The Ipe manual

Otfried Cheong

June 5, 2004

1 Welcome to the Wonderful World of Ipe!

... where making pictures is as easy as `$\pi$` ...

Preparing figures for a scientific article is a time-consuming process. If you are using the LaTeX document preparation system in an environment where you can include (encapsulated) Postscript figures or PDF figures, then the extendible drawing editor Ipe may be able to help you in the task. Ipe allows you to prepare and edit drawings containing a variety of basic geometry primitives like lines, splines, polygons, circles etc.

Ipe also allows you to add text to your drawings, and unlike most other drawing programs, Ipe treats these text object as LaTeX text. This has the advantage that all usual LaTeX commands can be used within the drawing, which makes the inclusion of mathematical formulae (or even simple labels like "q_i") much simpler. Ipe processes your LaTeX source and includes its Postscript or PDF rendering in the figure.

In addition, Ipe offers you some editing functions that can usually only be found in professional drawing programs or CAD systems. For instance, it incorporates a context sensitive snapping mechanism, which allows you to draw objects meeting in a point, having parallel edges, objects aligned on intersection points of other objects, rectilinear and c-oriented objects and the like. Whenever one of the snapping modes is enabled, Ipe shows you *Fifi*, a secondary cursor, which keeps track of the current aligning.

One of the nicest features of Ipe is the fact that it is *extensible*. You can easily write your own functions, so-called *ipelets*. Once registered with Ipe by adding them to your ipelet path, you can use those functions like Ipe's own editing functions. (In fact, some of the functions in the standard Ipe distribution are actually implemented as ipelets.) Among others, there is an ipelet to compute Voronoi diagrams.

Making an on-line presentation, or just slides for a presentation, is another task that requires drawing figures. You can use Ipe to prepare presentations in PDF format, and either print them on transparencies or display them using Acrobat Reader using a beamer during the presentation. Ipe gives you access to the presentation-features of Acrobat Reader, such as automatic advance to the next page and fancy transition modes.

Ipe tries to be self-explanatory. There is online help available, and most commands tell you about options, shortcuts, or errors. Nevertheless, it would probably be wise to read at least a few sections of this manual. Strongly suggested is the chapter on general concepts, and the chapter that explains the snapping functions. If you want to use Ipe to prepare slides, you should also read the chapter on page views. And if you have used Ipe 5 before, please read this.

2 About Ipe files

Ipe 6.0 creates (Encapsulated) Postscript or PDF files. These files can be used in any way that PDF or Postscript files are used, such as viewed with Ghostview, with Acrobat Reader or Xpdf,

edited with Acrobat, or included in Latex/Pdflatex documents. However, Ipe cannot read arbitrary Postscript or PDF files, only files it has created itself. This is because files created by Ipe contain a special hidden stream that describes the Ipe objects. (So if you edit your Ipe-generated PDF file in a different program such as Adobe Acrobat, Ipe will not be able to read the file again afterwards.)

You decide in what format to store a figure when saving it for the first time. Ipe gives you the option of saving with extensions "eps" (Encapsulated Postscript), "ps" (Postscript), "pdf" (PDF), and "xml" (XML). Note that only documents of a single page can be stored in Encapsulated Postscript format, as this format doesn't support multi-page documents. Files saved with extension "xml" are—obviously—XML files and contain no Postscript of PDF information. The precise XML format used by Ipe is documented later in this manual. XML files can be read by any XML-aware parser, and it is easy for other programs to generate XML output to be read by Ipe. You probably don't want to keep your figures in XML format, but it is excellent for communicating with other programs, and for converting figures between programs.

There are perhaps two major uses for Ipe documents. The first is for inclusion into Latex documents, the second is for making presentations. There isn't much to be said about the second use: You create a PDF file with Ipe, and either print it on transparencies or present it using a laptop with Acrobat Reader. You should read the section on page views if you plan to make on-line presentations, as Ipe now allows you to create pages that are displayed incrementally in Acrobat Reader.

So let's concentrate on the first and original use of Ipe documents, inclusion in Latex documents. Most Latex installations support the inclusion of figures in Encapsulated Postscript (EPS) format (the "Encapsulated" means that there is only a single Postscript page and that it contains a bounding box of the figure).

The standard way of including EPS figures is using the `graphicx` package. If you are not familiar with it, here is a quick overview. In the preamble of your document, add the declaration:

```
\usepackage{graphicx}
```

One useful attribute to this declaration is `draft`, which stops LaTeX from actually including the figures—instead, a rectangle with the figure filename is shown:

```
\usepackage[draft]{graphicx}
```

To include the figure "figure1.eps", you use the command:

```
\includegraphics{figs/figure1}
```

Note that it is common *not* to specify the file extension ".eps". The command `\includegraphics` has various options to scale and rotate the figure. For instance, to scale the same figure to 50%, use:

```
\includegraphics[scale=0.5]{figs/figure1}
```

To scale such that the width of the figure becomes 5 cm:

```
\includegraphics[width=5cm]{figs/figure1}
```

Instead, one can specify the required height with `height`.

Here is an example that scales a figure to 200% and rotates it by 45 degrees counter-clockwise. Note that the scale argument should be given *before* the `angle` argument.

```
\includegraphics[scale=2,angle=45]{figs/figure1}
```

Let's stress once again that these commands are the standard commands for including EPS files in a LaTeX document. Unlike in previous versions of Ipe, Ipe files neither require nor support any

special treatment.[1] If you are used to other commands for EPS inclusion, such as the old-fashioned `epsfig` package,[2] you can use them as well for Ipe figures. If you want to know more about the LaTeX packages for including graphics and producing colour, check the `grfguide.tex` document that is probably somewhere in your TeX installation.

If you are a user of Pdflatex (a version of Latex that produces PDF instead of DVI output), you cannot include EPS files. Instead, save your Ipe figures in PDF format, and include them in the way described above.

Unfortunately, versions of Pdflatex earlier than 1.10 have a problem including PDF figures. Each page of a PDF document can carry several "bounding boxes", such as the *MediaBox* (which indicates the paper size), the *CropBox* (which indicates how the paper will by cut), or the *ArtBox* (which indicates the extent of the actual contents of the page). Ipe automatically saves, for each page, the paper size in the *MediaBox*, and a bounding box for the drawing in the *ArtBox*. Versions of Pdflatex earlier than 1.10, however, look at the *CropBox*, or, if the *CropBox* is not set, the *MediaBox*. To include PDF figures using an earlier Pdflatex-version, you therefore have to instruct Ipe to include a *CropBox* by ticking the *Use CropBox* checkbox in the *Document properties* (in the *Edit* menu). (This is currently the default for new documents. The only disadvantage is that Acrobat Reader will not display full pages in documents saved with this option, so when making PDF presentations you probably want to untick this option.)

If you have Pdflatex 1.10 or higher, you can also solve the problem by including this line in the preamble:

```
\expandafter\ifx\csname pdfoptionalwaysusepdfpagebox\endcsname\relax\else
\pdfoptionalwaysusepdfpagebox5
\fi
```

(Note that this will simply be ignored if you are using normal LaTeX or an older version of Pdflatex.)

You can save all your figures in both EPS and PDF format, so that you can run both Latex and Pdflatex on your document—when including figures, Latex will look for the EPS variant, while Pdflatex will look for the PDF variant. (Here it comes in handy that you didn't specify the file extension in the `\includegraphics` command.)

You may find it cumbersome to save an Ipe figure in both formats each time you modify it. If so, you can always save in, say, EPS format, and automate the conversion to PDF by writing a shell script or batch file that calls ipetoipe to do the conversion.

On the other hand, if you *only* use Pdflatex, you might opt to exploit a feature of Pdflatex: You can keep all the figures for a document in a single, multi-page Ipe document, with one figure per page. You can then include the figures one by one into your document by using the `page` argument of `\includegraphics`.

For example, to include page 3 from the PDF file "figures.pdf" containing several figures, you could use

```
\includegraphics[page=3]{figs/figures}
```

3 Command line options and auxiliary programs

Ipe command line options Ipe supports the following two options:

`-sheet` ***style sheet name*** Adds the designated style sheet to any newly created documents.

`-geom` ***WxH+X+Y*** Places the Ipe main windows at the desired position and size. (Note the slight difference with the standard Unix option `-geometry`).

[1] In particular, the `ipe.sty` package will not work with figures made with Ipe 6.

[2] In fact, in modern LaTeX installations, `epsfig.sty` is just a small wrapper around `graphics.sty`.

In addition, you can specify the name of an Ipe file to open on the command line. Finally, Ipe also understands some standard X11 options on Unix (this is support built into the Qt library, see there for details).

ipetoipe: converting Ipe file formats The auxiliary program *ipetoipe* converts between the different Ipe file formats:

```
ipetoipe ( -xml | -pdf | -eps | -ps ) [ -export ] infile outfile
ipetoipe -png <page> <resolution> infile outfile
```

For example, the command line syntax

```
ipetoipe -pdf figure1.eps figure1.pdf
```

converts `figure1.eps` to PDF format.

When you use the `-export` flag, no Ipe markup is included in the resulting output file. Ipe will not be able to open a file created that way, so make sure you keep your original!

With the option `-png` ipetoipe converts a page of the document to a bitmap in PNG format. (Of course the result contains no Ipe markup, so make sure you keep your original.) For instance, the following command line

```
ipetoipe -png 3 150.0 presentation.pdf pres3.png
```

converts page 3 of the Ipe document `presentation.pdf` to a bitmap, with resolution 150.0 pixels per inch.

pdftoipe: Importing Postscript and PDF You can convert arbitrary Postscript or PDF files into Ipe documents, making them editable. The auxiliary program *pdftoipe* converts (pages from) a PDF file into an Ipe XML-file. (If your source is Postscript, you have to first convert it to PDF using Acrobat Distiller or *ps2pdf*.) Once converted to XML, the file can be opened from Ipe as usual.

The conversion process should handle any graphics in the PDF file fine, but doesn't do very well on text—Ipe's text model is just too different.

4 General Concepts

After you start up Ipe, you will see a window with a large gray area containing a light yellow rectangle. This area, the *canvas*, is the drawing area where you will create your figures. The light yellow rectangle is your "sheet of paper", the first page of your document. (While Ipe doesn't stop you from drawing outside the paper, such documents generally do not print very well.)

