number of simple objects as inputs and combines them into a vector on the output. Because **collector** must know when to stop collecting, it has a flag input at the lower port. As long as the flag is 0, it will keep on collecting. When the flag is 1, it forms the vector and sends the output. The **sequencer** has a flag output for the purpose of communicating with the **collector**. In our example, the vector of variances is sent to the **graph** operator for plotting. **Graph** is a powerful operator which enables you to plot multiple curves on

the same graph, select how each graph is displayed (line, bar chart, scatter plot, etc), and zoom in and out freely. See Section [Graph](#) for more information about **graph**.

Also of interest in this example are the **rdObj** and **display** operators connected to **sequencer**. Let us consider the **display** operator. We put it there because we want to see what each individual image looks like. However, if we simply specify a name for the **name** parameter of the **display** operator, all the images will go to the same window. In this Igraph, we want to see all the images at the same time, so instead of specifying a fixed value for the **name** parameter of display, we made it into an input port instead (see Section [Parameters as Inputs](#) for how it is done), and we connect the output of **sequencer** to this input. Parameter input ports always appear on the bottom of an icon. This way, every time **display** receives a new image from **rdObj**, it will also receive the corresponding file name from **sequencer**, so the images will be displayed in separate windows, with the appropriate names on the title bars.

#### 4.4.3 Hunter

Igraph `hunter`  
(Figure omitted on purpose on demos)

The `hunter` Igraph (Figure [Igraph `hunter`](#)) demonstrates color image processing and user defined operators. This algorithm involves operations on both color and grayscale images and demonstrates an interesting example of heuristic-based searching. The algorithm involves color thresholding ( **brightWhite** and **darkReddish** operators), binary segmentation ( **getBlobs**), and feature formation ( **getFeatures**). The **hunter** operator is an example of a user-defined operator specifically designed for this algorithm. It accepts as input two sets of feature vectors. One set describes all the bright white objects in the color scene whereas the second set describes all dark red objects. By stepping through the two feature vectors, the **hunter** operator identifies a sailboat whenever a white object, the *sail*, is located above a dark narrow object, the *hull*, and both objects are comparable in size. User-defined operators are covered in detail in the *Programmer's Guide*.

The resulting match generates a circle which is sent on to an **overlay** operator to mark the sailboat in a grayscale representation of the color image computed by converting from **rgb** to **hsv** space.

#### 4.4.4 Collect

Igraph `collect`  
(Figure omitted on purpose on demos)

When a series of images is collected, such as from a video camera monitoring some moving object, it is convenient to store the images as a group, or vector. The **sequencer** and **collector**

operators make it easy to deal with image vectors. Moreover, the image control panel has special controls which allow you to quickly browse individual images from an image vector.

The 'collect' Igraph illustrates how the **collector** operator can be used without a matching **sequencer** operator. The Boolean operators **true** and **false** are used to supply the flags that **collector** needs to decide when to stop collecting.

This Igraph also demonstrates how the image vector controls in the image properties panel work. When the final image is brought up by **display**, bring up the image properties panel. Notice that the **Image index** slider is now active (not stippled gray). Displaying an image involves a fair amount of computation. When viewing a vector of images, it is often helpful to be able to quickly show individual images within the vector (like animation). For this reason, **WiT** uses a cache to speed up image display from a vector of images.

Normally **WiT** loads images into the cache only on demand. But you can force all the images in the vector to be loaded by clicking the **Load all** button. With all images in the cache, you can use the slider for smooth animation! You should bear in mind that caching a large number of images do take up considerable memory and unless you are certain you want to view each image within an image view, you should let **WiT** load the cache as required.

The image scale is maintained across all elements in an image vector. For example, if you resize the image frame using the resize corners, you will see the **Custom scale** item in the image properties panel change to reflect the new scale. Now type in another image index. Notice that this new image is also taking on the scale that you have specified. Notice also the delay in displaying the image. The cache is emptied every time the scale is changed. If you want to scroll the images quickly at this new scale, you will have to click the **Load all** button again.

## 4.5 Building an Igraph

After you have tried all the demo Igraphs, you should have a fairly good idea of how powerful the Igraph concept is. Let us now build an Igraph from scratch, and see how *simple* it all is. As a practical example, let us build an Igraph to invert and then brighten an image, then save the result. Our algorithm consists of the following steps:

1. Read an image.
2. Invert.
3. Brighten.
4. Display and write result.

Let us start by clearing the workspace: select the **New** item from the **File** pull-down menu. We are now ready to create a new Igraph.

### 4.5.1 Creating Operator Instances

The first operation we need to perform is reading an image. To do this, bring up the **Operators** menu from the main panel. You will get a list of operator libraries available. Operators are stored

in different libraries based on function. Since reading an object is a read operation, we should expect to find a read operator in the **Read/Write** library. Select therefore the **Read/Write** menu item. A panel like Figure Read/Write Library should appear.

Read/Write Library  
(Figure omitted on purpose on demos)

All data types, images as well as other data structures, are treated as objects in **WiT**. So what we need now is the **rdObj** operator, which, as the name (and the graphics in the icon) implies, reads an object from a file. Invoke the **rdObj** operator by clicking (press then release) the left mouse button at the **rdObj** icon.

Now when you move your mouse in the workspace, you will see a copy of the icon moving along with the mouse. Move the icon to some place on the left hand side of the Igraph and drop it into place by clicking the left mouse button. This creates an instance of the **rdObj** operator. We will sometimes refer to operator instances on an Igraph as *nodes*, since that is a common term in describing graphs in general (in technical terminology, a graph consists of nodes connected by arcs). and you should get a panel like Figure Operator Properties Panel in Data-Driven Mode.

Operator Properties Panel in Data-Driven Mode  
(Figure omitted on purpose on demos)

This is the property panel for the **rdObj** operator. The **filename** field is the single parameter that **rdObj** requires, namely the name of the file that you want to read. In general, each operator has a number of parameters that you need to specify. Parameters are simple object types such as integers, characters or strings (they cannot be complex data structures). They also have a name (e.g. **filename** in this case). On the properties panel, each parameter is listed with its name followed by a data entry area such as a slider or choice list for you to enter a value.

There are other controls such as **Disable** in this operator panel, but for now, we are only interested in the **filename** item. Click the left mouse button on the **filename** field. Enter **saturn**.

Now click the **Apply** button (the **Reset** button cancels all data entered and takes down the properties panel). The properties panel will go down and you should see the parameter value **saturn** shown under the icon. We are now finished with step 1 of our algorithm.

After you have placed an icon on the workspace, you can move it around by pressing and holding the left mouse button while pointing at the icon. Now as you move your mouse, the icon will follow. The icon will be placed where you release the mouse button.

If you want to change the operator parameters, click the right mouse button at the icon, and the properties panel for that operator will be displayed again.

If you want to delete an operator, you need to select it first. **WiT** handles graphical objects in a consistent manner (see Section [Selecting Graphical Objects](#) and Section [Moving and Adjusting Graphical Objects](#) for details). An operator icon is a graphical object, since it has both shape and size. Currently the only other graphical object type allowable in an Igraph is a link, which we will come to later.

A graphical object is selected by clicking (press and then release) the left mouse button while pointing at it. The object turns red when it is selected (stippled on a monochrome). Selecting another object will cause the previous one to be deselected.

After some objects have been selected, you can choose the **Cut** item from the workspace menu to delete them. Bring up the workspace menu (cursor in workspace, press and hold right mouse button). The items below are available from the workspace menu in data-driven mode:

<b>Select</b>	Select workspace for printing (see Section <a href="#">Printing</a> ).
<b>Cut</b>	Delete the selected objects (operator or link) from the Igraph.
<b>Copy</b>	Copy selected objects to buffer.
<b>Duplicate</b>	Duplicate selected objects on Igraph (copy followed by paste).
<b>Paste</b>	Paste objects from buffer to Igraph.
<b>Undo</b>	Undo the last <i>edit</i> action. <b>Undo</b> only keeps track of graph editing events. This means that you cannot undo things such as parameter changes or loading in the wrong Igraph.
<b>View all</b>	Scale Igraph so that all objects are visible.
<b>Zoom/Pan...</b>	Bring up a zoom/pan control panel (See Section <a href="#">Zoom/pan</a> ).
<b>Redisplay</b>	Redisplay the Igraph in case it is not updated properly.

Some of these same items are also available from the main panel **Edit** pull-down menu. Which menu to use is merely a personal preference.

Now let us proceed on to step 2. We require an operator to invert an image. Select the **Point** library from the **Operators** menu. The **Point** pop-up panel will appear. Select **invert** and place this icon in the Igraph to the right of the **rdObj** icon. **Invert** does not require any parameters, so just click **Apply**.

Next we require an operator that brightens an image. This is done by adding a fixed amount to each pixel in the image. The **unaryOp** in the **Point** library will do this. Place **unaryOp** to the right of the **invert** icon. Define the **unaryOp** parameter to be addition (' +').

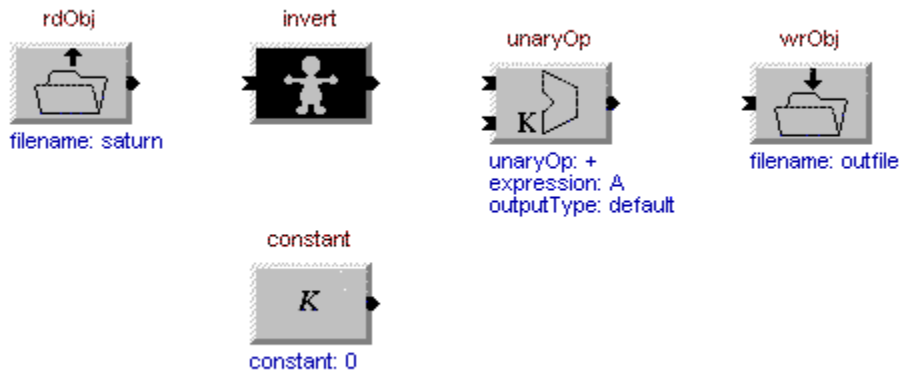
We will also need to supply an amount to brighten the image with. Select the **constant** operator



from the **Interactive** library, and place it on the lower left of **unaryOp** (later we will connect the output of **constant** to the 'K' input of **unaryOp**). Enter 20 for the **constant** parameter. Finally, create an instance of the **wrObj** operator to the right of **unaryOp**.

You should now have a set of operators laid out as in Figure Operators in Simple Example Igraph.

Operators in Simple Example Igraph



#### 4.5.2 Creating and Editing Links

Unconnected inputs and outputs of operators on an Igraph are denoted by arrows, called *ports*, on the left and right sides of icons respectively. For example, in the case of **rdObj**, no inputs are required, and one output is generated, whereas **wrObj** accepts one input and produces no output. We now need to connect the icons together with the proper data-flow. Start by pointing at the output port of the **rdObj** icon and *double click* the left mouse button. Now when you move your mouse, a rubber band wire will be created between the port and the cursor. Move the cursor to the input port of the **invert** icon and double click the left mouse button. The two ports attached to the link will disappear. They are now connected by the link and data will flow from the **rdObj** output to the **invert** input when the Igraph is executed.

There is another way to initiate and terminate a link without double clicking: by hitting the period ('.') key on the keyboard with the mouse pointing at the port you want to connect to. This alternate method is provided because some people find the rapid finger motions in double clicking difficult. Also, double clicking deselects all previously selected objects. So if you want to create a link without losing your current selection (very useful in tracing, for example. See Section Tracing), this method is invaluable.

You may create an arbitrary path for a link by clicking with the left mouse button every time you need a corner. Do not worry about making a 'pretty' wire as **WiT** will straighten it out for you once the wire is attached at both ends. Use the middle mouse button to backup. When you have backed up all the way to the port, the rubber band will be deactivated, and the link is canceled.

Wire the output port of **invert** to the top input port of **unaryOp**. Wire the output port of **constant** to the lower input port of **unaryOp**. Finally, wire the output of the **unaryOp** to the input port of **wrObj**.

We now need to tell **WiT** to display the final result before writing it out to a file. We could have used the **display** operator, but an easier and more compact way to do this is to add a probe to the link between the **unaryOp** and **wrObj**. To do this, click the right mouse button at about the mid-point of the link and a pop-up panel like Figure [Link Properties Panel](#) will appear.

### Link Properties Panel (Figure omitted on purpose on demos)

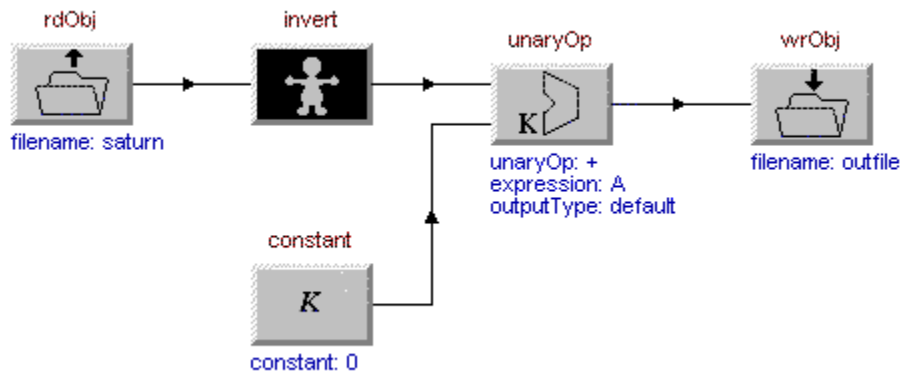
Erase the default name on the **Name** field and enter **inverted** (which will appear on the window title bar when the image is displayed), then click the **Yes** choice for **Probe**. A magnifying glass will appear on the link where you invoked the panel.

If you want to delete a probe, bring up the link properties panel and select the **No** choice for **Probe**.

If you delete the probe now, you will see that the name 'inverted' is still displayed at the link arrow. The name 'inverted' is associated with the link, not just the probe. Default link names are not displayed on the Igraph, but user names are. If you do not want 'inverted' displayed, erase the name field from the properties panel. See Section [Object Names](#) for a more thorough discussion of object names on an Igraph.

Our Igraph is now complete, and should look something like Figure [Completed Simple Igraph Example](#). Select **start** from the **Run** menu and *watch* your algorithm work!

### Completed Simple Igraph Example



## 4.6 Splitting Links

Suppose we want to display the final result of our simple Igraph example as a surface plot as well as an image. Select the **surface** operator from the **Interactive** library, and place it somewhere above **wrObj**. Now we need to send the image from **unaryOp** to *both* **surface** and **wrObj**. To do this, point at the link about half way between the **unaryOp** and **wrObj** icons and *double click* the left mouse button. A rubber band wire is again created, but this time between the mouse pointer and a newly created *junction* on the link, shown as a large dot. Connect this link branch to the top input port of **surface**. You can have as many branches from a link as you want. This is how multiple instances of an object can be sent to different operators. Again, you can use the middle mouse button to back up from any incorrect corner points that you have created. When you back up all the way to the link junction, the junction is also deleted.

Now if you run your graph again, you will see the token branch off at the link junction, and the brightened image will be shown with a surface plot of the same image. For more information about the **surface** operator, see Section [Surface](#).

## 4.7 Editing Links

Links can be selected and deleted just like operators. However, when a link is deleted from a junction, **WiT** employs some intelligence to determine whether the junction can be deleted or not. If a junction has only two links connected to it after a link is deleted, and one link is pointing towards the junction and the other one is pointing away from it, then the junction can be deleted, otherwise it is retained. It is therefore possible for a junction to have only links bringing data to it but no links carrying them away, or *vice versa*. Such a construct of course will not run properly, but is a valid state during the Igraph design process.

You can interactively adjust (as opposed to moving) a link after it has been created. Move your cursor to a link corner which you want to modify. Make sure the link is *not* selected (otherwise the entire link will move as you drag your mouse). Notice the cursor is square. You only need to position the cursor so that the link corner is within the square. After positioning the cursor, click and hold the left mouse button. Now when you drag your mouse, the link corner will move with the cursor. When you adjust links, **WiT** no longer tries to make the wire pretty by forcing it to lie along grid lines or multiple of 90 degree angles.

It is also possible for a link to become disconnected from everything (for example, when you delete a node which has some connected links). Such dangling link ends are displayed with a frayed end on the Igraph. You can connect a link to a frayed end by double clicking the left mouse button at the frayed end to start wiring, or you can connect an operator port to it by simply dropping an operator icon on top of the link end, provided the direction of the operator port is proper. For example, you can replace **invert** with **flip** by deleting **invert** and then dropping **flip** in its place. The links on both ends of the operator will be automatically connected.

Another note about making links: the direction of a link is often implicitly determined by the kind of ports it is connected to. If you attempt to connect a link between two input ports, or between two output ports, **WiT** will warn you and refuse to make the link. When the direction of a newly created link cannot be determined (e.g. a branch between two links), the data is assumed to flow in the direction that the link is created, i.e. following the order in which the points on the

link are entered.

## 4.8 Moving Operators and Links

Adjusting links and moving operators and links follow the same general rules for all graphical objects. (see Section [Moving and Adjusting Graphical Objects](#) for details.) There is, however, one behavior which is specific to moving operators. When *one* operator with links connected is moved, all the links stretch with the movement of the operator. This allows you to shift an operator small amounts without having to reconnect all its ports. This behavior does not apply when multiple operators are moved.

## 4.9 Zooming and Panning the Workspace

The zoom and pan techniques that you tried with images in Section [Resizing Images](#) also apply to the workspace. In addition, if you select the **Zoom/pan...** item from the workspace pop-up menu, you will get a panel for incremental zooming and panning. Details about this panel can be found in Section [Zoom/pan](#).

## 4.10 Saving and Deleting Igraphs

Save the Igraph by selecting the **Save as...** item from the **File** pull-down menu. This will bring up the file dialog. Enter a name for the new Igraph or choosing an existing file to overwrite, then hit the **Apply** button. Igraph names have a `.igr` extension on disk. **WiT** adds the extension automatically to the name you enter, so do not type that yourself. When you are editing an existing Igraph, you normally use the **Save** item, and **WiT** will quietly save your Igraph. If you use **Save as** to save an Igraph, **WiT** will check if that Igraph already exists and ask for your confirmation to overwrite the previous file if it exists.

An Igraph is stored with most of the editing options that are used at the time the Igraph is saved. These are:

- Workspace window size.
- Grid size.
- Grid visibility.
- Pick tolerance.

With this information, **WiT** can restore the (possibly different) working environments that you used to create different Igraphs. For example, you probably want to use a dense but invisible grid to design a very complex Igraph with many small objects, but a larger visible grid for simple Igraphs.

If there are old Igraphs that you want to delete, you should do it using the **Delete** panel, which is invoked by selecting the **Delete** item from the file pull-down menu. Do *not* delete Igraphs from a shell or file manager. **WiT** needs to keep track of related Igraphs and hierarchical operators (discussed later), if you do not use the **Delete** panel to delete an Igraph, **WiT** would not know

about it and may have problems later.

Although the **Delete** panel is primarily for deleting Igraphs, you can also use it to delete other **WiT** files, such as images or other data objects. Unlike Igraphs however, you can delete **WiT** data files from a shell or file manager without any ill effects.

## 4.11 Getting Help

### 4.11.1 Getting Help for Operators

You can get help for an operator by pointing at the operator icon in the operator library panel and hitting the **Help** key. You can also bring up the help window by pointing at an instance of the operator in an Igraph.

### 4.11.2 Getting Help for an Operator Port

Sometimes an operator has so many ports that it is not obvious what each port does. Usually the graphics in the icon should indicate the functions of the ports, but when there are many ports, attempting to show port functions graphically may result in extremely complex looking icons, which would defeat the purpose of icons, which should be something simple and easy to recognize. For this reason, a help window can be invoked for an individual port by pointing the mouse at the port and hitting the **Help** key. The help window will tell you the port number and name, and the type of object that the port is expected to receive or produce.

### 4.11.3 Getting Help for Igraphs and Links

You can also get help for what an Igraph does by hitting the **Help** key at an empty spot on an Igraph. Similarly, if you hit **Help** on a link, you will get general information about links.

## 4.12 Operator Selector

Often you know the name of the operator you wish to execute, but you are not sure which library it is in. Sometimes you only remember part of name of the operator you want. Going through the graphical operator library panels to locate such an operator can be a frustrating task. The **Operator Selector** panel is provided to address this requirement. Invoke this panel from the **Operators** pull-down menu. You will see a panel like the one shown in Figure [Operator Selector Panel](#).

Operator Selector Panel  
(Figure omitted on purpose on demos)

This panel provides many useful ways to select an operator:

- On the left list, you can select a library you are interested in, or you can choose to look at *all* libraries at the same time. Once a choice is made by clicking the mouse at the appropriate item on this list, the corresponding list of operators is shown on the right list.

- Double-clicking on an item on the right list launches the operator, or you can select an operator first by clicking once and then click the **Apply** button.
- Regardless of the library selection, you can type the name of the operator you want on the **Name** line. You can use the '\*' wild-card on the **Name** line. When you hit return, operators within the library currently selected which match the name typed are shown in the right list.

Notice the file name entered on the **Name** field is *always* in effect. It is a common mistake to leave some specific name on this field while switching libraries looking for an operator. If you do this, you may find the operator list on the right appear unusually short! So be sure that the **Name** field is set to '\*' if you want to see all operators listed.

The **Filter** item allows us to display only primitive or hierarchical (Section [Hierarchical Operators](#)) operators. For now, just make sure that **Filter** is set to **All**, which means that all operators will be shown.

The functions of the buttons on the right hand side are:

<b>Help...</b>	Help information (see Section <a href="#">Getting Help</a> ).
<b>Edit...</b>	Edit the icon (see Section <a href="#">Icon Editor</a> ).
<b>Delete...</b>	Delete the operator from <b>WiT</b> , use caution!
<b>Apply</b>	Launch operator in <b>Name</b> field.
<b>Cancel</b>	Bring down window without launching any operator.

## 4.13 The Graph Menu

The **Graph** menu (from the main panel **Graph** button) consists of controls applicable only to Igraphs. The meaning of its items have been listed in Section [The Main Panel](#).

### 4.13.1 Group Properties

Sometimes it is desirable to disable whole sections of an Igraph, or set a number of links to a thicker width to emphasize a data path. Such collective changes can be done easily from the **Node group properties** and **Link group properties** panel (Figure [Node and Link Group Properties Panels](#)), invoked from the main panel **Graph** button. The attributes are the same for individual operators or links, the only exception is that you can choose the **As is** option to preserve the original attributes.

### Node and Link Group Properties Panels

(Figure omitted on purpose on demos)

### 4.13.2 Tracing

Igraphs tend to get larger and larger. Particularly because **WiT** can be used to control hardware where Igraph operators can be made to represent small operations in order to provide more graphical programming flexibility. When Igraphs contains hundreds of operators and links, with parallel data-flow, it can become difficult to trace how data will travel on the graph without actually running it. The trace utility from the **Graph** menu provides a convenient way of studying data paths in a large Igraph.

For example, in an Igraph with a large cross-bar switch, it is useful to be able to select some links or operators and trace what other operators are related in terms of data flow. Tracing even works across sub-graphs. So you can select an operator at the top level Igraph, which contains hierarchical operators. When you trace the operator, and if the data path related to that operator involves any of the sub-graphs, at any level, all such items, regardless of whether the sub-graphs are currently displayed or not, will be put on the selected list (and therefore highlighted).

In wiring crossbars, it is desirable to use the tracing function to locate which ports of the crossbar are to be connected. However, if you start the connection by double clicking at one of the ports, all selected objects (i.e. traced objects) become dehighlighted, so the port that you want to connect to is no longer highlighted. You can easily avoid this problem by using `.` on the keyboard to start and terminate the wiring (see Section [Creating and Editing Links](#)).

When tracing outputs, all outputs are followed for primitive operators. For hierarchical operators, the underlying graph structure is used to trace only those outputs related to the selected objects.

When tracing inputs, if the operator is fire-on-any (see Section [Firing Strategy](#)), then the input is traced only if there is only one input connected. If the operator is fire-on-all, then all inputs are traced.