At the top of the window, above the canvas, you find the following toolbars:

- File tools (new document, load, save, as well as cut and paste),
- Resolution tools (screen resolution selection),
- Page tools (page and view number),
- Color tools (stroke and fill color),
- Line tools (line width, dash style, arrow shape and size),
- Mark and text size settings,
- Snap tools (snapping modes, grid size and angular resolution),

- Mode tools (mode selection).

On the left hand side of the canvas you find a list of the *layers* of the current page.

All user interface elements have tool tips—if you move the mouse over them and wait a few seconds, a short explanation will appear. For a longer explanation, use the *"What's This?"* button on the file tools toolbar (it looks like a cursor pointer with a question mark). Press the button, then click with the mouse on any part of the user interface for a short explanation.

The mode toolbar allows you to set the current *Ipe mode*. Roughly speaking, the mode determines what the left mouse button will do when you click it in the figure. The leftmost five buttons select modes for selecting and transforming objects, the remaining buttons select modes for creating new objects.

Holding the Control key and pressing the right mouse button, by the way, pops up the object attribute menu in any mode.

In this chapter we will discuss the general concepts of Ipe. Understanding these properly will be essential if you want to get the most out of Ipe.

4.1 Order of objects

An Ipe drawing is a sequence of geometric objects. The order of the objects is important—wherever two objects overlap, the object which comes first in Ipe's sequence will hide the other ones. When new objects are created, they are added in *front* of all other objects. However, you can change the position of an object by putting it in front or in the back, using the "Front" and "Back" functions in the *Edit* menu.

4.2 The current selection

Whenever you call an Ipe function, you have to specify which objects the function should operate on. This is done by *selecting* objects. The selected objects (the *selection*) consists of two parts: the *primary* selection consists of exactly one object (of course, this object could be a group). Any additional selected objects form the *secondary* selection. Some functions (like *Show properties*) operate only on the primary selection, while others treat primary and secondary selections differently (the align functions, for instance, align the secondary selections with respect to the primary selection.)

The selection is shown by outlining the selected object in color. Note that the primary selection is shown with a slightly different look.

The primary and secondary selections can be set in selection mode. Clicking the left mouse button close to an object makes that object the primary selection and deselects all other objects. If you keep the Shift key pressed while clicking with the left mouse key, the object closest to the mouse will be added to or deleted from the current selection. You can also drag a rectangle with the mouse—when you release the mouse button, all objects inside the rectangle will be selected.

To make it easier to select objects that are below or close to other objects, it is convenient to understand exactly how selecting objects works. In fact, when you press the mouse button, a list of all objects is computed that are sufficiently close to the mouse position (the exact distance can be set as the *Select distance* in the *preferences dialog*). This list is then sorted by increasing distance from the mouse and by increasing depth in the drawing. If Shift was not pressed, the current selection is now cleared. Then the first object in the list is presented. Now, while still keeping the mouse button pressed, you can use the Space key to step through the list of objects near the mouse in order of increasing depth and distance. When you release the right mouse button, the object is selected (or deselected).

When measuring the distance from the mouse position to objects, Ipe considers the boundary of objects only. So to select a filled object, don't just click somewhere in its interior, but close to its boundary.

Another way to select objects is using the *Select all* function from the *Edit* menu. It selects all objects on the page. Similarly, the *Select all in layer* function in the *Layer* menu selects all objects in the active layer.

4.3 Moving and scaling objects

There are three modes for transforming objects: *move* (translate), *stretch*, and *rotate*. If you hold the shift key while pressing the left mouse button, the stretch function keeps the aspect ratio of the objects (an operation we call *scaling*), and the move function is restricted to horizontal and vertical translations.

Normally, the transformation functions work on the current selection. However, to make it more convenient to move around many different objects, there is an exception: When the mouse button is pressed while the cursor is not near the current selection, but there *is* some other object close to the cursor, *that* object is moved, rotated, or scaled instead.

By default, the *rotate* function rotates around the center of the bounding box for the selected objects. This behavior can be overridden by specifying an axis system (Section 6.3). If an axis system is set, then the origin is used as the center.

The *scale* and *stretch* functions use a corner of the bounding box for the selected objects as the fix-point of the transformation. Again, if an axis system is set, the origin of the system is used instead.

It is often convenient to rotate or scale around a vertex of an object. This is easy to achieve by setting the origin of the axis system to that vertex, using the *Snap to vertex* function for setting the axis system.

4.4 Stroke and fill colors

Most Ipe objects can have two different colors, one for the boundary and one for the interior of the object. The Postscript terms *stroke* and *fill* are used to denote these two colors. They can be selected independently in the *Color* toolbar. You can also set stroke and fill to be *void*. A void stroke color means that no outline of the object is drawn, a void fill color means that no interior will be drawn. Setting both colors to void will render an object invisible. Imagine preparing a drawing by hand, using a pen and black ink. What Ipe draws in its *stroke* color is what you would stroke in black ink with your pen. Probably you would not use your pen to fill objects, but you would use a brush, and maybe even a different kind of paint like water color. Well, the *fill* color is Ipe's "brush".

This explains why text objects, mark objects, and arrows only use the stroke color, even for the filled marks (discs and squares) and filled arrows. You would also use a pen for these details, not the brush (unless you draw very large marks—in which case you probably meant to draw a filled *circle* anyway).

An interesting exception to the above rules are lines with arrows. If a line with an arrow has stroke color *void*, the arrow will be drawn with the fill color, and the line will not be drawn at all. This is useful to create arrowheads without body, which can be used to be attached to objects that cannot have arrows.

4.5 Line width, line dash pattern, and arrows

The *Line* toolbar permits setting the dash-dot pattern (solid line, dashed, dotted etc.) as well as the width of lines. This has effect for the boundaries of *path objects*, such as polygons and polygonal lines, splines, circles and ellipses, rectangles and circular arcs. It does not effect text or marks.

Line width is given in Postscript points (1/72 inch). A good value is something around 0.4 or 0.6.[3]

On the same toolbar you can set the current arrow mode and arrow size. Only polygonal lines, splines, and circular arcs can have arrows.

Arrows are by default drawn as triangles in the stroke color. Note that they extend beyond the endpoint of the line by an amount linear in the line width. If that is a problem, you can duplicate the object, give one copy the arrows and line width zero, and move the endpoints of the other copy slightly so they fall in the interior of the arrows of the first copy.

4.6 Symbolic and absolute attributes

Attributes such as color, line width, dash style, mark size, or text size, can be either absolute (a number, or a set of numbers specifying a color) or symbolic (a name). Symbolic attributes must be translated to absolute values by a *style sheet* for rendering.

One purpose of style sheets is to be able to reuse figures from articles in presentations. Obviously, the figure has to use much larger fonts, markers, arrows, and fatter lines in the presentation. If the original figure used symbolic attributes, this can be achieved by simply swapping the style sheet for another one.

The Ipe user interface can be switched between displaying absolute and symbolic attributes (using *Absolute attributes* in the *View* menu). When creating an object, it takes its attributes from the current user interface settings, so if you are in symbolic mode, the object gets symbolic attributes, otherwise it gets absolute attributes. Absolute attributes have the advantage that you are free to choose any value you wish, including picking arbitrary colors using a color chooser. In symbolic mode, you can only use the choices provided by the current *style sheet*.

The choices for symbolic attributes provided in the Ipe user interface are taken from your style sheet. The standard style sheet is deliberately short, to encourage users to figure out how to make their own.

The settings for grid size and axis angle can be switched between symbolic and absolute independently. The current setting has no influence on objects being created.

4.7 Zoom and pan

You can zoom in and out the current drawing by changing the resolution in the *Resolution* toolbar. The number displayed there is the number of pixels that correspond to one inch in your document.

Related are the functions *Normal size* (which sets the resolution to 72 pixels per inch), *Fit page* (which chooses the resolution so that the current page fills the canvas), *Fit objects* (which chooses the resolution such that the objects on the page fill the screen), and *Fit selection* (which does the same for the selected objects only). All of these are in the *Zoom* menu.

You can *pan* the drawing either with the mouse in *Pan* mode, or by pressing the "." (period) key ("here") with the mouse anywhere on the canvas. The drawing is then panned such that the cursor position is moved to the center of the canvas. This shortcut has the advantage that it also works while you are in the middle of any drawing operation. Since the same holds for the *zoom in* and *zoom out* buttons and keys, you can home in on any feature of your drawing *while* you are adding or editing another object.

4.8 Groups

It is often convenient to treat a collection of objects as a single object. This can be achieved by *grouping* objects. The result is a geometric object, which can be moved, scaled, rotated etc. as

[3]The line width can be set to zero to get the thinnest line the device can produce (i.e. approximately the same as 0.15 for a 600 dpi printer or 0.3 for a 300 dpi printer). The PDF and Postscript authorities discourage using this feature, since it makes your Postscript files device-dependent.

a whole. To edit its parts or to move parts of it with respect to others, however, you have to *un-group* the object, which decomposes it into its component objects. To un-group a group object, select it, bring up the object menu, and select the *Ungroup* function.

Group objects can be elements of other groups, so you can create a hierarchy of objects.

4.9 Layers

A page of an Ipe document consists of one or more layers. Each object on the page belongs to a layer. There is no relationship between layers and the back-to-front ordering of objects, so the layer is really just an attribute of the object.

Layers have several attributes. They may be editable or locked. Objects can be selected and modified only if their layer is editable. A layer may be visible, invisible, or dimmed. A layer may have snapping on or off—objects will behave magnetically only if their layer has snapping on.

When editing a page in Ipe, there is one *active layer*. New objects are always created in the active layer.

Layers are also used to create pages that are displayed incrementally in Acrobat Reader. Once you have distributed your objects over various layers, you can create *page views*, which defines in what order which layers of the page are shown.

4.10 Mouse shortcuts

For the beginner, choosing a selection or transformation mode and working with the left mouse button is easiest. Frequent Ipe users don't mind to remember the following shortcuts, as they allow you to perform panning, and transformations without leaving the current mode:

	Left Mouse	Middle Mouse
Plain	(*)	move
Shift	(*)	pan
Ctrl	stretch	rotate
Ctrl+Shift	scale	move horizontal/vertical

The fields marked (*) depend on the current mode.