## Chapter 5 Advanced Igraph Building

### 5.1 Operator Properties

We have seen a simple use of the node properties panel in Section [Creating Operator Instances](#). Now we can describe this panel in more detail:

<b>Operator</b>	<p>By default, when you create an operator instance on an Igraph, the first instance of that operator will have the same name as the name of the operator shown on the library. We will refer to the name of the operator on the library as the <i>operator name</i>, and the name of the operator on the Igraph as the <i>instance name</i>. If you have more than one instance of one operator in an Igraph, subsequent instances will have a number appended to the operator name, e.g. <b>rdObj #1</b>, <b>rdObj #2</b>, etc.</p> <p>The <b>Operator</b> control allows you to override this default behavior. For example, you may want to name some of your <b>rdObj</b> operators as <b>readXRayImage</b>, <b>readProfileData</b>, etc., so that the Igraph is easier to understand, and you can refer to a particular instance of <b>rdObj</b> in your documentation. See Section <a href="#">Object Names</a> for even more ways to control instance names.</p>
<b>Execute</b>	<p>A <b>Disabled</b> means that although the operator is shown on the Igraph, it will not be executed. For example, you may have an Igraph with multiple starting nodes. By disabling a starting node, you can disable a whole branch of the Igraph.</p> <p>A disabled operator functions as though it is not on the Igraph at all. Tokens sent to a disabled operator will stop there. Disabled operators are shown with a cross-hatch pattern.</p> <p><b>Bypass</b> means the operator performs no function, and yet allows tokens to pass through. For example, you may have a low-pass filter in an Igraph, and you want to see what the result will be if the filter is taken out. Only operators which have the same number of inputs and outputs <i>and</i> the input and output types have a one-to-one correspondence can be bypassed. Bypassed operators are shown with a large arrow superimposed on the icon.</p>
<b>Flip</b>	<p>Flip the icon in the X or Y direction, or both. This is useful when you have loop-backs or branches coming from the top of an icon.</p>

**Apply** and **Cancel** are self-explanatory. **Promote** will be discussed in Section [Promoting Parameters](#).

For each operator parameter, a choice control is presented to the left which allows you to control whether a parameter is displayed with its name and value ( **All**, value only ( **Value**), or not displayed at all ( **None**). The **Input** choice will be discussed in Section [Parameters as Inputs](#).

### 5.2 Link Properties

The link properties panel was first introduced in Section [Creating and Editing Links](#). Now we can describe this panel in more detail:



<b>Name</b>	Change the name of a link. See Section <a href="#">Object Names</a> for more information about object names.
<b>Execute</b>	Enable/disable a link. Disabled links are shown as a dashed line. No data flows along a disabled link.
<b>Width</b>	Set how thick a link should be drawn . This can be useful if you want to emphasize certain links. Link thickness does not affect Igraph execution in any way.
<b>Probe</b>	If <b>Yes</b> is selected, a probe is place at where you clicked your mouse to bring up the properties panel. If you want to move the probe, you have to select <b>No</b> first and <b>Apply</b> the properties. Then click the right mouse button on the position of the link that you want the probe to be, and select <b>Yes</b> for <b>Probe</b> .

**Apply** and **Cancel** are self-explanatory. **Promote** will be discussed in Section [Promoting Parameters](#).

### 5.3 Customizing the Grid

The grid that is shown on the workspace is not just for decoration only. When snap is enabled (from the **Data Driven Options** panel), objects will snap to a grid point when placed or moved on the workspace. When making a link, however, operator ports and link corners have priority over the grid. For example, if you branch off from a link in the middle of one of its link segments, then **WiT** will choose to break the segment at a grid point. But if you branch off from near a link corner, **WiT** will break the segment right at the corner. This usually results in a better looking Igraph.

If you want to position objects at a different resolution, you can change the grid size from the **Data Driven Options** panel. If you set the grid size to 0, the grid will not be displayed at all.

Sometimes you may find the grid distracting, so you can **Hide** the grid. The effects of snapping are still enabled even when the grid is hidden.

### 5.4 Pick Tolerance

When you need to select something on the Igraph, the size of the square cursor tells you how accurate you need to be. You can change the pick tolerance from the **Data Driven Options** panel. When you change the pick tolerance, the size of the square cursor will change accordingly.

When objects are close together, you might want to lower the pick tolerance to allow finer, pin-point control. Conversely, you may set a coarse pick tolerance on sparse Igraphs so that it is easier to pick things. The pick tolerance is not related to snap or the grid size.

### 5.5 Object Names

Every node and link on an Igraph must have a name. By default, the first instance of an operator is given the operator name . Any more occurrences of the same operator will have a number

appended to the basic operator name. You can rename an operator to whatever name you want, provided that same name is not already in use. If it is, then a number will be appended to the name to make it unique. If you delete the name altogether, **WiT** automatically replaces it with the default invisible base name: a single underscore character ('\_').

Most printable characters can be used in instance names, including space (' '), but you should avoid the backslash ('\, used internally for hierarchical information) and the number ('#', used for instance numbers) characters. Also, **WiT** does not display any name that starts with the underscore character ('\_'). So do not use it as the first character of a name if you want that name displayed.

Link names are assigned following the same rules. The only exception is that by default, link names are not displayed (they start with an underscore).

By default, instance numbers are shown. However, when dealing with large Igraphs, instance numbers can make the Igraph look cluttered and are best turned off. In addition to specifying individual instance names, you can control how instance names are generally displayed using the **Show names** control item on the **Data-driven Options** panel. They can be turned off completely (**None**), with only the base name showing (**Base**), or with the instance number showing as well (**Instance**).

## 5.6 Cut and Paste Across Top Level Igraphs

Sometimes you may want to incorporate in part or all of one Igraph into another. This is easily done because the copy buffer (filled when you copy something using the **Copy** item from the workspace pop-up menu) is preserved when you load a new Igraph. So all you need to do is load the Igraph that you want to copy from, select the operators and links you want, and copy them. Then load the Igraph you want to copy the objects to (or clear the workspace for a new Igraph), and paste the objects. When you paste, **WiT** automatically goes into drag mode so that the objects can be placed interactively.

## 5.7 Parameters as Inputs

Often it is necessary to redefine an operator parameter as an input. For example, you may want to set the filter width of a low pass filter dynamically depending on the contents of the image itself. So you would like the filter width to be an input rather than a fixed value. Such an input could then be connected to a link to get its data. **WiT** will produce an input port for a parameter if the parameter field in the property panel of the operator is empty, or if the **Input** choice is selected for the parameter. Parameter input ports are created on the bottom of the operator icon.

## 5.8 Hierarchical Operators

**WiT** allows you to make an Igraph into a new operator, which can then be incorporated in another Igraph. Such operators will be referred to as either hierarchical or derived operators. You can use hierarchical operators to group a series of operations that you commonly use, or to make a large Igraph easier to handle and understand. Hierarchical operators can be compared to procedures in a programming language. If a user is not interested in the internal structure of a hierarchical operator, he can simply use the operator like he would any other operator.

An example of hierarchical operator usage can be found in the `hierEg` example Igraph in the `demo` directory.

### Hierarchical Operator Usage Example (Figure omitted on purpose on demos)

Load this Igraph, and notice the **derivedHipass** operator has a red dot on its upper right corner (see Figure [Hierarchical Operator Usage Example](#)). This is how derived operators are shown on the Igraph. If you want to see what is contained in **derivedHipass**, bring up its properties panel. You will see that in addition to the usual parameters and buttons, there is another button, labeled **Expand**. Click this button, and you will get a pop-up window with an Igraph in it. This is the Igraph contained in **derivedHipass**. **DerivedHipass** will be referred to as the *super-node* of this graph, while the Igraph will be referred to as the *sub-graph* of **derivedHipass**.

Notice that on the sub-graph, there is an input and an output port which look like the ordinary ports on an operator, but are hollow and a bit larger. These ports correspond to the ports that are on the super-node. Try running the Igraph with the sub-graph displayed, and with animation turned on (select **Walk** or **Run** speed from the **Setup** panel). You will see the token traveling to the input port of **derivedHipass**, and then the token will appear at the input port of the sub-graph. After going through all the operations in the sub-graph, the token will reach the output port on the sub-graph, and then it will emerge from the output of **derivedHipass**.

We will discuss how hierarchical operators are created in Section [Making a New Hierarchical Operator](#).

#### 5.8.1 Changing Operator Parameters in a Sub-Graph

You might have noticed that on the sub-graph of `hierEg`, the parameters of **lopass2d** are shown in reverse. This indicates that these parameters have been *promoted* to the upper level, i.e., they are linked to the parameters of the super-node. In this example, the **width** and **height** parameters of **derivedHipass** are linked to **width** and **height** of **lopass2d**. Try to change **width** of **derivedHipass** to some other value, with the sub-graph displayed. You will see that the **width** parameter of **lopass2d** in the sub-graph also changes to the new value that you entered.

A sub-graph can in turn contain hierarchical operators. There is no limit to the number of levels on an Igraph. There is also no limit to the number of input or output ports on a sub-graph, or the number of promoted parameters.

Even if the parameters in a sub-graph are not promoted, you can still change them by bringing up the properties panel of the node in the sub-graph whose parameters you want to change, and making the changes there.

Since it is possible that hierarchical operators are incorporated in many Igraphs (just like procedures can be called from different programs), the changes that you make to either promoted or unpromoted parameters will only affect that particular invocation of that hierarchical operator. On the other hand, if you actually load and edit the Igraph associated with the hierarchical operator itself, and change some parameters there, then every invocation of that operator will be

affected, unless the parameter values have been overridden.

### 5.8.2 Modifying a Sub-Graph

You are *not* allowed to modify the existing *structure* of a sub-graph. For example, you cannot delete a node from a sub-graph, or change the path of a link. The reason is that sub-graphs can potentially be incorporated into many Igraphs, so modifying a sub-graph may lead to unexpected behavior in other Igraphs. If you really want to modify a sub-graph, you have to exit the current top level Igraph and read the sub-graph in as the top level Igraph (using **Load** from the **File** menu).

Even though you cannot delete or copy objects in a sub-graph, you are allowed to select objects in a sub-graph, and they will be highlighted just as when the objects are in the top level Igraph. This is useful for example when you have a complex link network and you want to see the path of a particular link.

Moreover, you are allowed to *add* links (but not operators) to a sub-graph. Links thus added only apply to one particular instance of the sub-graph. For example, suppose you have two Igraphs, *G1* and *G2*, both of which include a sub-graph *SG*. Suppose you load *G1* as the top level Igraph, then add some links to the *SG* sub-graph and save *G1*. The instance of *SG* which is part of *G2* will *not* have the new link.

Such sub-links can be edited like top level links. They can be moved, adjusted, and deleted.

### 5.8.3 Copy, Paste, and Undo Across Sub-Graphs

You can copy objects from one sub-graph and paste it to another (including the top level Igraph). If you have selected objects from different Igraphs and then execute a copy, duplicate, or cut, the action only applies to the objects within the Igraph which you invoked the action. But you can copy objects from one graph and paste it in another. You are only allowed to paste *operators* to the top level Igraph, not sub-graphs.

### 5.8.4 Making a New Hierarchical Operator

Creating a hierarchical operator is simple. Select the **Make operator** item from the main panel **Graph** button. You will be prompted with a panel like Figure Make Operator Panel.

Make Operator Panel  
(Figure omitted on purpose on demos)

The size of the operator icon is automatically set so that all input and output ports can comfortably fit on the left and right of the icon respectively. The location of these ports are also computed automatically, and follows the order in which they appear in the Igraph. You can alter the default icon size if you want, but normally you should not reduce the height of the icon, since that can leave insufficient room for the placement of ports.

Select the operator library that you want the new operator to be put in, then click **Apply**. A new

operator, with the same name as the Igraph that is currently being edited, will then appear in the library that you specified.

If you want to delete an operator from a library, move the mouse inside the library panel and to the operator you want to delete, then bring up the pop-up menu (right mouse button). Select the **delete** item. You will be given a warning before **WiT** actually removes the operator. The corresponding Igraph will *not* be removed, though.

Sub-graph input and output ports are operators found in the **Dataflow** library. You should try to give ports meaningful names, since they will be used to label the super-node ports, and will come up on the help panel when the user asks for help.

The location of the ports on the super-node is determined by their relative vertical positions in the sub-graph. Input ports will be placed on the left, output ports on the right. If two ports of the same type are at the same vertical position, then the one on the left (smaller horizontal coordinate) will be assigned first.

**WiT** tries to set the port positions as wide apart and evenly distributed on each icon side. In most cases, you should check the port positions, and change them if necessary. Bring up the pop-up menu from the library panel that the new operator is placed, and select the **Edit icon** item. A new window will come up showing a large version of the operator icon with a grid and the ports at their current positions. You can change the port positions by dragging them. When you are finished, click **Apply**. See Section [Icon Editor](#) for details about the icon editor.

When you modify a hierarchical Igraph, **WiT** will maintain the positions of all previously defined port positions. Only new ports will be given automatically computed positions.

### 5.8.5 Promoting Parameters

We have so far avoided the mention of the **Promote** button in the operator property panels. When you click this button, another panel will come up for you to enter the name(s) of the promoted parameters, to be used on the super-node. If you leave the name field blank for a parameter, then that parameter will not be promoted. If that parameter was previously promoted, it will be un-promoted.