The right mouse button is used to select objects (just like the left mouse button would do when in *Select* mode). Holding the Control key when pressing the right mouse button brings up the object menu.

If you have to use Ipe with a two-button mouse, you obviously cannot use the shortcuts for moving, panning, and rotating, and will have to use the mode buttons. In all other situations where you would normall use the middle mouse button (for instance, to move a vertex when editing a path object), you can hold the Shift-key and use the right mouse button.

5 Object types

Ipe supports six different types of geometric objects, namely path objects (which includes all objects with a stroked contour and filled interior, such as (poly)lines, polygons, splines, splinegons, circles and ellipses, circular and elliptic arcs, and rectangles), mark objects, text objects, image objects, group objects, and reference objects (which means that a template is reused at a certain spot).

Most objects are created by clicking the left mouse button somewhere on the canvas in the right Ipe mode. Group objects are created using the *Group* function on the *Edit* menu, and image objects can be added to the document using the *Insert image* ipelet.

5.1 Path objects

Path objects are defined by a set of *subpaths*, that is, curves in the plane.[4] Each subpath is either open or closed, and consists of straight line segments, circular or elliptic arc segments, parabola segments, and cubic B-spline segments. The curves are drawn with the stroke color, dash style, and line width; the interior of the object specified is filled using the fill color.

The distinction between open and closed subpaths is meaningful for stroking only, for filling any open subpath is implicitely closed. Stroking a set of subpaths is identical to stroking them individually. This is not true for filling: using several subpaths, one can construct objects with holes, and more complicated pattern. The filling algorithm is the *even-odd rule* of Postscript/PDF: To determine whether a point lies inside the filled shape, draw a ray from that point in any direction, and count the number of path segments that cross the ray. If this number is odd, the point is inside; if even, the point is outside.

Ipe can draw arrows on the first and last segment of a path object, but only if that segment is part of an open subpath.

There are several Ipe modes that create path objects in different ways. All modes create an object consisting of a single subpath only. To make more complicated path objects, such as objects with holes, you create each boundary component separately, then select them all and use the *Compose paths* function in the *Edit* menu. The reverse operation is *Decompose path*, you find it in the object menu of a path object that has several subpaths.

You can also create complicated paths by joining curves sequentially. For this to work, the endpoint of one path must be (nearly) identical to the begin point of the next—easy to achieve using snapping. You select the object you wish to join, and call *Join paths* in the *Edit* menu. You can also join several open path objects into a single closed path this way.

Circles can be entered in three different ways. To create an ellipse, create a circle and stretch and rotate it. Circular arcs can be entered by clicking three points on the arc or by clicking the center of the supporting circle as well as the begin and end vertex of the arc. They can be filled in Postscript fashion, and can have arrows. You can stretch a circular arc to create an elliptic arc.

A common application for arcs is to mark angles in drawings. The snap keys are useful to create such arcs: set arc creation to *center & 2 pts*, select *snap to vertex* and *snap to boundary*, click first on the center point of the angle (which is magnetic) and click then on the two bounding lines.

There are two modes for creating more complex general path objects. The difference between the *line* mode and the *polygon* mode is that the first creates an open path, the latter generates a closed one. As a consequence, the *line* mode uses the current arrow settings, while the *polygon* mode doesn't. Also, objects created in *line* mode will only have the stroke color set, even if the current fill color is not void. (However, if the current stroke color is void and the fill color is not, it will be filled.)

The path created consists of segments of various types. The initial setting is to create straight segments. By holding the shift-key when pressing the left mouse button one can switch to uniform B-splines. One can also add parabolic segments and circular arcs to the path by pressing the "q" and "a" keys. The mode for splines is identical to the polyline mode, but starts in the uniform B-spline setting.

For the mathematically inclined, a more precise description of the segments that can appear on a subpath follows. More details can be found in Foley et al.[5] and other text books on splines.

A subpath consists of a sequence of segments. Each segment is either a straight line segment, an elliptic arc, a quadratic Bézier spline, a cubic Bézier spline, or a uniform cubic B-spline.

[4] Unlike Ipe 5, we can now represent objects consisting of more than one subpath. Note that Ipe 5's spline and arc objects are now represented as IpePath objects.

[5] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990.

The quadratic Beziér spline defined by control points p_0, p_1, and p_2, is the curve

$$P(t) = (1-t)^2 p_0 + 2t(1-t)p_1 + t^2 p_2,$$

where t ranges from 0 to 1. This implies that it starts in p_0 tangent to the line p_0p_1, ends in p_2 tangent to the line p_1p_2, and is contained in the convex hull of the three points. Any segment of any parabola can be expressed as a quadratic Beziér spline.

For instance, the piece of the unit parabola $y = x^2$ between $x = a$ and $x = b$ can be created with the control points

$$\begin{aligned} p_0 &= (a, a^2) \\ p_1 &= (\frac{a+b}{2}, ab) \\ p_2 &= (b, b^2) \end{aligned}$$

Any piece of any parabola can be created by applying some affine transformation to these points.

The cubic Beziér spline with control points p_0, p_1, and p_2 is the curve

$$R(t) = (1-t)^3 p_0 + 3t(1-t)^2 p_1 + 3t^2(1-t)p_2 + t^3 p_3.$$

It starts in p_0 being tangent to the line p_0p_1, ends in p_3 being tangent to the line p_2p_3, and lies in the convex hull of the four control points.

Uniform cubic B-splines approximate a series of $m+1$ control points p_0, p_1, ..., p_m, $m \geq 3$, with a curve consisting of $m-2$ cubic polynomial curve segments s_0, s_1, ..., s_{m-3}. Every such curve segment is defined by four of the control points. In fact, curve segment s_i is defined by the points p_i, p_{i+1}, p_{i+2}, and p_{i+3}. If the curve is closed (a *splinegon*), it contains three additional curve segments s_{m-2}, s_{m-1}, and s_m, defined by appending p_0, p_1, and p_2 to the end of the sequence. A uniform B-spline segment on an open subpath of an Ipe path object is defined by repeating both the first and last control point three times, so as to make the segment begin and end in these points.

The segment s_i is the cubic curve segment with the following parametrization.

$$Q(t) = \frac{(1-t)^3}{6} p_i + \frac{3t^3 - 6t^2 + 4}{6} p_{i+1} + \frac{-3t^3 + 3t^2 + 3t + 1}{6} p_{i+2} + \frac{t^3}{6} p_{i+3},$$

where t ranges from 0 to 1.

Since the point $Q(t)$ is a convex combination of the four control points, the curve segment s_i lies in the convex hull of p_i to p_{i+3}. Furthermore, it follows that any affine transformation can be applied to the curve by applying it to the control points. Note that a control point p_i has influence on only four curve segments, s_{i-3}, s_{i-2}, s_{i-1}, and s_i. Thus, when you edit a spline object and move a control point, only a short piece of the spline in the neighborhood of the control point will move.

5.2 Text objects

Text objects come in two flavors: simple *labels*, and *minipages*. There are two variants of these: *titles* (a label that serves as the title of the page), and *textbox* (a minipage that spans the entire width of the page).

The position you have to click to start creating a *label* object is the lower left corner of the piece of text. A popup window appears where you can enter Latex source code.

A *minipage* object is different from a simple text object in that its width is part of its definition. When you create a minipage object, you first have to drag out a horizontal segment for the minipage. This is used as the top edge of the minipage—it will extend downwards as far as necessary to accomodate all the text. Minipages are formatted using, not surprisingly, Latex's

`minipage` environment. Latex tries to fill the given bounding box as nicely as possible. It is possible to include center environments, lemmas, and much more in minipages.

To create a *textbox* object, simply press "F10". Ipe automatically places the object so that it spans the entire width of the page (the *margins* settings in the style sheet determine how much space is left on the sides), and places it vertically underneath the textboxes already on the page. This is particularly convenient for creating presentations with a lot of text, or with items that appear one by one.

(*Title* objects are not yet supported.)

You can use any LaTeX-command that is legal inside a `\makebox` (for labels) or inside a `minipage` (for minipages). You cannot use commands that involve a non-linear translation into PDF, such as commands to generate hyperlinks or to include external images.

You can use color in your text objects, using the `\textcolor` command, like this:

```
This is in black. \textcolor{red}{This is in red.} This is in black.
```

All the symbolic colors of your current style sheet are also available as arguments to `\textcolor`. You can also use absolute colors, for instance:

```
This is in black. \textcolor[rgb]{1,1,0}{This is in yellow.} This is in black.
```

If you need LaTeX-commands that are defined in additional LaTeX packages, you can include (`\usepackage`) those in the LaTeX preamble, which can be set in *Document properties* in the *Edit* menu.

After you have created or edited a text object, the Ipe screen display will show the beginning of the Latex source. You can select *Run Latex* from the *File* menu to create the PDF/Postscript representation of the object. This converts all the text objects in your document at once, and Ipe will display a correct rendition of the text afterwards.

If the Latex conversion process results in errors, Ipe will automatically show you the log file created by the Latex run. If you cannot figure out the problem, look in the section on troubleshooting (Section 10).

You can use Unicode text, such as accented characters, Greek, Cyrillic, Chinese, Japanese, or Korean, in your text objects, once you have set up the necessary style files and fonts (Section 12).

5.3 Mark objects

Mark's objects are useful to mark points in your drawing. They come with different looks (little circles, discs, squares, boxes, or crosses). Note that marks behave quite different from other objects. In particular, you should not confuse a disc mark with a little disc created as a circle object:

- a mark only obeys the stroke color
- when you scale a mark, it will not change its size (you can change the mark size from the configuration panel, though)
- the bounding box of a mark only contains the mark's center
- when you rotate a mark, it does not change its orientation

You can change a mark's type and size later.

Sometimes you may wish you had a mark with, say, black boundary and white interior (because you want to place it on some dark, but irregular background). The best way to achieve this is to place *two* marks, a white disc and a black circle on top of each other. You could either group one such double-mark and then paste it to the desired locations (the pasting mechanism will put the center of the mark at the point location, so you can work as usual with marks), but you can also just place the white discs. Then set *snap to vertex* on, and place the black circles. Or you may want to install such a marker as a template.

5.4 Image objects

Images are inserted using the *Insert image* ipelet. Once in a drawing, you can scale, stretch, and rotate an image. You can read in some scanned drawing and draw on top of it within Ipe. This is useful if you have a drawing on paper and want to make an Ipe version of it.

Note that there are two separate functions for inserting JPEG images and inserting "other" images. JPEG images are literally included in the PDF output (using PDF's `DCTDecode` filter). "Other" images are stored as a pixel array, using Flate compression. Note that on some systems the function for "other" images will also accept JPEG files—obviously, the resulting PDF files will be much larger than if they had been included as JPEG images.

Images are stored most efficiently in PDF format. It is reasonable to create PDF presentations with lots of JPEG photographs in Ipe. While the screen display is too slow, Ipe does create good quality PDF files for these. Storing images in (Encapsulated) Postscript, however, cannot be recommended for larger images, as each image has to be embedded twice!

5.5 Group objects

Little needs to be said about group objects. You create them by selecting any number of objects and using the *Group* function from the *Edit* menu. The group objects then behave like one. To modify a group object, it has to be decomposed into its parts using *Ungroup*.

5.6 Reference objects and templates

A reference object stores nothing more than the name of a template. This name is looked up in the document's style to retrieve an object (a *template*) to be displayed. The reference can be moved, rotated, and stretched, but that's it—to modify it in any other way, the original template in the style sheet has to be modified.

If a templated named "background" exists in your style sheet, a reference to it is automatically inserted to each new page of your document.

6 Snapping

One of the nice features of Ipe is the possibility of having the mouse *snap* to other objects during entry or moving. Certain features on the canvas become "magnetic", and it is very easy to align objects to each other, to place new objects properly with respect to the present objects and so on.

Snapping comes in three flavors: *grid snapping*, *context snapping*, and *angular snapping*.

In general, you turn a snapping mode on by pressing one of the buttons in the *Snap* toolbar, or selecting the equivalent functions in the Snap menu. The buttons are independent, you can turn them on and off independently. (The snapping modes, however, are not independent. See below for the precise interaction.) The keyboard shortcuts are rather convenient since you will want to toggle snapping modes on and off while in the middle of creating or editing some object.

Whenever one of the snapping modes is enabled, you will see a little cross near the cursor position. This is the secondary cursor *Fifi*.[6] Fifi marks the position the mouse is snapped to.

6.1 Grid snapping

Grid snapping is easy to explain. It simply means that the mouse position is rounded to the nearest grid point. Grid points are points whose coordinates are integer multiples of the *grid size*, which can be set in the box in the *Snap* field. You have a choice from a set of possible grid sizes. The units are Postscript points (in LaTeX called `bp`), which are equal to 1/72 of an inch.

[6] Fifi is called after the dog in the `rogue` computer game installed on most Unix systems in the 1980's, because it also keeps running around your feet.

You can ask Ipe to show the grid points by selecting the function *Grid visible* from the *View* menu. The same function turns it off again.

6.2 Context snapping

When *context snapping* is enabled, certain features of the objects of your current drawing become magnetic. There are three buttons to enable three different features of your objects: vertices, the boundary, and intersection points.

When the mouse is too far away from the nearest interesting feature, the mouse position will not be "snapped". The snapping distance can be changed by setting *Snapping distance* value in the preference dialog. If you use a high setting, you will need to toggle snapping on and off during drawing. Some people prefer to set snapping on once and for all, and to set the snap distance to a very small value like 3 or 4.

The features that you can make "magnetic" are the following:

vertices are vertices of polygonal objects, control points of multiplicity three of splines, centers of circles and ellipses, centers and end points of circular arcs, and mark positions.

boundaries are the object boundaries of polygonal objects, splines and splinegons, circles and ellipses, and circular arcs.

intersections are the intersection points between edges of polygonal objects, circles, or circular arcs.[7] Note that intersection points involving splines or ellipses are not recognized.

6.3 Angular snapping

When *angular snapping* is enabled, the mouse position is restricted to lie on a set of lines through the *origin* of your current *axis system*. The lines are the lines whose angle with the *base direction* is an integer multiple of the snap angle. The snap angle can be set in the second box in the Snap toolbar. The values are indicated in degrees. So, for a snapping angle of 45°, we get the snap lines indicated in Figure 1. (In the figure the base direction—indicated with the arrow—is assumed horizontal.)

For a snap angle of 180 degrees, snapping is to a single line through the current origin.

In order to use angular snapping, it is important to set the axis system correctly. To set the origin, move the mouse to the correct position, and press the `F1`-key. Note that angular snapping is *disabled* while setting the origin. This way you can set a new origin for angular snapping without leaving the mode first. Once the origin has been set, the base direction is set by moving to a point on the desired base line, and pressing the `F2`-key. Again, angular snapping is disabled. Together, origin and base direction determine the current *axis system*. Remember that the origin is also used as the fix-point of scale, stretch, and rotate operations, if it is set.

You can un-set the current axis system by pressing `Shift-F1`. This also turns off angular snapping.

You can set origin and base direction at the same time by pressing `F3` when the mouse is very near (or snapped to) an edge of a polygonal object. The origin is set to an endpoint of the edge, and the base direction is aligned with it. This is useful to make objects parallel to a given edge.

For drawing rectilinear or c-oriented polygons, the origin should be set to the previous vertex at every step. This can be done by pressing `F1` every time you click the left mouse button, but that would not be very convenient. Therefore, Ipe offers a second angular snap mode, called *automatic angular snapping*. This mode uses an independent origin, which is automatically set every time you add a vertex when creating a polygonal object. Note that while the origin is independent of the origin set by `F1`, the base direction and the snap angle used by automatic angular snapping is

[7] Snapping to intersections involving circles are not yet implemented.

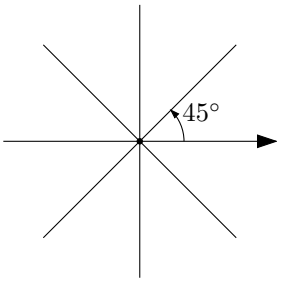


Figure 1: Snap lines

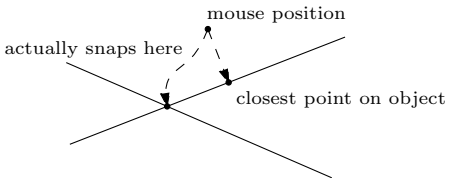


Figure 2: Snapping priorities

the same as for angular snapping. Hence, you can align the axis system with some edge of your drawing using `F3`, and then use automatic angular snapping to draw a new object that is parallel or orthogonal to this edge.

This snapping mode has another advantage: It remains silent and ineffective until you start creating a polygonal object. So, even with automatic angular snapping already turned on, you can still freely place the first point of a polygon, and then the remaining vertices will be properly aligned to make a c-oriented polygon.

The automatic angular snapping mode is never active for any non-polygonal object. In particular, to *move* an object in a prescribed direction, you have to use normal angular snapping.

A final note: Many things that can be done with angular snapping can also be done by drawing auxiliary lines and using context snapping. It is mostly a matter of taste and exercise to figure out which mode suits you best.

6.4 Interaction of the snapping modes

Not all the snapping modes can be active at the same time, even if all buttons are pressed. Here we have a close look at the possible interactions, and the priorities of snapping.

The two angular snapping modes restrict the possible mouse positions to a *one-dimensional* subspace of the canvas. Therefore, they are incompatible with the modes that try to snap to a zero-dimensional subspace, namely vertex snapping, intersection snapping, and grid snapping. Consequently, when one of the angular snapping modes is *on*, vertex snapping, intersection snapping, and grid snapping are ineffective.

On the other hand, it is reasonable to snap to boundaries while in an angular snapping mode, and this function is actually implemented correctly. When both angular and boundary snapping are *on*, Ipe will compute intersections between the snap lines with the boundaries of your objects, and whenever the mouse position *on* the snap line comes close enough to an intersection, the mouse is snapped to that intersection.

The two angular snapping modes themselves can also coexist in the same fashion. If both angular and automatic angular snapping are enabled, Ipe computes the intersection point between the snap lines defined by the two origins and snaps there. It the snap lines are parallel or coincide, automatic angular snapping is used.

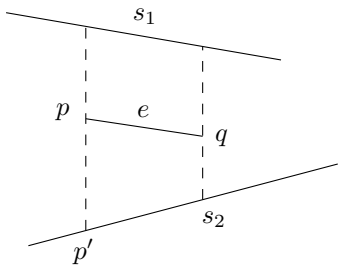
When no angular snapping mode is active, Ipe has three priorities. First, Ipe checks whether the closest vertex or intersection point is close enough. If that is not the case, the closest boundary edge is determined. If even that is too far away, Ipe uses grid snapping (assuming all these modes are enabled).

Note that this can actually mean that snapping is *not* to the *closest* point on an object. Especially for intersections of two straight edges, the closest point can never be the intersection point! See Figure 2.

6.5 Examples

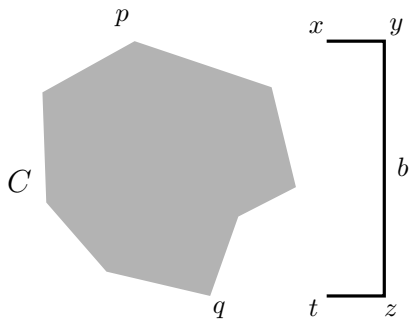
It takes some time and practice to feel fully at ease with the different snapping modes, especially angular snapping. Here are some examples showing what can be done with angular snapping.

Example 1: We are given segments s_1, s_2, and e, and we want to add the dashed vertical extensions through p and q.



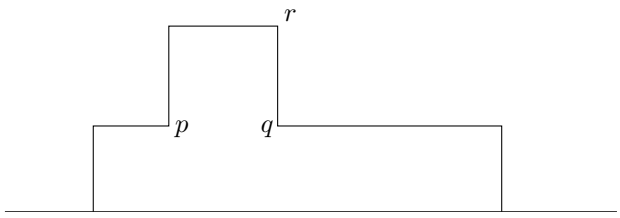
- set `F4` and `F5` snapping on, go into *line* mode, and reset axis system with `Shift-F1`,
- go near p, press `F1` and `F8` to set origin and to turn on angular snap.
- go near p', click left, and extend segment to s_2.
- go near q, press `F1` to reset origin, and draw second extension in the same way.

Example 2: We are given the polygon C, and we want to draw the bracket b, indicating its vertical extension.