## Chapter 6 Data-Flow Model

**WiT** uses the data-flow model so that algorithms can be constructed as intuitive block diagrams and can run in parallel when the computer hardware is capable. However, because most computer languages are control driven, the data-flow model may be foreign to most users. In this chapter, we will take a more analytical look at the data-flow model that **WiT** uses. Such knowledge may be necessary in the design of more advanced Igraphs.

Data-flow is a common concept used especially in the area of parallel processing computer architecture. In the data-flow model, an algorithm is expressed in terms of operators that process input data and produce output data. An operator may be a simple procedure, such as addition or subtraction, or it may be an intricate function consisting of thousands of lines of code. An operator becomes ready for execution when all its input objects (called tokens) are available. It is this property that makes the data-flow model ideal for describing parallel algorithms. At any time, there may be many operators ready for execution. If there is an infinite supply of processors, then each operator can be mapped to a unique processor, and all the operations that are ready can proceed in parallel. In practice, the degree of parallelism is limited by the number of processors available, and how efficient the mechanism is that maps operators to processors. In **WiT**, you can install multiple servers (of potentially different architectures consisting of special hardware, more about this later), in which case the internal scheduler in **WiT** will take care to keep as many of the servers as busy as possible, thus fully utilizing your networked resources and the inherent parallelism in your algorithm.

When an Igraph starts running, only operators that do not require any input tokens, such as **rdObj**, are ready for execution. After an operator finishes processing its inputs, it may generate one or more output tokens, which will be sent along the links down the execution chain, and the input tokens will be consumed. A token can be sent to the input port of a following operator only if that port is empty. If an operator cannot send all its outputs successfully to all its descendants because some of the ports are full, it is blocked until the descendants are ready to receive the new tokens.

An Igraph execution is considered complete if there are no more operators ready for execution and all the servers are idle, which means that no tokens can be produced that may unblock any blocked operators. With animation turned on (**walk** or **run** speed), you can easily see where the Igraph execution is blocked (if it is) by the location of any unconsumed tokens.

### 6.1 The Scheduler

**WiT** uses a scheduler to decide which operators to run on which servers to maximize parallelism but maintain correct data-flow. Some operators are always run on the GUI program. These include all operators in the **dataflow** and **interactive** libraries. **Dataflow** library operators must run on the GUI because they are intimately tied to the scheduler. **Interactive** library operators either need access to the GUI display, or are fine grain operations which take little time to execute.

#### 6.1.1 Data transport

Token data generally are not transported between the GUI and servers or among the servers themselves unless it is necessary. Often an entire algorithm, including input/output operations,

can be executed on a single server without *any* data transfer at all. Data transfer is necessary when the object needs to be displayed, or when the server that has data does not support a required operation.

When multiple servers are involved, the scheduler maintains information about where each data object resides. When an operator needs to be scheduled, the scheduler assigns it to the available (not busy) server which requires the least amount of data transfer.

When a link branches into parallel branches, copies of the token data will be physically copied to multiple servers. But after this initial copying, the servers will be able to execute in parallel.

### 6.1.2 Firing Strategy

In general, operators are executed in the order in which they become ready. But if they become ready at the same time, then the scheduler is free to select whichever to execute first. Also, for efficiency and implementation reasons, UI operators are always scheduled before server operators, even if the server operators become ready before the UI operators.

An operator becomes ready whenever all its inputs are available (see Section Fire-On-Any Operators for exceptions). After execution, it tries to send all its outputs to its descendants. However, it may not be successful because some of its descendants may still be processing previous inputs, and are not ready to accept any new inputs. In this case the ancestor operator becomes blocked. When an operator has finished execution, it will clear its inputs and try to wake up any direct ancestors that were previously blocked.

You can have more than one output connected to the input of an operator. In this case, whichever token arrives first at the port will be accepted, and all other tokens will be blocked until the operator has consumed the first token.

### 6.1.3 Flat and Hierarchical Scheduling

The **Data-Driven Options** panel has a control for selecting between flat and hierarchical scheduling. This distinction only applies when an Igraph contains hierarchical operators.

In flat scheduling mode, the scheduler is free to choose any of the ready operators to execute, regardless of which Igraph the operators belong to. Hierarchical operators therefore act as though they are Igraphs shrunk to the size of icons. Not only do data tokens enter hierarchical operators and get executed immediately, outputs from hierarchical operators are sent downstream as soon as they become available.

In hierarchical scheduling mode, hierarchical operators behave as though they are primitive operators. In other words, data tokens tend to be collected before they get passed inside the underlying sub-graph to be processed. Also, output tokens are sent downstream only when all that can be produced from the sub-graph has been generated. This behavior is due to the fact that, in hierarchical scheduling, each sub-graph is considered as a distinct unit. Once scheduling has started in an Igraph, only operators from that Igraph are scheduled until there are no more operators to schedule from that Igraph.

The reason for providing two scheduling modes is that flat scheduling maximizes parallelism, whereas hierarchical scheduling executes functionally related operators in groups, which is

important in generating readable program code.

## 6.2 / Extensions to Basic Data-Flow Model

In order to address the different requirements of diverse algorithms, **WiT** provides some useful variations to the basic data-flow model.

### 6.2.1 Unconnected Output Ports

If any output of an operator is not connected, then the operator will not block on these output ports. This makes it easier to construct Igraphs when you do not need one or more of the outputs from some of the operators. See the `sequence` Igraph for an example of unconnected outputs.

### 6.2.2 Fire-On-Any Operators

Some operators are best implemented if they fire as soon as *any* one of their inputs are ready, as opposed to waiting until all the inputs have arrived. An example is the **memory** operation in the **interactive** library. **Memory** takes either one of two inputs: an object and a flag. When the object input is fed to **memory**, it stores the object while sending the object to the output at the same time. If the flag input is fed, then **memory** recalls the stored object and sends it to the output.

If more than one input arrives at the same time to a fire-on-any operator, the tokens are consumed one by one, starting from the top port.

### 6.2.3 Sync Tokens

When **WiT** is used to control external hardware, pure data-flow is not adequate. For example, often you need to initialize the hardware before it can be used. The initialization can be represented as **WiT** operators. Because hardware devices can often run in parallel, you would like to take advantage of the parallelism on the Igraph. So you would want the image processing to start only after *all* the initialization operators are finished.

You can create a null operator for this purpose. For example, if you have three hardware devices, you will have three parallel initialization branches, whose output you can send to a null operator that accepts three inputs to produce one output. The output is there simply to tell the rest of the Igraph that the hardware is now ready (we can depend on the fact that, in data-flow, an operator fires only after all its inputs have arrived). However, depending on the number of branches you want to merge, you will need null operators that accept different numbers of inputs. Our work will be easier and our Igraph more elegant if we can somehow tell a link junction (there are no limits to the number of links connected to a junction) to wait for all its input links to bring in a token before proceeding.

A special data token type called a **sync** token has been created for this purpose. **Sync** tokens do not carry any data, and they are treated specially by link junctions. If the tokens sent to a junction are **sync** tokens, then the junction will not propagate the token until *each* input link has brought in a **sync** token. If any of the links connected to the junction has been disabled (see Chapter [Advanced Igraph Building](#)), then the junction will not wait for a **sync** token to come from the disabled link. In other words, disabled links behave as if they have been deleted as far as **sync** behavior is concerned.



When all the input links connected to a junction have brought in a **sync** token, the **sync** token (one only) is broadcast to all the output links connected to the junction.

You should never send **sync** tokens and regular data tokens to the same junction. There is no logical interpretation of how the junction should behave in such a construction.

### 6.3 Some Caveats of Data-Flow

Data-flow is a natural way of expressing parallel algorithms. However, most of us are much more familiar with the traditional control-flow model, where all data is stored in a commonly accessible area, and we specify the sequence of operations to be performed on the data. Almost all common programming languages, such as C, Pascal, or FORTRAN, work this way. In this paradigm, operations are performed on the data regardless of whether the data is valid or not. If the execution sequence is correct, then the data will be valid at the right time.

If you are not familiar with data-flow, you may find the behavior of **WiT** sometimes strange, but after you have a good understanding of the data-flow concept, you should find that it all makes sense and it is easy to construct data-flow algorithms in **WiT**. Some caveats of data-flow that you may encounter are explained in this section.

#### 6.3.1 Deadlock

One of the first problems you may encounter while building your own Igraphs is that sometimes execution is blocked before the desired outputs are produced. For example, suppose we want to overlay a small image (say a company logo) onto a list of images. We can use the **sequencer** and the **dir** operator to form the image list, and use **aluOp** to do the overlay. Afterwards, we want to display the results, as illustrated in the Figure [Example of Deadlock in an Igraph](#) (Igraph `deadlock` in `demo` directory).

Example of Deadlock in an Igraph  
(Figure omitted on purpose on demos)

Deadlock Fixed  
(Figure omitted on purpose on demos)

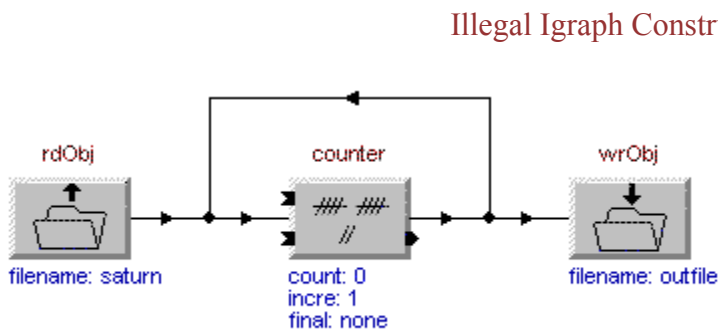
We used the file names as inputs to **display** so that the results will be shown in different windows.

However, this Igraph will stop after the first image is processed. With animation turned on, you will see that **aluOp** has one token waiting on the upper input. This means that **aluOp** is blocked because it needs a token on the lower port before it can fire. The logo token has been consumed when the first image was processed, and is no longer available. You may think that perhaps we

should not consume input tokens, but this is the only way that an operator can signify to its ancestors that it is ready to process new tokens, and to eliminate unnecessary tokens, which require memory to store. The correct solution is to use the **memory** operator, which retains its output token so that it can be recalled later. The correct Igraph is shown in Figure Deadlock Fixed (Igraph `deadlockFixed` in demo directory).

### 6.3.2 Connecting an Operator to Itself

In general, loops are perfectly acceptable in an Igraph. However, you cannot connect an output of an operator directly (i.e. no other operators in the loop) to one of its own inputs. An example of such *unsupported* construct is shown in Figure Illegal Igraph Constructs. Such a construct is a problem because operator execution conceptually proceeds as follows:



1. The inputs are used to produce the outputs, if there are errors, exit.
2. Try to send the outputs to all descendants.
3. If successful, goto 4, else goto 2.
4. Delete the inputs and exit.

If you connect an output directly to one of the inputs, then step 2 will always fail, since a token can only be sent to a port if that port is empty, but we are sending the token to our own input ports, and the input ports are not cleared until step 4 is executed. In practice, you should not do this anyway since that would result in an infinite loop. If you really want an infinite loop (for use as a free running demo, for example), then you should do something in the loop. See the `if-demo` Igraph for an example.

## Chapter 7 Demand-Driven Mode

Although Igraphs are extremely powerful and easy to use, there are times when you just want to apply one or two operators to some images. For example, you may want to read an image and transpose it, then save it back. Having to build a tiny graph and then run it is inconvenient. So **WiT** provides a demand-driven mode of execution. In this mode, operators that you select are executed and the results shown immediately. Demand-driven mode is useful in exploring and understanding the rich set of imaging operations **WiT** offers, as well as debugging any new operators that you have developed yourself.

You can switch between data-driven and demand-driven mode interactively from the **WiT Setup** panel. In demand-driven mode, there are no Igraphs. Instead, the workspace has a number of rectangular regions which look like the back of playing cards (see Figure [WiT in Demand-Driven Mode](#)).

### WiT in Demand-Driven Mode (Figure omitted on purpose on demos)

These cards are storage places for results (objects). As results are produced as a consequence of executing an operator, they are written to these cards in a stack fashion (last in first out).

#### 7.1 Reading and Displaying Images

Let us start by reading and displaying an image. Switch to demand-driven mode, and invoke the **rdObj** just like when in data-driven mode. You should get a panel like Figure [Operator Properties Panel in Demand-Driven Mode](#).