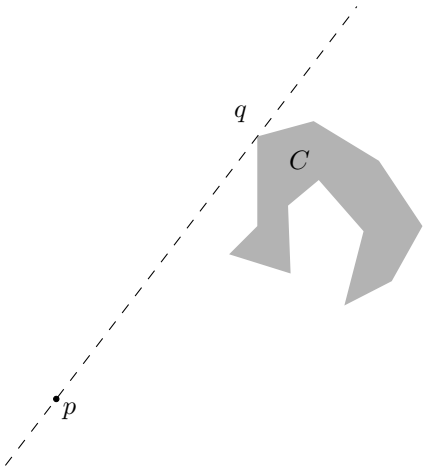
- set `F4` and `F9` snapping on, go into *line* mode, reset axis system, set snap angle to 90°.
- go near p, press `F1` and `F8` to set origin and angular snapping
- go to x, click left, extend segment to y, click left
- now we want to have z on a horizontal line through q: go near q, and press `F1` and `F8` to reset origin and to turn on angular snapping. Now both angular snapping modes are on, the snap lines intersect in z.
- click left at z, goto x and press `F1`, goto t and finish bracket.

Example 3: We want to draw the following "skyline". The only problem is to get q horizontally aligned with p.



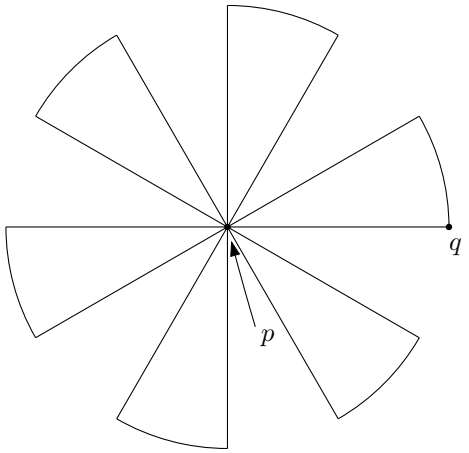
- draw the baseline using automatic angular snapping to get it horizontal.
- place p with boundary snapping, draw the rectilinear curve up to r with automatic angular snapping in 90° mode.
- now go to p and press `F1` and `F8`. The snap lines intersect in q. Click there, turn off angular snapping with `Shift-F1`, and finish curve. The last point is placed with boundary snapping.

Example 4: We want to draw a line through p, tangent to C in q.



- with vertex snapping on, put origin at p with `F1`
- go to q and press `F2`. This puts the base direction from p to q.
- set angular snapping with `F8` and draw line.

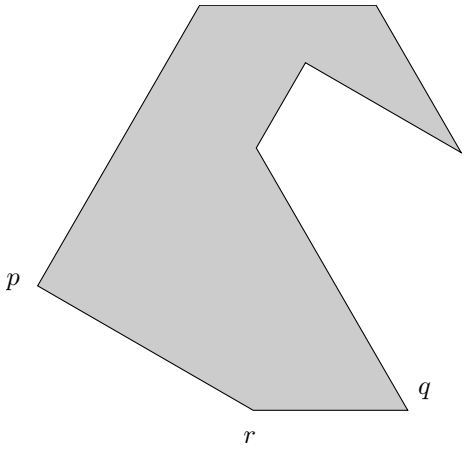
Example 5: We want to draw the following "windmill". The angle of the sector and between sectors should be 30°.



- set vertex snapping, snap angle to 30°, reset axis system with `Shift-F1`,
- with automatic angular snapping, draw a horizontal segment pq.
- go to p, place origin and turn on angular snapping with `F1` and `F8`,
- duplicate segment with `d`, go to q and pick up q for rotation (with Ctrl and the middle mouse button). Rotate until segment falls on the next snap line.
- turn off angular snapping with `F8`. Choose arc mode, variant "center & two points".

- go to p, click for center. Go to q, click for first endpoint of arc, and at r for the second endpoint. Select all, and group.
- turn angular snapping on again. Duplicate sector, and rotate by 60° using angular snapping.
- duplicate and rotate four more times.

Example 6: We want to draw a c-oriented polygon, where the angles between successive segments are multiples of 30°. The automatic angular snapping mode makes this pretty easy, but there is a little catch: How do we place the ultimate vertex such that it is at the same time properly aligned to the penultimate and to the very first vertex?



- set snap angle to 30°, and turn on automatic angular snapping.
- click first vertex p and draw the polygon up to the penultimate vertex q.
- it remains to place r such that it is in a legal position both with respect to q and p. The automatic angular snapping mode ensures the position with respect to q. We will use angular snapping from p to get it right: Go near p and turn on vertex snapping. Press `F1` to place the origin at p and `F8` to turn on angular snapping. Now it is trivial to place r.

7 Style sheets

The symbolic attributes appearing in an Ipe document are translated to absolute values for rendering by a *style sheet* that is attached to the document. Documents can have multiple "cascaded" style sheets, the sheets form a stack, and symbols are looked up from top to bottom. At the bottom of any style sheet cascade is always the *standard* style sheet, which is built into Ipe. When you create a new empty document, it automatically gets a copy of this standard style sheet.

The style sheet dialog (in the *Edit* menu under *Style sheets*) allows you to inspect the cascade of style sheets associated with your document, to add and remove style sheets, and to change their order. You can also save individual style sheets. You can, for instance, save the standard style sheet and use this as a basis for making your own style sheets.

The style sheets of your document also determine the choices you have in the Ipe user interface in "symbolic" mode. If you feel that Ipe does not offer you enough choice of colors, line widths, etc., you are ready to make your own style sheet! As an example for defining a style sheet with new colors, here is a style sheet that defines all the colors of the X11 color database.

Style sheets can also contain *templates*, such as background patterns, or logos to be displayed on each page. These are named Ipe objects that can be reused in documents. If your document's style sheets define a template named *background*, it will be added automatically to all newly created pages. You can create and use templates using the *template* ipelet. Here is a (silly) example of a

style sheet that defines a background template. This template is automatically added to all new pages:

```
<ipestyle name="background">
<template name="background">
<text pos="10 10" stroke="black" size="LARGE">
Background text
</text>
</template>
</ipestyle>
```

Style sheets can also define a piece of LaTeX-preamble for your document. When your text objects are processed by LaTeX, the preamble used consists of the pieces on the style sheet cascade, from bottom to top, followed by the preamble set for the document itself.

Here is a style sheet *presentation.xml* that can be used for presentations. It enlarges all standard sizes by a factor 3:

```
<ipestyle name="presentation">
<linewidth name="normal" value="1.2"/>
<linewidth name="heavier" value="2.4"/>
<linewidth name="fat" value="3.6"/>
<linewidth name="ultrafat" value="6"/>
<marksize name="normal" value="9"/>
<marksize name="large" value="15"/>
<marksize name="small" value="6"/>
<marksize name="tiny" value="3.3"/>
<arrowsize name="normal" value="21"/>
<arrowsize name="large" value="30"/>
<arrowsize name="small" value="15"/>
<arrowsize name="tiny" value="9"/>
<textstretch name="normal" value="3 3"/>
<textstretch name="large" value="3 3"/>
<textstretch name="Large" value="3 3"/>
<textstretch name="LARGE" value="3 3"/>
<textstretch name="huge" value="3 3"/>
<textstretch name="Huge" value="3 3"/>
<textstretch name="small" value="3 3"/>
<textstretch name="footnote" value="3 3"/>
<textstretch name="tiny" value="3 3"/>
<preamble>
\renewcommand\rmdefault{cmss}
\newenvironment{ITEM}{\begin{itemize}\item}{\end{itemize}}
</preamble>
<margins tl="72 72" br="72 72"/>
<shading type="axial" colora="1 0.9 0.5" colorb="1 0.8 0.75"
         coords="0 0 0 200" extend="0 1"/>
</ipestyle>
```

Note the use of the `<textstretch>` element to magnify text. The text size you choose from the Ipe user interface (*"large"*, for instance) is in fact used for *two* symbolic attributes, namely `textsize` (where *large* maps to `\large`) and `textstretch` (where it maps to no stretch in the standard style sheet). By setting the text stretch, you can magnify fonts (independently for height and width).

Also note the `<margins>` element—it determines the boundaries of the "standard" text area. The *Insert text box* function (in the *Edit* menu) uses these margins.

The LaTeX-preamble defined in the `<preamble>` element redefines the standard font shape to `cmss` (Computer Modern Sans Serif). Many people find sans-serif fonts easier to read on a screen.

Finally, note the `<shading>` element. You can use this to define a shading that will be applied to every page in your PDF/Postscript output, before anything else is drawn on the page. The shading is invisible in the Ipe user interface, so use this it with care.

8 Page views

When making a PDF presentation with Acrobat Reader, one would often like to present a page incrementally. For instance, I would first like to show a polygon, then add its triangulation, and finally color the vertices. *Page views* (or simply *views* in the following) make it possible to do this nicely.

An Ipe document consists of several pages, each of which can consist of an arbitrary number of views. When saving as PDF or Postscript, each view generates a separate PDF/Postscript page (if you only look at the result in, say, Acrobat reader, you cannot tell whether two pages are actually two views of the same Ipe page or two different Ipe pages).

An Ipe page consists of a number of objects, a number of layers, and a number of views. Each object belongs to exactly one layer. A layer can be shown by any number of views—a view is really just a list of layers to be presented. In addition, a view keeps a record of the current active layer—this makes it easy to move around your views and edit them. Finally, views can specify a *transition style*, a graphic effect to be used by the PDF viewer when proceeding to the following PDF page.

To return to our polygon triangulation example, let's create an empty page. We draw a polygon into the default layer "alpha." Now use the *New layer, new view* function (in the *Views* menu), and draw the triangulation into the new layer "beta." Note that the function not only created a new layer, but also a second view showing both "alpha" and "beta". Try moving back and forth between the two views (using the PageUp and PageDown keys, or the little buttons on the *View* counter). You'll see changes in the layer list on the left: in view 1, layer "alpha" is selected and active, in view 2, both layers are selected and "beta" is active. Create a third layer and view, and mark the vertices. In the *Document properties* (in the *Edit* menu), turn *cropbox* off and *fullscreen* on, and save in PDF format. Voila, you have a lovely little presentation.

In presentations, one often has slides with mostly text. The textbox object is convenient for this, as one doesn't need to use the mouse to create it. To create a slide where several text items appear one by one, one only needs to press F10 to create a textbox, then Shift+Ctrl+I to make a new view, F10 again for the next textbox, and so on. Finally, one moves the textboxes vertically for the most pleasing effect (*Shift+Ctrl+Middle Mouse* does a constrained vertical move, or *Shift+Left Mouse* in *Move* mode).

9 Writing ipelets

An ipelet is a dynamically loaded library (DLL), that you place on Ipe's ipelet search path. Ipe loads all DLLs it can find on that path during start-up. The DLL has to be written in C++, and must export a function `NewIpelet` that creates an object derived from the class `Ipelet` (defined in *ipelet.h*). Here is minimal ipelet implementation:

```
#include "ipelet.h"

class MyIpelet : public Ipelet {
public:
```

```
  virtual int IpelibVersion() const { return IPELIB_VERSION; }
  virtual const char *Label() const { return "My label"; }
  virtual void Run(IpePage *page, IpeletHelper *helper);
};

void MyIpelet::Run(int function, IpePage *page, IpeletHelper *helper)
{
  // this is where you do all the work
}

IPELET_DECLARE Ipelet *NewIpelet()
{
  return new MyIpelet;
}
```

When the ipelet is executed, Ipe hands it a pointer to the current page of the document. The ipelet can examine the selected objects, and modify the page in any way it wishes. It can also request services from the Ipe application through the `IpeHelper` object, for instance to display a message in the status bar, to pop up message boxes, to obtain input from the user, etc. Through the `IpeHelper`, it is also possible to access the complete document (for instance to write an ipelet that allows the user to reorganize the pages of the document), or to access some Ipe settings.

The Ipelib documentation in HTML-format is available as part of the Ipe distribution (check the on-line manual). You may want to have a look at the standard ipelets. *Kgon*, for instance, is a minimal ipelet that you can use as the basis for your own development. *Goodies* is an example of an ipelet that contains more than one function—it also needs to implement the member functions `NumFunctions` and `SubLabel`. Note that it is possible for ipelets to define keyboard shortcuts (the *Align* ipelet does that, for instance), but in general it is not a good idea to do that for ipelets you plan to make available for others.

Compiling ipelets on Windows The ipelet must be compiled as a DLL and must be linked with the Ipe library "libipe.lib". C++ mandates that it must be compiled with the same compiler that was used to compile Ipe. If you use the binary Ipe distribution for Windows, that means you have to use the free Borland C++ compiler. (Its command-line version is available as a download after registering on the Borland webpage, the same compiler is also in Borland C++ builder.) The Ipe Windows distribution contains the necessary header files and the library to compile ipelets, as well as the source of the "kgon" and "goodies" ipelets as examples. You can compile the "kgon" example as follows:

```
bcc32 -WD -tWR -DWIN32 -Iinclude kgon.cpp lib/libipe.lib
```

Place the resulting *kgon.dll* in the *ipelets* subdirectory, and restart Ipe.

Compiling ipelets on Unix The ipelet must be compiled as a shared library and must be linked with the Ipe library "libipe.so". C++ mandates that it must be compiled with the same compiler that was used to compile Ipe. Have a look at the ipelet sources in the Ipe source distribution, and their project files for details on compiling them.

10 Troubleshooting the LaTeX-conversion

Ipe converts text objects from their Latex source representation to a representation that can be rendered and included in Postscript and PDF by creating a Latex source file and running Pdflatex. This happens in a dedicated directory, which Ipe creates the first time it is used. The Latex source

and output files are left in that directory and not deleted even when you close Ipe, to make it easy to solve problems with the Latex conversion process.

You can determine the directory used by Ipe by pressing the *Search paths* button in the *Preferences dialog* (from the *Help* menu). If you'd prefer to use a different directory, set the environment variable `IPELATEXDIR` before starting Ipe.

If Ipe fails to translate your text objects, and you cannot find the problem by looking at the log file displayed by Ipe (or Ipe doesn't even display the log file), you can terminate Ipe, go to the conversion directory, and run Pdflatex manually:

```
pdflatex text.tex
```

11 Using Truetype fonts

To make PDF presentations that are as "fancy" as the PowerPoint presentations of competing speakers one needs to use fancy fonts. It's not hard to find nice fonts, but they are mostly in Truetype (TTF) format. This section explains how to use TTF fonts in Ipe.

Ipe relies on Pdflatex to translate the text source representation into a string of PDF operators and font subsets, that can then be used to generate Postscript, PDF, and to display the text on the screen. Ipe can therefore use any font that Pdflatex can handle, and to use a TTF font we just have to add it to Pdflatex's font reportoire.

I've made a webpage[8] explaining the steps necessary to add a TTF font to Pdftex's font repertoire, using the *lhandw.ttf* font as an example. Let's assume that you have performed these steps, and that you can access the font when running Pdflatex normally (not from Ipe).

We are then ready to try the font from within Ipe. Let's first assume you only want to use the new font in a few places in your Ipe document. You should define a command analogous to `\textrm` to switch to the new font. Open the *Document properties* dialog in the *Edit* menu, and add this line to the *Latex preamble*:

```
\DeclareTextFontCommand{\textlh}
{\fontencoding{T1}\fontfamily{lhandw}\selectfont}
```

You can now use `\textlh` inside Ipe text objects to typeset in Lucida-Handwriting.

Finally, let's make a multi-page presentation typeset wholly using Lucida-Handwriting. This declaration in the Latex preamble will change the document fonts:

```
\renewcommand{\encodingdefault}{T1}
\renewcommand{\rmdefault}{lhandw}
\renewcommand{\sfdefault}{phv}
\renewcommand{\ttdefault}{pcr}
```

Note that this switches all text fonts to TTF or Postscript fonts. This is necessary, as we use the `T1` encoding (an 8-bit encoding) for Lucida-Handwriting. Keeping Computer-Modern as the font for `\textsf` or `\texttt` would cause LaTeX to load the `T1` version of Computer-Modern. These are bitmapped "Type3" fonts, which Ipe cannot handle.

12 Unicode text

If you make figures containing text objects in languages other than English, you will need to enter accented characters, or characters from other scripts such as Greek, Hangul, Kana, or Chinese characters. Of course you can still use the LaTeX syntax `K\"onig` to enter the German word

[8] `http://ipe.compgeom.org/pdftex.html`

"König", but for larger runs of text it's more convenient to exploit the fact that the Ipe user interface (thanks to the Qt toolkit) is Unicode-aware, and let's you enter text in any script supported by your system.

However, the Unicode text also has to be processed by Pdflatex. The easiest solution, sufficient for German, French, and other languages for which support is already in a standard LaTeX-setup, is to add the line

```
\usepackage{ucs}
```

in your *Latex preamble* (set in the *Document properties* dialog, available on the *Edit* menu). You will need to install the *ucs package* for Latex by Dominique Unruh[9], if it not yet on your system.

For more complicated needs, you'll need to read further. When Ipe writes the Pdflatex source file, it replaces all Unicode characters by a Latex macro, such as `\unichar{44032}` for the Korean syllable "ga". The `ucs` package implements `\unichar` for many scripts, including Chinese, Japanese, and Korean. See the `ucs` documentation to set this up and for the options you need to use.

If you have Truetype (TTF) fonts that include the scripts you wish to use in your Ipe document, there is an alternative solution. You can set up Pdflatex to directly map the `\unichar` macro to the right glyph in this font.

Follow the instructions on my webpage[10] to declare a Truetype font to be used for Unicode characters in the document. We first test it "manually", by running Pdflatex on this test file:

```
% File 'unitest.tex'
\documentclass{article}
\usepackage{ttfucs}
\DeclareTruetypeFont{cyberb}
\begin{document}
Here is a character from Cyberbit: \unichar{44032}.
\end{document}
```

Assuming this works fine, we can now try to use the font from Ipe. All you need to do is to add the lines

```
\usepackage{ttfucs}
\DeclareTruetypeFont{cyberb}
```

in the Latex preamble. Unicode characters entered from the Ipe user interface should now be displayed correctly.

You can use more than one TTF font, and add several `\DeclareTruetypeFont` declarations to the Latex preamble of your Ipe document. The last package determines the standard font for Unicode characters. To select a different Unicode font, use the `\TruetypeFont` command defined in the `ttfucs` package.

You can also mix this strategy with using the `ucs` package—the command `\ucsfamily` will switch to using `ucs`, until you switch back to using a Truetype font by saying `\TruetypeFont`.

13 Customizing Ipe

A few features of the Ipe user interface can be changed in the *Preferences* dialog (in the *Help* menu).

Ipe contains support for localizing the user interface into another language. Contact me if you wish to do such a localization project.

[9] http://www.unruh.de/DniQ/latex/unicode/

[10] `http://ipe.compgeom.org/pdftex.html`

The keyboard shortcuts used by Ipe can be customized to your personal taste. Save the current configuration into, say, a file *ipekeys.qm* using the function *Save keys* from the *Help* menu. Translate *ipekeys.qm* into XML format using `qm2ts` (from Qt 3), edit the resulting *ipekeys.ts* file (you can discard all the definitions you do not wish to change), and compile back into *.qm* format using `lrelease`. Install the resulting file as *.ipekeys.qm* in your home directory, and restart Ipe.

Here is an example of an edited *ipekeys.ts* by Kostas Oikonomou:

```
<!DOCTYPE TS><TS>
<context>
    <name>Key</name>
    <message>
        <source>Ctrl+V</source>
        <comment>Edit|Paste</comment>
        <translation>F18</translation>
    </message>
    <message>
        <source>Ctrl+X</source>
        <comment>Edit|Cut</comment>
        <translation>F20</translation>
    </message>
    <comment>
    If this is not here, although it's unchanged from the default setting,
    File|New Window will show as Ctrl+X,Ctrl+C, because of the translation for
    File|Exit, which also has source "none".  Tricky!  Same goes for other
    similar situations.
    </comment>
    <message>
        <source>none</source>
        <comment>File|New Window</comment>
        <translation>none</translation>
    </message>
    <message>
        <source>none</source>
        <comment>File|Save as bitmap</comment>
        <translation>none</translation>
    </message>
    <message>
        <source>none</source>
        <comment>File|Exit</comment>
        <translation>Ctrl+X,Ctrl+C</translation>
    </message>
</context>
</TS>
```

If you are using Qt version 3.1.0 or higher (see *Help menu*, *About Qt* if you don't know), you can use shortcuts consisting of more than one key, such as *Ctrl+X,Ctrl+C* as in the example above (use commas to separate the key presses—up to four are supported).