### Operator Properties Panel in Demand-Driven Mode (Figure omitted on purpose on demos)

Notice that some graph related parameters, such as **Flip** and **Disable**, are not present. For this example, enter **saturn** as the file name. When you click the **Apply** button, **rdObj** is executed immediately (instead of being placed on the Igraph), and the image should appear on the first card in the workspace. All objects are shown with a fixed standard size on a card. To get a look at the **saturn** image at the original resolution, click the left mouse button on the first card. The card will momentarily move to a lower position and then fall back to its original position (more about this behavior later), and a larger image of saturn will now appear in a detached window.

Notice the name **rdObj** shown on the title bar of the detached window. It is inherited from the name of the operator that produced this image. So if you want to read another image and have it displayed in another detached window, you need to change the operator instance name before you **Apply** it.

#### 7.2 Demand-Driven Mode Cards

The number of cards in the workspace is the number of memory cells available for displaying

intermediate results. Because **WiT** always searches from left to right for input objects when executing operators, you may want to move the cards around to control what input objects are to be used for the next operator. To move a card, point the cursor at the card you want and press and *hold* the left mouse button. The card will move to a lower position, and an arrow will appear to indicate the insertion point that the image will go to. With the left mouse button held down, move the cursor until the arrow is at the desired position, then release the button. Notice all the cards to the right of the inserted card are automatically pushed to the right. If you release the button while the arrow is pointing to the original position of the card (this is the same as clicking the left mouse button on the card), **WiT** will take that as a request to display the object in a detached window.

More control is available from the pop-up menu in the workspace (cursor in workspace, press and hold right mouse button):

<b>Select</b>	Select workspace for printing (see Section <a href="#">Printing</a> ).
<b>Cut</b>	Delete the first object.
<b>Duplicate</b>	Make a copy of the first object, and push all the objects down one slot.
<b>Clear</b>	Delete all objects.
<b>Zoom/Pan...</b>	Bring up a zoom/pan control panel (See Section <a href="#">Zoom/pan</a> ).
<b>Redisplay</b>	Redisplay the workspace.

**WiT** will not use a card more than once in the execution of an operator. If you want to feed the same object to an operator more than once, make a copy of the object using **Duplicate** from the workspace menu.

### 7.3 Executing a Series of Operators

When you execute an operator that requires one or more inputs (as opposed to parameters), **WiT** will search the current set of cards for an object of the appropriate type, starting from the left. When a new object is created, it becomes the first card in the workspace, and all previous cards are pushed one position to the right.

For an exercise, perform the following operations:

- Transpose the image ( **flip** in **Point** library).
- Add noise to it ( **addNoise** in **Point** library).
- Rotate it 30 degrees ( **rotate** in **Point** library).

## 7.4 Demand-Driven Options

You can customize the demand-driven environment with the **Demand Driven Options** panel (Fig. [Demand-Driven Options Panel](#)). Currently there is only one control:

### Demand-Driven Options Panel (Figure omitted on purpose on demos)

<b>No. registers</b>	Set the number of registers that are used when operating in demand-driven mode. More registers give you more flexibility in data manipulation, but require more memory for storage. This is an important consideration if your computer has a small amount of memory, particularly when large images are manipulated.
----------------------	---

## 7.5 Demand-Mode Scripts

Every imaging operation you perform (including moving the cards around) in demand-mode is automatically logged. Try doing a few simple operations, such as reading an image, apply some operations (e.g. take the transpose, low-pass filter), move the cards around, and apply some more operations. Now select the **Start** item from the **Run** menu, and watch the sequence of actions you just did being played back.

You can adjust the speed at which scripts are played back. See Section [Setup Panel](#) for details.

You can erase the recorded actions with the **New** item from the **File** menu, and you can save and read demand-mode scripts with **Save** and **Load** from the **File** menu. Demand-mode scripts are saved with a `.wdh` (which stands for WiT Demand History) extension.

## Chapter 8 General Tools and Techniques

Every effort has been made to make **WiT** a user-friendly program. Consequently, many common operations, such as zooming and panning, or choosing a font, which are needed in different situations, are done in a similar way. Sometimes such shared techniques involve some simple combinations of the mouse and/or keyboard. For example, to magnify a portion of a window, whether the window contains an Igraph, an image, or a 3-D plot, the same combination of mouse button and motion is used.

In other cases, shared panels are used. For example, when selecting a color for links in an Igraph, or a color for a graphical data object associated with an image, the same **Color** panel is used.

Most of these techniques have been introduced in previous chapters, but only with just enough explanation for the user to proceed through the tutorial smoothly. This chapter contains complete information about these techniques.

### 8.1 Zooming and Panning with the Mouse

Most windows in **WiT** can be zoomed and panned. For example, you may find the default icon sizes too large when designing a large Igraph, so zooming out (making the icons smaller) is more convenient. Or when you are examining images, you may want to study the image on a larger scale, but you do not want to make the window bigger, which may take up too much room on your screen. Zooming changes the scale of whatever the window contains without changing the window size. **WiT** provides you with a consistent and convenient way to do zooming and panning on all windows where such actions are appropriate.

#### 8.1.1 Zooming

To activate zooming, press and hold down the shift key on the keyboard, then press and hold the left mouse button at the upper left corner of the area that you wish to magnify. When you move your mouse to the *right* of the position that you initiated the zoom, a magnifying glass with a '+' sign will appear. If you move to the left, the magnifying glass will have a '-' sign instead. The '+' sign indicates that if you release your mouse button now, objects will appear larger (zoom in). The '-' sign means the opposite (objects smaller, zoom out).

When zooming in, the area within the rubber band rectangle will be magnified to take up the entire window. When zooming out, the entire window will be shrunk to fit into the rubber band rectangle.

#### 8.1.2 Panning

You pan the contents of a window by dragging with the middle mouse button and shift key down. Notice the cursor changes to a truck to indicate a move operation.

#### 8.1.3 Restoring to Original Scale and Position

Windows which support zooming and panning will have a **View all** item on their pop-up menu, displayed when the right mouse button is pressed within the window. Choosing this item will cause the window contents to be scaled such that all objects are visible. For example, when viewing an image, the image will be scaled so that it exactly fills the window. Or when applied to

an Igraph, operators and links will be scaled so that all objects can be seen and exactly fills the available window area.

The **Normal** item in the pop-up menu will restore all objects at original scale.

## 8.2 Resizing Windows

When a **WiT** window that displays any graphics (Igraph, image, object text, etc) is resized, the contents of the window will be scaled according to the new window size. For example, when you display a  $64 \times 64$  image, the window size will default to  $64 \times 64$  when it is first displayed, i.e., the image exactly fills the window. If you then resize the window to a size of  $256 \times 128$  pixels, the displayed image will be scaled so that it still fills the new window exactly. Which means every pixel in the image will occupy a  $4 \times 2$  rectangular area on the screen.

One exception to this rule is the main **WiT** workspace. The workspace is a design area, a resize of this window does *not* change the scale of the contents, but simply alters the size and shape of the design area.

If the contents of a window has been zoomed or panned, then resizing the window will preserve the *relative scale and* offset between contents and window. For example, suppose you have a  $128 \times 128$  image. When it is first displayed, the window size will also be  $128 \times 128$ . Suppose you have zoomed in on this image so that now you are looking at an area of  $64 \times 64$  of the image. Each pixel in the image therefore occupies  $2 \times 2$  pixels on the screen. Further suppose that you have panned the image so that pixel (50, 50) of the image is at the upper left corner of the window. If you now resize the window to a size of  $256 \times 64$ , pixel (50, 50) of the image will still be at the upper left corner of the window. Also, the new window will still show the same  $64 \times 64$  portion of the image, so that each pixel in the image will now occupy a  $4 \times 1$  area on the screen.

## 8.3 Selecting Graphical Objects

**WiT** attempts to be as consistent as possible in its treatment of similar graphical objects. For example, operators and links in an Igraph, or a graphical object used with the **getData** operator (Section [GetData](#)), can all be considered graphical objects, since they have shape and size, and so can be moved, adjusted, deleted, etc.

A graphical object is selected by clicking (press and then release) the left mouse button while pointing at it. The object turns red when it is selected (stippled on a monochrome). Selecting another object will cause the previous one to be deselected. To *add* an object to the selected list, click the *middle* mouse button instead. Clicking the middle mouse button on an object already selected will cause it to be deselected.

You can also select objects by area. Move the cursor to some blank spot, then press and hold the left mouse button. Now as you drag your mouse, you will see a rubber band rectangle. When you release your button, all objects that are *completely* within this rectangle will become selected.

## 8.4 Moving and Adjusting Graphical Objects

Where appropriate, graphical objects (operators, links, or graphical objects), can be moved around or their sizes and shapes can be adjusted. To move an object, first select it, then press and

hold the left mouse at the selected object. Now as you drag your mouse, the object will move along with the mouse. The cursor also changes to a truck to indicate you are moving objects. The object will be placed wherever you release the left mouse button. If multiple objects are to be moved, select them all, and then start dragging your mouse while pointing at *any* one of the selected objects.

Some objects, such as Igraph links and polygon graphical objects, can be adjusted (moving the link corners or polygon vertices). Others, such as Igraph operators, are fixed in both size and shape and cannot be modified. Where applicable, you modify an object by first making sure it is *not* selected. Then press and hold the left mouse button at the feature (link corner, polygon vertex, etc.) which you want to modify. Now as you drag your mouse, you will see the cursor changes to a screw driver, and the feature will rubber band to the position of your mouse.

## 8.5 Color Panel

WiT Color Panel  
(Figure omitted on purpose on demos)

Whenever the user needs to specify a new color for **WiT** to use, **WiT** brings up a **Color** panel (Figure [WiT Color Panel](#)). The **Color** panel presents different methods of specifying a color, because different people have different preferences, and the most appropriate method also varies depending on the application. The color disc on the upper left corner (based on the HSV color model, which stands for Hue, Saturation, Value) is useful when you want to specify a color approximately. As you drag the mouse in this area, the color under the mouse is displayed in a rectangular box on the lower right corner. This box is divided into two parts, the actual (exact) color on the left, and the closest color that **WiT** can provide on the right. The closest color cannot always be exact because **WiT** only uses a limited number of colors from the window manager. The **R**, **G**, and **B** sliders on the lower left are also updated as you drag the mouse inside the color disc. These sliders represent the red, green, and blue components of the selected color.

RGB is of course another method of specifying colors. People who are used to mixing paints may find that RGB provides more intuitive control. Even though mixing light and mixing paint work in opposite ways, the procedure of adding small amounts of primary colors to achieve a desired color can easily be mastered by someone with an understanding of color. For example, to get a specific shade of brown, we can start with gray, add a little red, adjust with a little green (which when mixed with red *light*, gives a shade of yellow). Proceeding this way, it is not difficult to achieve the *exact* shade of brown one wants. As you change either one of the R, G or B components, the crosshair on the color disc moves to reflect the corresponding HSV representation of the same color.

The color disc only allows you to select the hue and saturation for the HSV model. The value component can be altered with the **V** slider above the RGB sliders. A high **V** means bright colors, and a low **V** means dark colors.

In addition to RGB and HSV, the **Color** panel also provides a list of common colors on the



upper right. You can select these colors by clicking with the left mouse button. Again, both the RGB and HSV input gadgets are adjusted accordingly to reflect the selected color.

Whichever input method you employ, you must click the **Apply** button to activate the color.

## 8.6 Zoom/pan

In addition to using combinations of mouse buttons and motion to do zoom and pan (Section [Zooming and Panning with the Mouse](#)), **WiT** sometimes also offers a **Zoom/pan** panel (Figure [Zoom/pan Panel](#)) as an alternative. For example, zoom/pan on the main **WiT** workspace, whether in data-driven or demand-driven mode, can be performed using the mouse or by bringing up the **Zoom/pan** panel from the workspace pop-up menu.

**Zoom/pan Panel**  
(Figure omitted on purpose on demos)

The arrow buttons on the left controls panning up, down, left, and right. Every click of the button pans the view area in the indicated direction a certain amount, which is controlled by the **Pan %** slider below these buttons. The amount is specified in percentage of the visible area. The center button bring the viewing area 'home.' For example, when using this panel with an Igraph, 'home' positions the viewing area such that the upper-left most object in the Igraph is positioned at the upper-left corner of the workspace.

The magnifying glass buttons in the middle of the **Zoom/pan** panel control zooming. The zoom amount is controlled by the **Zoom %** below these buttons. Again, as in panning, the amount is specified in percentage of the visible area. The **Normal** rescales such that objects appear in their original size, and **View all** scales the view area so that all objects are visible.

## 8.7 Printing

Most graphics windows in **WiT** can be printed either directly to a printer or to a file. To print something, select the **Print...** item from the main panel **File** button to bring up the **Print** panel (Figure [Print Panel](#)). Currently only PostScript is supported. The options are:

**Print Panel**  
(Figure omitted on purpose on demos)

<b>Windows to print</b>	<b>All</b>	Print all windows.
	<b>Selected</b>	Print only selected windows.
<b>Format</b>	<b>As displayed</b>	Frames printed with same relative positions as displayed on screen.

<b>Image quality</b>	<b>One/page</b>	Each frame on a separate page, centered.
	<b>Draft</b>	Use the resolution of the screen. If you are using a monochrome display, the output will have exactly the same number of dots as you see on the screen.
	<b>High</b>	Images are sent to the printer in grayscale. This allows the printer the freedom to use whatever halftone screen to produce the best results. On low-cost printers, the results may not look very good, but you can send the same file to a professional printer and obtain flawless images. Grayscale images requires much more data to be transferred to the printer, and so may take more time to print, particularly if the link to the printer is serial. The images can be more difficult to photocopy too.
<b>Show frame</b>		Enable or disable a frame with the window name around each printed window (Igraph, images, etc.)

Most graphics windows in **WiT** can be selected for printing. If you bring up the pop-up menu in a graphics window, you should see that the first item is **Select**. Choosing this item will cause a check mark to be displayed on the upper left corner of the window. When **Frames to print** is set to **Selected** on the **Print** panel, only frames with the check marks will be printed.

After you have finished choosing all the applicable options, click the **Print** button.

## 8.8 Icon Editor

**WiT** has a built-in icon editor (Figure [Example Icon Editor Session](#)) which not only lets you design the appearance of operator icons, but also allows you to move input and output ports, among other things. Because it is built-in to **WiT**, when you change an icon with the icon editor, those changes are updated immediately on the Igraphs you are looking at.

### Example Icon Editor Session (Figure omitted on purpose on demos)

The icon editor is invoked by selecting the **Edit** item from the library pop-up menus, or clicking the **Edit** button in the **Operator Selector** panel. The following controls are available:

<b>Mode</b>	Input mode, see below.
<b>Title Justify</b>	Specifies how the operator title text is justified.
<b>Position</b>	Position of mouse in icon coordinates.
<b>Grid</b>	Size of grid in icon coordinates. Ports, origin, and title in the icon editor always snap to this grid (not the same as the Igraph grid).
<b>Frame</b>	Specify whether the icon should be drawn with a frame or not. On a color display, framed icons have a 3-D look.

The editor operator works in four possible input modes, selectable with the **Mode** item as follows:

<b>Ports</b>	Move input or output ports.
<b>Icon</b>	Change icon graphics. A click of the left mouse button sets a pixel, the middle button clears it.
<b>Origin</b>	Move the origin (hot spot) of the icon. The origin is indicated as a green center of gravity symbol in the icon editor. When an operator icon is placed on the Igraph which <b>snap</b> turned on, the origin of the icon is snapped to the closest grid intersection point. You should place the origin in such a way that the input and output ports will all be lying on the same grid as the origin.
<b>Title</b>	The title of an operator is normally shown on the top of the operator icon, centered. You can change the position and alignment of the operator title text in this mode. The title position is shown as a blue flag in the icon editor.

## 8.9 Array Editor

When **WiT** requires the user to input two-dimensional data, such as convolution kernels, it brings up an array editor (Fig. [Example Array Editor](#)). The editor consists of three regions: a menu bar, a panel, and the actual array.

Example Array Editor  
(Figure omitted on purpose on demos)

### 8.9.1 Menu bar

The menu bar contains a single item **File**, which gives you the following functions:

<b>Load...</b>	Load a data file into the array.
<b>Save</b>	Save array data to the file the data came from originally.
<b>Save as...</b>	Save array data to new file.
<b>Done</b>	Apply contents of editor and take it down.

### 8.9.2 Panel

The array editor panel contains these controls:

<b>Entry</b>	Value of selected item (see below) in array.
<b>Rows</b>	Number of rows in array.
<b>Cols</b>	Number of columns in array.
<b>Apply</b>	Change array size to <b>Rows</b> and <b>Cols</b> .

The selected item is highlighted with a bold line box in the array region.

When the editor first comes up, it usually defaults to a certain meaningful size. For example, most imaging filters defaults to a  $3 \times 3$  array for the kernel. This size can be changed interactively by modifying the **Rows** and **Cols** text entries on the array editor. After you changed these entries, you must click the **Apply** button for them to take effect.

### 8.9.3 Array

The array region consists of three areas: row labels, column labels, and the array data. At any one time, one of the array items is the selected item, and it is highlighted with a bold line box. The value of this entry also appears in the **Entry** item on the array editor panel. When you modify the **Entry** item value, the new value is immediately copied to the actual array item. To change the selected item to a different element of the array, click the left mouse button on the desired element. You can also *copy* the value in the **Entry** item to any element by clicking the middle mouse button at the desired element.

For some applications, it is necessary to define an 'origin' in the array. This is indicated by a center of gravity symbol on the upper right corner of the array element. You can define the origin at a new position with the right mouse button.

When the array size is larger than the window size, scroll bars will appear. You can use these

scroll bars to look at different regions of the array. You can increase or decrease the visible portion of the array by resizing the window itself. Notice that the actual size (rows and columns) of the array is not affected by changes in the window size.

## Chapter 9 Interactive Operators

Some operators in the **Interactive** library allow you to interact with the operator, with graphical feedback. Several of these, such as **getData**, provide such extensive functionalities that they are not described in the online operator help files, whose main purpose is to document simple-to-explain operators, but are explained here instead.

### 9.1 GetData

The **getData** operator allows you to place graphical objects: points, rectangles, text, etc, to specific positions on an image. It accepts an image as input, and it displays the image together with a **getData** properties panel when the image is received (Figure [GetData Panel](#)). Graphics objects are then entered interactively on the image. Objects can be given different colors or fonts. They can be adjusted, moved, copied, etc, after they are entered.

GetData Panel  
(Figure omitted on purpose on demos)

Graphics objects on images can be printed from the **Print** panel. The usual conventions apply.

#### 9.1.1 Entering Graphical Objects

The **Type** button menu in the **getData** panel allows you to choose among the various graphical objects that you can enter, or to select and edit objects already entered with the **Select** item.

Because each graphic object type is so different from the others, the input procedures for each are necessarily different. Following is a description of how each graphical object type is entered. **GetData** also reminds you of what you should do next in the **WiT** status window.

<b>Select</b>	Select or edit existing objects.
<b>Point</b>	Each click defines a point.
<b>Line</b>	Press and hold left mouse button to define first end of line, then drag to other end of line and release button.
<b>Polyline</b>	Click (press and <i>release</i> ) left mouse button to define first point. Each subsequent click defines a corner. Terminate with double-click.
<b>Rectangle</b>	Press and hold left mouse button to define first corner, then drag to second corner and release button.
<b>Circle</b>	Press and hold left mouse button to define center of circle, then drag to desired size and release button.
<b>Text</b>	Click at where you want text to begin. You will see a vertical bar (the text cursor) appear. Enter text string from keyboard. Click left mouse button or hit escape key to terminate text entry.

Try to enter some of each of the object types. Do not try to use **Select** until you are finished experimenting with the different graphic object types. Its use is covered in the following section.

### 9.1.2 Selecting, Moving, and Adjusting Objects

When you enter graphic objects, you are almost certain to make mistakes. The **getData** operator is in fact a miniature drawing program. It allows you to cut, copy, paste, move, and adjust objects which you have entered.

After you have finished entering your last object, choose the **Select** item from the **Type** menu. You are now in edit mode. Follow the techniques described in Section Selecting Graphical Objects.

### 9.1.3 Editing Objects

The **Edit** button provides all the common editing features in a drawing program:

<b>Undo</b>	Undo the last edit action.
<b>Cut</b>	Cut the selected objects.
<b>Paste</b>	Paste deleted or copied objects.
<b>Copy</b>	Copy selected objects to buffer.

### 9.1.4 Fill Patterns

You can choose to fill graphical objects with solid color, or have them appear hollow. This is done via the **Fill** choice widget.

### 9.1.5 Fonts

**GetData** allows you to associate a specific font (family, style, and size) to each text string you enter. To try this out, click the **Font...** button from the image properties panel. This brings up the familiar font dialog.

After you have selected a font and clicked the **Apply** button on the **Font** panel, the next text string you enter will take on the new font. There is no limit on the number of text strings with different fonts that can appear on each image.

### 9.1.6 Graphical Object Colors

You can select a color for each graphical object you enter. Click the **Color...** button on the **getData** properties panel, a **Color** panel will pop up. See Section Color Panel for details about how to use this panel. After you have selected a color and clicked the **Apply** button on the

**Color** panel, all selected objects will adopt the new color, as well as all subsequent graphical objects you enter. There is no limit on the number of colors that can appear on each image. Because **WiT** uses only a small number of colors from the window manager (see Section [Color](#) for details), the color that you see for the graphical objects may not be exactly the color you specified, but the exact color *is* stored internally with the objects.

## 9.2 Surface

The **surface** operator draws a gray scale image as a three dimensional surface, with the intensity of the image as the height of the surface. Figure [Example Surface](#) shows an example of **surface** and its associated properties panel. (The properties panel is brought up from the window pop-up menu, as in image windows).

Example Surface  
(Figure omitted on purpose on demos)

In addition to the usual zoom and pan controls, **surface** provides you with 3-D controls on its properties panel. Figure [3-D Controls](#) illustrates some of the terms related to 3-D viewing manipulation.

3-D Controls  
(Figure omitted on purpose on demos)

In our 3-D viewing model, we imagine ourselves to be looking through the viewfinder of a camera, which we will refer to as the eye ( $E$ ), at some 3-D object (in this case, a surface that looks like a piece of mountainous land). The focus ( $F$ ) is the point in 3-D space that we are pointing our camera at. Axes are assigned to the 3-D surface: the X-axis is along the width of the image, Y is along the height of the image, and Z is the intensity of the image, represented as the elevation of the 3-D surface. The line ( $L$ ) between the eye ( $E$ ) and the focus ( $F$ ) is called the view axis.

We always stand upright (in other words, we do not rotate the eye about the view axis). Other than this restriction, we are free to move the eye (but not the focus) to wherever we want. If we imagine ourselves to be standing on the X-Y plane of the 3-D surface we are looking at, and we walk around the focus, then we are varying the angle between  $L$  and  $X$  ( $A_h$ ), which we will call the horizontal angle. (We are also varying the angle between  $L$  and  $Y$ , but that is redundant.) If we step on a ladder to look at the surface from a higher vantage point, then we are varying the angle between  $L$  and  $Z$  ( $A_v$ ), this we will call the vertical angle.

The **surface** properties panel provides you with 3-D viewing controls and different ways of rendering the surface as follows:



<b>Horz. angle</b>	The angle between the view axis projected on the X-Y plane and the X-axis ( <i>A_h</i> ).
<b>Vert. angle</b>	The angle between the view axis and the Z-axis (vertical axis) ( <i>A_v</i> ). When <b>Vert. angle</b> is zero, you are looking straight down at the image.
<b>Distance</b>	<p>The distance between the camera and the focus, i.e. the length of (<i>L</i>). This has a different effect than the two-dimensional zooming available in most windows (also available on <b>surface</b> windows too). Our camera analogy will help explain the difference.</p> <p>The view displayed in the window is like the picture taken on film in a camera. Move closer to or away from the object we are taking a picture of is the same as changing <b>distance</b>. Two-dimensional zooming is equivalent to enlarging or reducing the picture <i>after</i> the object has been photographed. Therefore, varying <b>distance</b> has a perspective effect, zooming does not. Zooming and panning may also cause part of the object to be clipped.</p>
<b>Scale</b>	The magnification applied to the image intensity to compute the height of each point on the surface. A smaller magnification means the surface will appear flatter.
<b>Pix/sq</b>	<p>Pixel per square. The 3-D surface is drawn as a rectangular mesh. Each rectangle (square) on the surface represents an area of one or more pixels in the image. A small <b>Pix/sq</b> gives you a fine and more accurate surface, but takes longer to render. So when you are adjusting your view angle, distance, etc, it is a good idea to first set <b>Pix/sq</b> to a large value, do the viewing adjustments, and then set <b>Pix/sq</b> to the value you want.</p> <p><b>Pix/sq</b> behaves slightly differently on a color and on a monochrome display. On a color display, no boundaries are drawn between the squares. Boundaries are drawn with the size specified by <b>Wireframe</b> (see below). <b>Wireframe</b> is forced to be always larger than or equal to <b>Pix/sq</b>.</p> <p>On a monochrome display, boundaries are drawn between the <b>Pix/sq</b> squares, and the <b>Wireframe</b> is not drawn, <i>except</i> when <b>Pix/sq</b> is set to zero.</p> <p>When <b>Pix/sq</b> is zero, then for both color and monochrome displays, the 3-D surface will no longer appear solid. Instead, a wireframe is drawn at the size of <b>Wireframe</b>. Figure <a href="#">3-D Wireframe</a> shows such a view.</p>
<b>Wireframe</b>	Specifies the size of the visible grid overlaid on the 3-D surface. <b>Wireframe</b> behaves differently in color and monochrome displays. See <b>Pix/sq</b> above for details.