14 The Ipe file format

Ipe can store documents in several possible formats. Among them are standard PDF and Postscript, which can be read by any application capable of opening such files, such as Acro-

bat Reader, Xpdf, or Ghostview. (Ipe embeds its own information inside PDF and Postscript files. The way this is done is not documented here.)

There is one other Ipe file format, which is a pure XML implementation. Files stored in this format can be parsed with any XML-aware application, and you can create XML files for Ipe from your own applications. The tags understood by Ipe are described informally in this section. A formal DTD will be provided in due course (volunteers?).

Tags in the XML file can carry attributes other than the ones documented here. Ipe ignores all attributes it does't understand, and they will be lost if the document is saved again from Ipe.

14.1 The elements of an Ipe XML file

A file can optionally start with the `<?xml>` tag, which is simply ignored by Ipe.

The only element in the file must be `<ipe>`. The optional attribute `media` indicates the physical boundaries of the "paper" containing the page. The value is a sequence of four integers, separated by spaces, in the order min-x, min-y, max-x, max-y, in postscript points (1/72 inch). If the attribute is not present, A4 size is assumed. The optional attribute `version` indicates the earliest Ipelib version that can interpret the document. This allows for extensions to the Ipe file format, but not withdrawal of features. Ipe will refuse to load documents that require a version larger than its own. The optional attribute `creator` indicates the program that created the file, it is not interpreted by Ipe at all.

The `<ipe>` element contains, in this order, an optional `<info>` element, an optional `<preamble>` element, an optional series of `<ipestyle>` elements, an optional series of `<bitmap>` elements, and a series of `<page>` elements.

The `<info>` element takes the optional attributes `title`, `author`, `subject`, `keywords`, `pagemode`, `bbox`, `created`, and `modified`. The only value for `pagemode` currently understood by Ipe is `fullscreen`. If the value of `bbox` is `cropbox`, Ipe will create a `CropBox` attribute when saving to PDF. The value of `created` and `modified` should be a date in PDF format, that is a string like "D:20030127204100".

The contents of the `preamble` element is used as the LaTeX preamble when running LaTeX to process the text objects in the document. It should *not* contain a `\documentclass` command, but can contain `\usepackage` commands and macro definitions.

The contents of the `<ipestyle>` element is parsed as an Ipe style sheet, see below. Several style sheets form a stack or "cascade", with the last `<ipestyle>` element becoming the top-level style sheet. When symbolic names are looked up, the style sheets are checked from top to bottom. Ipe always appends the built-in standard style sheet at the bottom of the stack.

Each `<bitmap>` element defines a bitmap to be used by `<image>` objects. It takes the required attributes `id` (the value must be an integer that will define the bitmap throughout the Ipe document), `width` and `height` (integers, specifying the dimensions of the bitmap in pixels), `ColorSpace` (possible values are "DeviceGray", "DeviceRGB", and "DeviceCMYK"), `BitsPerComponent` (only 8 is currently allowed!), and `length` (indicating the number of bytes of image data). The optional attribute `Filter` can take one of the values "FlateDecode" or "DCTDecode" to indicate a compressed image (the latter is used for JPEG images). The `length` attribute can be omitted if there is no filter (Ipe can then deduce it from the other attributes).

The contents of the `<bitmap>` element is the image data in hexadecimal format. White space between bytes is ignored. If no filter is specified, pixels are stored row by row, with rows padded to a full byte boundary.

Note that images with color maps are not supported, and such support is not planned. (The *Insert image* ipelet does allow you to insert images with color maps, but they are stored as 24-bit images. Since the data is compressed, this does not seriously increase the image data size.)

14.2 The <page> element

The <page> tag has one optional attribute gridsize. The gridsize indicates the default grid size to be used on this page—Ipe remembers this as it is often advantageous to have the same grid available that was used when making the page.

The contents of the <page> element consists, in this order, of possibly empty sequences of <layer> elements, <view> elements, and Ipe object elements.

The <layer> tag takes a required attribute name, and two optional attributes visible and edit. The name has to be unique within the page. The value of visible should be yes, no, or dim, and indicates how the layer is displayed on the screen (*not* in the PDF output—this is determined by the <view> elements), the value of edit should be yes or no and indicates whether the user can select and modify the contents of the layer in the Ipe user interface (of course the user can always modify the setting of the attribute). The layer element must be empty.

If a page contains no layer element, Ipe automatically adds a default layer named "alpha", visible and editable.

The <view> tag takes a required attribute layers, and three optional attributes, duration, transition, and effect. The value of layers must be a sequence of layer names defined in this page, separated by white space. The value of duration and transition must be a real number, the value of effect must be an integer between 0 and 16 (see IpeView::TEffect for the exact meaning of those). The parameters translated directly into the page transition effect in PDF.

It is okay for a page not to contain any <view> element. Such a page will be saved to PDF including all its layers, with no special effect.

The remaining elements of the <page> are Ipe objects.

14.3 Ipe object elements

In the following, we explain the Ipe object elements currently understood by Ipe.

A "top-level" Ipe object, that is an object directly inside a <page> element, can take the optional layer attribute. This attribute indicates into which layer the object goes. If it is missing, the object goes into the same layer as the preceding object. If the first object has no layer attribute, it goes into the layer defined first in the page, or the default "alpha" layer.

Any Ipe object can take the optional attributes stroke and matrix. The value of stroke is a color—either a symbolic name defined in one of the style sheets of the document, one of the predefined names "void", "black", or "white", a single real number between 0 (black) and 1 (white) indicating a gray level, or three real numbers in the range $[0, 1]$ indicating the red, green, and blue component (in this order), separated by white space.

The value of matrix is a sequence of six real numbers, separated by white space, indicating a transformation matrix for all coordinates inside the element (including embedded elements if this is a <group> element). A missing matrix attribute is interpreted as the identity matrix.

14.3.1 The <mark> element

The <mark> element defines a mark object. It takes the required attributes size (a real number or symbolic name), type (an integer, see IpeMark), and <pos> (two real numbers, separated by white space).

14.3.2 The <ref> element

The <ref> element refers to an Ipe object defined in the style sheet. It has one required attribute name, which must be a name of a <template> defined in the style sheet. The object will be reused at the position defined in the template, unless the matrix is set.

14.3.3 The <image> element

The <image> element defines a bitmap object. The tag takes the required attributes bitmap (the value is an integer referring to a bitmap defined in a <bitmap> element in the document), and rect (four real coordinates separated by white space, in the order x_1, y_1, x_2, y_2, indicating two opposite corners of the image in Ipe coordinates), The optional matrix attribute can be used to transform the image into a non-rectangular shape.

It is also possible to embed a bitmap *directly*, without first creating a <bitmap> element. In that case, the bitmap attribute must be omitted, and instead the <image> element must carry all the attributes of the <bitmap> element, with the exception of id. The element contents is then the bitmap data, as described for <bitmap>.

14.3.4 The <text> element

The <text> element takes the required attributes size (the font size—either a symbolic name defined in a style sheet, or a real number) and pos (two real numbers separated by white space, defining the position of the text on the paper).

The required attribute type, with the possible values *label*, *minipage*, *textbox*, and *title* determines the type of object, and the attributes width, height, and depth give its dimensions. Note that these dimensions are recomputed by Ipe when running LaTeX, with the exception of width for minipage objects whose width is fixed. If the dimensions are missing, Ipe uses some default values until LaTeX has been run. (Obviously, width must not be missing for a minipage object.) If type is missing, the object is a minipage if width is present, otherwise a label.

The optional attributes valign (with values *top* (default for a minipage object), *bottom* (default for a label object), *center*, and *baseline*) and halign (with values *left*, *right*, and *center*, with *left* the default) determine the position of the reference point with respect to the text box.

The optional attribute transformable (possible values are *yes* and *no*, the latter is the default) determines whether the text object can be transformed, that is, stretched and rotated. Moving, of course, is always allowed.

The contents of the element must be a legal LaTeX fragment that can be interpreted by LaTeX inside \hbox, possibly using the macros or packages defined in the preamble.

14.3.5 The <path> element

The <path> element is the most complex element, and represents any filled and/or stroked PDF path, that is, any sequence of Postscript moveto/lineto/curveto/closepath operations, followed by a single fill/stroke operation. In particular, paths consisting of more than one closed loop are allowed, and so is any mix of straight segments and Bezier curves in the paths.

The <path> element takes the following optional attributes: stroke, fill, dash, pen, cap, join, fillrule, matrix, arrow, backarrow.

The value of dash is either the predefined name "solid", a symbolic name defined in a style sheet, or a dash pattern in PDF format, such as "[3 1] 0" for "three pixels on, one off, starting with the first pixel".

The value of pen is the line width, either symbolic (defined in a style sheet), or as a single real number.

The values of cap, join are the *line cap* and *line join* settings of PDF, as integers.

The value of the fillrule attribute selects one of two algorithms for determining whether a point lies inside a filled object. Possible values are wind and eofill (the latter is the default if the attribute is missing).

The value of the <arrow> and <backarrow> attributes is the size of an arrow, either a symbolic name defined in a style sheet, or a real number. If the attribute is missing, no arrow is drawn.

The contents of the <path> element describes a path using a series of "path construction operators" with arguments. This generalizes the PDF path construction syntax.

Each operator follows its arguments. The operators are

- `m` (moveto) (1 point argument): begin new subpath,
- `l` (lineto) (1 point argument): add straight segment to subpath,
- `c` (curveto) (3 point arguments): add a cubic Bézier curve,
- `e` (ellipse) (1 matrix argument): add a closed subpath consisting of an ellipse, the ellipse is the image of the unit circle under the transformation described by the matrix,
- `a` (arcto) (1 matrix argument, 1 point argument): add an elliptic arc, on the ellipse describe by the matrix, from current position to given point,
- `s` (spline) (n point arguments): add a uniform cubic B-spline with $n + 5$ control points (current position and nth point have multiplicity 3),
- `u` (closed spline) (n point arguments): add a closed subpath consisting of a closed uniform B-spline with n control points.
- `h` (closepath) (no arguments): close the current subpath. No more segments can be added to this subpath, so the next operator (if there is one) must start a new subpath.