### 3-D Wireframe (Figure omitted on purpose on demos)

Surfaces can be printed from the **Print** panel. The usual conventions apply.

## 9.3 Graph

The **graph** operator accepts a variety of input types and plots the data on a graph.

If the input is a vector of integers or floating-point numbers then each item in the vector is plotted against its position within the vector.

If the input is a vector of points (ordered pairs), then the  $y$  value of each point is plotted against its corresponding  $x$  value.

The input can also be a vector of any of the above vector types, in which case each vector will be plotted as a separate curve within the same graph.

Figure [Multi-Data-Set Graph](#) shows an example of a multi-data-set plot.

### Multi-Data-Set Graph (Figure omitted on purpose on demos)

The title of the graph and the labels on the X and Y axes are parameters of the **graph** operator. When multiple data sets (curves) are shown, different colors (on color displays) or line styles (on monochrome displays, and when graphs are printed) are used to distinguish between the sets. The legend on the bottom identifies the curves. Data sets can be plotted in a variety of ways, such as histograms or scatter plots, or be turned off altogether. A data set must be selected before its attributes can be changed. Data sets are selected by selecting their corresponding legends. A selected legend is shown in reverse.

You can also display the coordinates of various positions on the graph. Simply click the left mouse button at the position you want (see Figure [Multi-Data-Set Graph](#)). If you drag your mouse on the graph, the coordinates will be continuously updated.

Zoom and pan is of course supported also, with automatic recalculation of axis labels. Because it is highly unlikely that someone will want to zoom in or out on the axis labels, zooming and panning only applies to the data set area, not the axis labels or legends.

As in displayed images, if two **graph** operators have the same name, then only one window will be used. Whenever a new set of data is sent to a **graph** operator, the previous data sets will be erased before the new one is plotted on the same window. This is useful to keep track of changing data sets as an Igraph is executed.

**Graph** also has an associated properties panel (Figure [Graph Properties Panel](#)) with the following controls:

### Graph Properties Panel (Figure omitted on purpose on demos)

<b>Curve type</b>	Select how the selected data sets are plotted.
<b>Select all</b>	Select all curves.
<b>Deselect all</b>	Deselect all curves.
<b>Next set</b>	Whenever a large number of curves are plotted on the same graph, the legend on the bottom only show a subset of the curves (initially, the first few). Click this button to display the next set of curves on the legend.
<b>Previous set</b>	Display the previous set of legends in a multi-curve plot.

Graphs can be printed from the **Print** panel. The usual conventions apply.

## 9.4 System

The **system** operator allows you to invoke another program from within **WiT**. It takes a single **sync** token as input and produce a **sync** token output. The program to execute and command line arguments should be specified in the single parameter as a single string, exactly as you would type when invoking the command from the shell. The program is executed when the input token arrives, and the output token is produced when the program is terminated.

In addition to allowing you to execute programs you did not write yourself, **system** can be a useful alternative to invoke your own operators, instead of using the client/server mechanism. You can use operators that have a GUI interface, and they can even be compiled with compilers totally incompatible with the compiler used to compiled the **WiT** libraries.

If the program you want to execute requires more than one input which are produced upstream in **WiT**, and you want to make sure the program is only executed when all the inputs are available, you can convert those object tokens to **sync** tokens and have the **sync** tokens converge in a junction which leads to the **system** operator.

**System** generates a **sync** output so that it can synchronize whatever outputs the command it executes produces to the rest of the Igraph. The operator **syncRead** is specifically designed to work with **system**. The program executed by **system** can create **WiT** objects and save them in files, then the **sync** output from **system** can trigger any number of **syncRead** operators to read the **WiT** objects back into the Igraph and then processed further by other operators.

If you want your program to process **WiT** data objects or produce objects that you can send back to **WiT** operators, you will need to read and write data in **WiT** format (with a `wit` extension). See Chapter [WiT File Format](#) for sample code and object format information.

Figure [Example Use of System Operator](#) shows an example of the use of **system** and **syncRead**.

Example Use of System Operator  
(Figure omitted on purpose on demos)

## Chapter 10 Configuration

**WiT** provides mechanisms for you to configure your work environment, such as where pop-up windows are placed, where servers, libraries, or icons are located, how many servers to run, and customizing operator icons, etc.

### 10.1 Command Line Options

You may want to start **WiT** with some personal preferences. For example, you may want to start up in data-driven mode, with no grid. **WiT** provides the following command line options:

<code>-speed</code>	Set execution speed. Default is <code>walk</code> .
<code>-autoclr</code>	Toggle object auto clear. Default is on.
<code>-igrDir &lt;string&gt;</code>	Set the Igraph directory.
<code>-objDir &lt;string&gt;</code>	Set the object directory.
<code>-grid &lt;size&gt;</code>	Set grid size amount (in pixels). Default is 60.
<code>-mode &lt;data/demand&gt;</code>	Set data or demand-driven mode. Default is <code>data</code> .
<code>-ngray &lt;integer&gt;</code>	Set the number of gray scales to be used for grayscale images. Default is 48.
<code>-nreg &lt;integer&gt;</code>	Set the number of registers to be used in demand-driven mode. Default is 5.
<code>-pick</code>	Set pick tolerance. Default is 5.
<code>-rgb &lt;choice&gt;</code>	Set number of colors for color images. Choose from <code>none</code> , <code>coarse</code> , <code>medium</code> or <code>fine</code> . Default is <code>medium</code> .
<code>-scheme</code>	Select color scheme to use <code>color</code> , <code>light</code> , or <code>dark</code> . Default is <code>light</code> .
<code>-showgrid</code>	Toggle show grid flag. Default is on.
<code>-snap</code>	Toggle snap on/off flag. Default is on.
<code>-window &lt;W&gt; &lt;H&gt;</code>	Set the size of the <b>WiT</b> window. Default is 800x400.

You can also specify an Igraph name to use on start up. For example, to set the options we mentioned previously and to use the Igraph **mygraph** on start up, type:

```
wit -mode data -showgrid mygraph
```

### 10.2 Directories

When **WiT** reads something from the file system, such as an Igraph or an image, it goes through

a list of directories (search path) to locate it. The first file that has the correct name will be used. See Section [Execution Environment](#) for an explanation of search paths, and how they can be set up.

When **WiT** writes out something, only two possible directories are used: an Igraph directory for Igraphs, and an object directory for objects. The Igraph directory can be changed when you invoke either the **Load** and **Save as** items from the **File** menu. The object directory can be changed from the **Object dir** item from the **File** menu.

One exception to this rule is that when saving an Igraph, if the Igraph has been read from the file system (as opposed to having been designed from scratch), then when it is saved using the **Save** item from the **File** menu, the Igraph will be saved to where it originally came from. If you do not want this, use the **Save as...** item instead, which will allow you to specify both a new name for the Igraph and choosing which directory the Igraph will be saved to.

Both Igraph and object directories default to the directory that you start up **WiT** initially.

### 10.3 Designing Your Own Icons

Every attempt has been made to provide icons that reflect the meaning of all operators in standard **WiT** distribution libraries. These are stored in the `$WITHHOME/lib/icons` directory. The file format is an ASCII format used commonly on UNIX, called X11 bitmap.

Simple changes can be conveniently done on line with the **WiT** built-in icon editor, details of which has been described in Section [Icon Editor](#). The size should preferably be 60x40, although **WiT** can handle any reasonable size. Notice the port positions are not defined with the icon file, but in the operator definition files in `$WITHHOME/lib/def`.

It is easiest to position ports interactively with the **WiT** built-in icon editor.

Store new icons that you have created under the directory `\$WITHHOME/lib/icons`.

We also suggest that you make a backup of the original icon, in case you or somebody else want to revert to the default icon. See the *Programmer's Guide* for information about making an icon for a new operator (as opposed to changing an icon for an existing operator).

Because of the large number of operators that **WiT** supports, reading in all the icons that are in ASCII can take considerable time. For this reason, **WiT** maintains an icon cache, normally stored as `witicons` in the home directory (which can be changed, see Section [Execution Environment](#)). New icons are parsed and stored in the cache every time **WiT** is run. However, if you modified an existing icon, **WiT** will not know about it and will still use the old icon from the cache. To force **WiT** to use the new icon, delete the `witicons` file. If you use the **WiT** icon editor, it is done automatically. This will cause some delay the next time **WiT** is started, because it has to parse all the ASCII files, but that will only need to be done once. The next time **WiT** is started, it will revert to the icon cache again.

### 10.4 Color

When **WiT** is run on color workstations, sharing of colors with other applications you may be running become an issue. Many workstations support only 256 colors. These colors must be

shared among all applications currently running, or else the system colormap will have to be swapped as you move your mouse into different applications, causing irritating color flashes. By default, **WiT** requests just enough colors from the window manager so that animation and Igraph editing (rubber banding, etc) can be nicely done and images can be shown with reasonable contrast, both grayscale and color. Specifically, **WiT** uses 8 colors for the Igraph and other graphics, 48 colors for grayscale images, and 64 colors for color images.

The default 48 colors used for grayscale images provide reasonably smooth rendition of grayscale images. However, if necessary you can increase the number of colors allocated with the `-ngray` command line option.

#### 10.4.1 Configuring Workspace Colors

You can change the colors that **WiT** uses for the workspace, such as colors used for drawing icons, the workspace background, and the grid. If you bring up the **Setup** panel from the **Options** menu, and select **Custom** for **Workspace Colors**, a color editing panel will come up. On the left of this panel is a list of object types whose colors can be changed. On the right is a color selection panel (described in Section [Color Panel](#)). The object types are:

<b>Background</b>	Background for Igraphs, operator panels, filer, etc.
<b>Foreground</b>	Links, probes, ports, and icon graphics.
<b>Selected</b>	Selected objects.
<b>Busy</b>	Operators currently being executed.
<b>Labels</b>	Operator names above icons.
<b>Parameters</b>	Parameter names and values below icons.
<b>Icons</b>	Background of icons.
<b>IconBright</b>	Top and left borders of icons (for 3D effect).
<b>IconDark</b>	Bottom and right borders of icons (for 3D effect).
<b>Grid</b>	Grid on Igraph.

When you are satisfied, click the **Apply** button. Your customized colors will be written to the file `witrc` in your home directory. You can discard all your changes by hitting **Reset**, although this does not affect your previously saved custom colors. If you really want to remove all records of custom colors, click the **Remove** button.

### 10.5 Servers

As shipped, **WiT** is configured to run all built-in operators within the GUI program (the `'WiT'` program that you see). However, **WiT** is based on a server/client model where the client is the GUI part and the server(s) is comprised of one or many computational servers. The client and servers are separate programs, possibly running on different computers! If your computer is connected to a local area network, you may want to run **WiT** servers on all machines that are not

busy, so that your Igraphs can execute faster if there is any inherent parallelism in them.

Even if your computer is not networked, the separation into client and server halves allows you to add your own operators to **WiT** by rebuilding the server part only. The client (GUI) part never needs to be changed. If you want to launch the server as a separate program, copy the `witrc.svr` file under the `$WITHHOME\config` directory to `witrc`. This will give you a single detached server running on the same machine as the GUI. If you want to switch back to a no server configuration, copy `witrc.gui` to `witrc`. See Section [Witrc File](#) for more information about the `witrc` file.

If you want to start multiple servers, you will need to modify the `witrc.svr` file. Copy `witrc.svr` to `witrc`, and have a look at `witrc`. It contains nested keyword/attribute pairs (enclosed with angle brackets) for various setup options that will be discussed in detail in Section [Execution Environment](#). For now, we are only interested in the `server` keyword. A separate `server` entry is needed for each server you want to run. Each `server` entry in turn consists of six sub-entries. For now, we need not worry about the meaning of these sub-entries. They are explained in full detail in the *Programmer's Guide*.