14.3.6 The `<group>` element

The `<group>` element allows to group objects together, so that they appear as one in the user interface. The contents of the element consists of a series of Ipe object elements.

The `<group>` element takes all the possible attributes of the `<path>` element (with the exception of the two arrow attributes), as well as `textsize`, `marksize`, and `markshape`.

If one of these attributes has been set, its value overrides the corresponding attribute of the elements inside the group. In other words, the attribute of an object is meaningful only if none of its parents in the tree structure formed by the grouped objects has this attribute set.

14.4 The Ipe style sheet format

Style sheets can either be embedded in an Ipe document, or reside in a separate file. Separate files can start with the optional `<?xml>` tag.

The style sheet itself consists of the single element `<ipestyle>`. It takes the optional attribute `name`, which only serves to identify the style sheet informally.

The contents of the style sheet element can consist of the following elements: `template`, `color`, `dashstyle`, `linewidth`, `textsize`, `marksize`, `arrowsize`, `grid`, `angle`, `media`, `preamble`, `textstretch`, `margins`, and `shading`.

The `template` element takes a required attribute `name`, which identifies the template and must be unique in the style sheet. Its contents is a single Ipe object.

The contents of the `preamble` element defines a (piece of) LaTeX-preamble.

The `margins` element takes the two required attributes `tl` and `br`, for the top-left and bottom-right margins of the standard text area on the page.

The `shading` element sets the background shading of a PDF page. Its required attributes are `type` (with the possible values `axial` and `radial`), `colora` and `colorb` (the two extreme colors that are being interpolated), `extend` (two integer flags, indicating whether to extend the shading to the full page), and `coords` (for axial shading, the coordinates of the endpoints of the axis, for radial shading, the center and radius of both circles). The shading will be applied to every PDF page, before anything else is drawn on it. It is completely invisible in the Ipe user interface, so use this it with care.

All other elements have empty contents and two required attributes, `name` and `value`. `name` defines a symbolic name, `value` an absolute value. The symbolic name must start with a letter 'a' to 'z' or 'A' to 'Z'. The value for the `color` and `dashstyle` elements must be a legal absolute value for the Ipe object attributes of the same name. The value for `linewidth`, `textsize`, `marksize`, `arrowsize`, `grid`, and `angle` must be single real number. The value for `media` must be two integers

(width and height in Postscript points, i.e. 1/72 inch), separated by white space. The value for `textstretch` must be two real numbers (stretch in horizontal and vertical direction).

Note that the symbolic names for `grid`, `angle`, and `media` cannot actually be used by objects in the document—they are only used to fill the grid size, angular snap angle, and page size selectors in the user interface with values.

15 If you have used Ipe 5 before...

...you may be shocked about some of the new developments in Ipe 6. In particular, the clever Ipe file format that gave Ipe its name has disappeared. (You do remember, of course, that Ipe used to stand for "Integrated Picture Environment", because Ipe files were at the same time legal Latex source code and Postscript files.)

Ipe 6 simply writes Postscript or PDF files. It still maintains text objects as Latex source, and you edit text objects by editing Latex source, but before creating an Ipe file, Ipe runs Pdflatex and stores the resulting PDF representation in the file. This is a much better solution, as Ipe figures now require no special handling. You no longer have to add page-long explanations when you send Ipe files to a co-author or publisher. The new approach was made possible by two important developments that have happened since 1993, when I first wrote Ipe. First, all relevant Latex fonts are now available as scalable Type1 fonts, and so it is possible to embed Latex text and formulas in figures that may still need to be scaled later. Second, Hàn Thê Thàn's pdfTeX, a version of TEX that produces PDF instead of DVI output, makes it possible for Ipe to extract the processed text objects from a Latex run and to include them in its own output.

The Ghostscript window is also no longer necessary, as Ipe can now render text objects the way they will look on paper.

Perhaps you are worried now that you cannot continue to use your megabytes worth of existing Ipe figures with Ipe 6. Fear not. Ipe 6 comes with a tool "ipe5toxml" that converts figures created with Ipe 5 and earlier to Ipe 6's XML format. In fact, you can just select an old Ipe file from the Ipe user interface, and Ipe will run the tool for you automatically. (There was no question that Ipe 6 would have to be able to communicate in XML, of course.)

Other than the file format, there aren't really that many changes to Ipe's functionality. I've added style sheets and layers, because René van Oostrum says that no self-respecting drawing program can do without. I've added page views, which allow you to incrementally build up a page in a PDF presentation, because Christian Knauer wants to make PowerPoint-like presentations in Ipe. And obviously it's now possible to use Korean, Chinese, and Japanese in figures.

I've also revised the interface to ipelets (which used to be called "Iums" in the good old times when people still thought that "applets" were small apples)—it is now based on dynamically loaded libraries (a technology that was still somewhat poorly understood ten years ago, at least by me). Ipelets are now much more powerful than they used to be, but buggy ipelets can also crash Ipe much more easily than in the past. Well, you can always do the real work in a separate program, I guess.

Finally, Ipe 6 has been rewritten from scratch in clean, beautiful, object-oriented C++. Quite unlike the previous version. Oh, and there's a Windows version of Ipe now. Who would have thought that ten years ago!

16 History and acknowledgments

Many people have contributed with ideas, inspiration, and moral support over the years that I wrote and used Ipe. Based on my experiences with Idraw, XFig, and Jean-Pierre Merlet's JPDraw, I wrote the first version of Ipe at Utrecht University in the summer of 1993. It used IRIS-GL and Mark Overmars' FORMS library, and run on SGI workstations only. Due to popular demand, I

finally gave in a year later, and spent two weeks in the summer of 1994 to teach myself Motif and to rewrite Ipe to run under the X window system. Unfortunately, two weeks were really not enough, and the 1994 X-version of Ipe has always been a hack. I didn't have time to port the code that displayed bitmaps on the screen, it crashed on both monochrome and truecolor (24-bit) displays, and was in general quite unmaintainable.

These two first versions of Ipe were supported by the Netherlands' Organization for Scientific Research (NWO), and I would never have started working on it without Geert-Jan Giezeman's PLAGEO library. For testing, support, and inspiration in that original period, I'm grateful to Mark de Berg, Maarten Pennings, Jules Vleugels, Vincenzo Ferrucci, and Anil Rao. Many students of the department at Utrecht University served as alpha-testers (although I would still like to find out who coined the phrase "the cute little core-dumper").

I gave a presentation about Ipe at the Dagstuhl Workshop on Computational Geometry in 1995, and made a poster presentation at the ACM Symposium on Computational Geometry in Vancouver in the same year. Both served to create a small but faithful community of Ipe addicts within the Computational Geometry community.

Ipe proved itself invaluable to me over the years, especially when we used it to make all the illustrations in our book "Computational Geometry: Theory and Applications" (Springer 1997, with Mark de Berg, Marc van Kreveld, and Mark Overmars). Nevertheless, the problems were undeniable: It was hard to compile Ipe on other C++ compilers and it only worked on 8-bit displays. It is only due to the efforts of Ipe fans such as Tycho Strijk, Robert-Paul Berretty, Alexander Wolff, and Sariel Har-Peled that the 1994 version of Ipe continued to be used until 2003.

I was teaching myself C++ while writing the first version of Ipe, and it shows—Ipe 5 is full of elementary object-oriented design mistakes. When teaching C++ to second-year students at Postech in 1996 I started to think about a clean rewrite of Ipe. My first notes on such a rewrite stem from evenings spent at a hotel in Machida, close to IBM Tokyo in July 1996 (the idea at that time was to embed Ipe into Emacs!). It proved impossible, though, to do a full rewrite next to teaching and research, and nothing really happened until the Dagstuhl Workshop on Computational Geometry in 2001, where Christian Knauer explained to me how he uses Pdflatex to create presentations. I realized that PDF was ideally suited for a new version of Ipe. By parsing the PDF output of Pdflatex, Ipe can obtain a PDF representation of text objects, including all the necessary fonts. The advent of scalable Latex fonts means that Ipe can then create a scalable PDF or Postscript figure including the processed text information. Directly after the workshop I implemented a proof-of-concept: I defined the Ipe XML format, wrote "ipe5toxml" (reusing my old Ipe parsing code) and a program that runs Pdflatex, parses its PDF output, extracts text objects and font data, and creates a PDF file for the whole Ipe figure. This was only possible, of course, due to the existence of Hàn Thê Thàn's Pdflatex and of an open-source PDF parser in the form of Derek Noonburg's Xpdf.

Now all that remained was to rewrite the user interface. Since there is plenty of demand for both Windows and Linux versions, I wanted Ipe to run on both operating systems, and had to use a toolkit supporting both. I had experimented with V, FLTK, and Qt over the years, and had used V and Qt in teaching. On the other hand, FLTK is small and compact, free software on both Windows and Unix, and smells pleasantly like the original FORMS library from which it is derived—the same that I had used in Ipe 2 in 1993. I actually started writing an FLTK main window and canvas for Ipe, until Sariel Har-Peled stated with authority that Qt was the only sensible choice. Fortunately, I had been able to purchase a Windows developers license for Qt using funding from the Hong Kong Research Grants Council. What a pleasure to work with Qt, compared to my Motif experiences from 1994!

Finally, Mark de Berg and the TU Eindhoven made it possible for me to take some time off from teaching and research. The final design changes were made during the Second McGill-INRIA Workshop on Computational Geometry in Computer Graphics at McGill's Bellairs Research Institute, and much inspiration is due to the atmosphere at the workshop and the magnificient cooking by Gwen, Bellair's chef.

For code actually used in Ipe, I wish to thank

- Hàn Thê Thàn for Pdflatex,
- Derek Noonburg for Xpdf,
- Werner Lemberg and the rest of the Freetype team for Freetype 2,
- Trolltech for Qt.

Looking at the history described briefly above, it is clear that the Dagstuhl series of workshops has always been a major influence in the existence of Ipe. It is for that reason that I released Ipe 6.0 formally at the Dagstuhl Workshop on Computational Geometry in March 2003.

17 Copyright

[11] at `http://www.gnu.org/copyleft/gpl.html`

Contents