In general, the user need only duplicate an existing `server` entry then change the machine name for where the server will run. For example, suppose we have a network which consists of four machines: `earth` running Windows NT, `mars` running Windows NT, `venus` running Windows 3.1, and `pluto` running SunOS. We want to run the **WiT** GUI on `earth`, with a **WiT** server running on each of `earth`, `mars`, `venus`, and `pluto`. The server on `earth` is a local server using DDE to communicate with the GUI. The others use sockets to communicate. Our `witrc` file should look like this:

```
<objpath      ...
  ...
  ...
<server
  <binary      witnt>
  <group       general>
  <arch        win32>
  <ipc         dde>
  <hostname    mars>
  <maxOp       200>
  <maxObj      10>
  >
<server
  <binary      witnt>
  <group       general>
  <arch        win32>
  <ipc         socket>
  <hostname    mars>
  <maxOp       200>
  <maxObj      10>
  >
<server
  <binary      witnt>
  <group       general>
  <arch        win31>
```



```

    <ipc          socket>
    <hostname     venus>
    <maxOp        200>
    <maxObj       10>
  >
<server
  <binary       witsun4>
  <group        general>
  <arch         sun4>
  <ipc          socket>
  <hostname     pluto>
  <maxOp        200>
  <maxObj       10>
  >
<operators
  ...
  ...

```

### 10.5.1 Multiple server types

When the same operator name appears for different server types in the `witrc` file, a server type property in the parameter panel will appear when the operator is selected (Figure [Multiple Server Types](#)). This permits the user to decide where an operator is to be executed. The icon for the duplicate operator will appear in the first operator library panel. For example, consider the case where you have two servers called **witsparc** and **witi860** where the former is a general server and the latter is a server supporting the Intel i860 processor. If the **rdObj** operator is defined in **ioLib** which is included in both the general and i860 servers then the property panel for the **rdObj** operator would appear as in Figure [Multiple Server Types](#).

Multiple Server Types  
(Figure omitted on purpose on demos)

There is a new item called **Server** on this property panel. Using our example, you can select either **witsparc** or **witi860**, and **WiT** will schedule the operator *only* on the specified server. If the specified server is busy but the other server is available, **WiT** will still wait for the specified server to become available. If you do not care which server the operator runs on, you can select **any**. When **any** is selected and all the servers are idle, the server whose name appears first in the **Server** list will be selected.

## 10.6 Execution Environment

Controls of execution environment such as the list of servers and search paths for **WiT** objects and Igraphs are performed by modifying the file `witrc`, which contains (possibly nested) attribute/value pairs surrounded by angle brackets. We have seen how to set the number of servers in Section [Servers](#). Here we will explain how the other attributes are set up.

### 10.6.1 Witrc File

All the configuration information for **WiT** is kept in the file `$WITHOME\config\witrc`.

If you want to customize your working environment, such as adding new operators, changing the number of servers, or the architecture types of the servers, you would modify `witrc`. If you have defined an environment variable `$HOME`, **WiT** will make a copy of `witrc` there, and that is the copy you should modify.

In addition to the default `witrc` file, the `config` directory also contains the default operator icon (when the user does not supply an icon), and help files for general topics (as opposed to specific operators).

The default `witrc` as shipped looks like this:

```
#
# WIT uses search paths to lookup the location of
# objects (objpath), igrphs (igrpath), operator libraries
# (libpath), and WIT servers (svrpath).
#
# A pathname is a directory which may utilize environment
# variables. Each pathname is separated by a colon.
# Any path entry may be used more than once in which case
# the collective set of paths are used starting from the
# search from the first definition.
#
<objpath      .;${WITHOME}\images>
<igrpath      .;${WITHOME}\lib;${WITHOME}>
<libpath      ${WITHOME}\lib>
<svrpath      ${WITHOME}\servers>
#
# The iconcache defines a cache file for reading
# operator icons. When this entry is present, icons for operators
# are read from this file to accelerate the start up process.
# If new icons are introduced, then this file must first be
# removed to force an update of the cache.
#
<iconcache    ${HOME}\witicons>
#
# The server keyword defines a WIT server which is to be started
# and utilized by the UI. The set of nested keyword-value
# pairs describe the name of the server (binary), the type
# of operators the server can run (group), the server's
# architecture (arch), which machine the server will run
# on (hostname), and two attributes to be used in the future
# (maxOp and maxObj).
#
<server
    <binary      witnt>
    <group       general>
    <arch        win32>
    <ipc         dde>
    <hostname    localhost>
    <maxOp       200>
    <maxObj      20>
>
#
# The operators keyword defines a list of operator libraries
# belonging to a particular group. This group is then associated
# with a server. The set of nested
```

```

# keyword-value pairs defines the group name (group) and
# the various operator definition files (define). A define
# has two attributes, the name of the actual C library where the
# operators can be found and a descriptive name to be used
# by WIT in the operators pull-down menu.
#
<operators
  <group          ui>
  <define         flow          Dataflow>
  <define         ui           Interactive>
  >
<operators
  <group          general>
  <define         io           Read/Write>
  <define         data        DataManipulation>
  <define         point       Point>
  <define         filter      Filters>
  <define         xform       Transforms>
  <define         meas        Measurement>
  <define         segment     Segmentation>
  <define         morpho      Morphology>
  <define         pyramid     Pyramids>
  <define         proto       Prototypes>
  >

```

## 10.6.2 Search Paths

`Objpath` specifies where objects can be found. The value is a colon (;) separated list of directories, much like the syntax used to specify search paths in MS-DOS. You can use environment variables in the paths, such as `${WITHOME}` and `${HOME}`.

You can use drive letters.

You can also use dot ( `.` ) and double dot ( `..` ) to specify the current directory and the parent directory respectively. Since the `.witrc` file is parsed when you start **WIT**, the current directory that dot refers to will be the directory in which you start **WIT**.

When **WIT** reads an object, the `objpath` is searched for the object name. Searching is performed from left to right. So for the example `objpath`, the current directory ( `.` ) will be searched first for the object. If found, it will be loaded. If not, the searching continues with the next directory on the path list until either the object is found or the directories are exhausted.

`Igraphpath` works similarly as `objpath`, except that it applies to `Igraphs`.

`Libpath` specifies where libraries used by the **WIT** servers can be found. The syntax of the value is similar to that for `objpath`.

The actual library names are specified by the `operators` attribute. `Operators` in turn consists of sub-attributes `group` and `definition`. `Group` specifies the functional class of the operators, and `definition` is a sub-classification of operators within the library `group`. Unless you want to add new operators or server types, you do not need to modify the `operators` entries. Refer to the *Programmer's Guide* if you want more information.

## Appendix A WiT File Format

When objects are written to files using **wrObj**, they are stored in **WiT** file format. This file data is binary but machine independent in the sense that you can read and interpret the binary data correctly on any machine architecture. The major machine incompatibilities lie in how floating point numbers are represented, i.e. IEEE or VAX, and in how bytes are ordered within words and long words. On Intel machines, bytes are ordered in what is called little endian. On most other machines such as Sparc or Motorola, bytes are ordered in big endian. Regardless of these differences, the **WiT** file format ensures the data integrity is preserved by standardizing on a network ordering scheme common in all network packet transmissions.

All **WiT** object files begin with an object descriptor which is comprised of two fields. The first field is a 4-byte integer describing the `size` of the object description string. This `size` must be rounded to the nearest multiple of 4. The next field is read in as a character string of `size` length.

Based on the object name, the rest of the object data can be read in appropriately. Here is the code to read in all of the simple **WiT** objects including images and vectors of simple objects.

```
#include <stdio.h>

static void readObject(char *fn);
static void readObjHeader(FILE *fp, char *s);
static void readObjData(FILE *fp, char *objString);
static void readWitImage(FILE *fp);

main(int argc, char **argv)
{
    if(argc < 2) {
        fprintf(stderr, "usage: readObj NAME.wit\n");
        exit(1);
    }
    readObject(argv[1]);
}

static void
readObjHeader(FILE *fp, char *s)
{
    int size;

    fread(&size, 4, 1, fp);
    /*
     * Padd size to multiple of 4
     */
    size = (size+3)/4*4;
```

```
    fread(s, size, 1, fp);
}
```

```
static void
readObject(char *fn)
{
    FILE *fp;
    char objString[80];

    if(!(fp = fopen(fn, "r"))) {
        fprintf(stderr, "cannot open \"%s for read", fn);
        exit (1);
    }
    readObjHeader(fp, objString);
    fprintf(stderr, "Reading object: %s...\n", objString);
    readObjData(fp, objString);
    fclose(fp);
}
```

```
static void
readObjData(FILE *fp, char *objString)
{
    unsigned tmp;
    unsigned char uc;
    char c;
    unsigned short us;
    short s;
    unsigned ui;
    int i;
    float f;
    double d;
```

```
    if(!strcmp(objString, "OBJ_IMAGE")) {
        readWitImage(fp);
    }
    if(!strcmp(objString, "OBJ_FLOAT"))
        fread(&f, 4, 1, fp);
    else if(!strcmp(objString, "OBJ_DOUBLE"))
        fread(&d, 8, 1, fp);
    else if(!strcmp(objString, "OBJ_VECTOR")) {
        /*
         * Vectors have four fields:
         *
         * # bytes      type      description
         *
         * 4             int       # of elements
         * 4             int       ncols
         * 4             int       object string length giving type of
         *                vector data
         * (len+3)/4+4  char *    object string (padded to multiple of
         *                4 bytes)
         * ?            void *    vector data
         */
        int nelems;
```

```

int ncols;
int size;
char vecString[80];

fread(&nelems, 4, 1, fp);
fread(&ncols, 4, 1, fp);
readObjHeader(fp, vecString);
for(i=0; i < nelems; i++) {
    /*
     * Here we call recursively this function to
     * extract the vector data. You'll have to modify
     * this code to save the vector data somewhere,
     * otherwise it will be lost after extraction.
     */
    readObjData(fp, vecString);
}
}
else {
    fread(&tmp, 4, 1, fp);
    if(!strcmp(objString, "OBJ_CHAR"))
        c = tmp;
    else if(!strcmp(objString, "OBJ_UCHAR"))
        uc = tmp;
    else if(!strcmp(objString, "OBJ_SHORT"))
        s = tmp;
    else if(!strcmp(objString, "OBJ_USHORT"))
        us = tmp;
    else if(!strcmp(objString, "OBJ_INT"))
        i = tmp;
    else if(!strcmp(objString, "OBJ_UINT"))
        ui = tmp;
    else if(!strcmp(objString, "OBJ_HEX"))
        ui = tmp;
}
}

/*
 *

```

A WiTImage object file is formatted as follows:

# bytes	Data Type	Description
4	int	Image type
4	int	Image bpp
4	int	Image width
4	int	Image height
4	int	not used, was Image org_W
4	int	not used, was Image org_H
n*bpp*w*h	variable	Image data
		If imageType is
		8-bit,    bpp=1, n=1

```

        16-bit,  bpp=2, n=1
        24-bit,  bpp=1, n=3 (R,G,B)
        float,  bpp=4, n=1
        complex, bpp=4, n=2 (Real, Imag)
*
*/

enum ImageType {
    WIT_UINT8 = 0,
    WIT_INT8,
    WIT_UINT16,
    WIT_INT16,
    WIT_FLOAT,
    WIT_COMPLEX,
    WIT_RGB
};

static void
readWitImage(FILE *fp)
{
    int  type, bpp, w, h, orgW, orgH;
    int  n;
    int  size;
    void *data;
    char objString[80];

    fread(&type, 4, 1, fp);
    fread(&bpp, 4, 1, fp);
    fread(&w, 4, 1, fp);
    fread(&h, 4, 1, fp);
    fread(&orgW, 4, 1, fp);
    fread(&orgH, 4, 1, fp);
    switch(type) {
    case WIT_UINT8:
    case WIT_INT8:
    case WIT_UINT16:
    case WIT_INT16:
    case WIT_FLOAT:
        n = 1;
        break;
    case WIT_COMPLEX:
        n = 2;
        break;
    case WIT_RGB:
        n = 3;
        break;
    }
    size = w*h*bpp*n;
    if(!(data = (void *)malloc(size))) {
        fprintf(stderr, "unable to allocate %d bytes\n", size);
        exit (1);
    }
    fread(data, size, 1, fp);
}

```





