

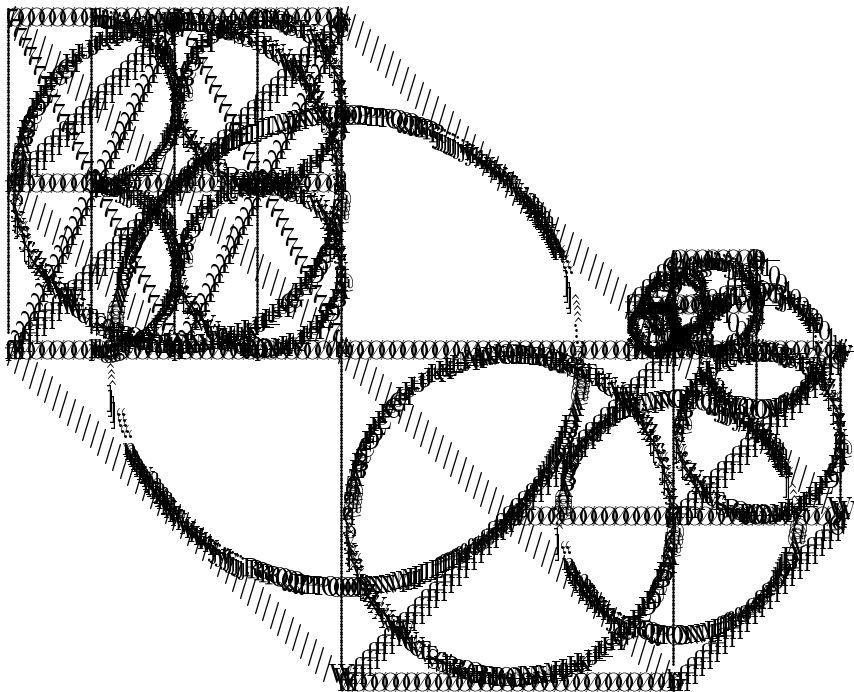
T_EXtyl: a line-drawing interface for T_EX

John S. Renner

14 March 1987 ¹

¹Copyright ©1987 John S. Renner All rights reserved.

*Geometry can produce legible letters,
but art alone makes them beautiful.*



*Art begins where geometry ends,
and imparts to letters a character
transcending mere measurement.*

Paul Standard

Contents

1	The Basics	1
2	Advanced Features	9
2.1	Transforms	9
2.2	Figures	10
2.3	Others	14
3	User-Level Details	17
4	When Things Go Wrong	23
5	Design Details	27
5.1	Vectors	27
5.2	Splines	30
6	The Implementation	33
6.1	Overview	33
6.2	Primitives	34
6.3	Procedural Handling	38
6.4	Figures	39
6.5	The Fonts	42
6.5.1	Vector Setting	43
6.5.2	Beam Setting	44
6.6	Buffering	45
6.7	Odds 'n Sods	46
7	Future Extensions	50

A	T_EXtyl Summary	53
B	Font Example	57
C	Macros and Extended Examples	59

Introduction

TEXtyl is a prototype post-processing system to be used in conjunction with \TeX [9], the typesetting program by Donald Knuth. The purpose of *TEXtyl* is to “draw” lines and curves in a device-independent manner. *TEXtyl* is a post-processor that interprets certain commands in a DVI (device-independent) file that is the output of \TeX and contains the actual typesetting commands. *TEXtyl* will produce as its output another DVI file that contains the commands to typeset the lines and curves on a printer or output device.

In the current version of *TEXtyl* (version 1.2 of March 1987), the simple line drawing capabilities include several general types of straight lines, arcs, and spline curves. Advanced features of *TEXtyl* are for *MusiCopy*, the music-typesetting project at the Ohio State University[7], namely typesetting ties, slurs, and beams.

The name for *TEXtyl* comes from three ideas: (1) a proper computer graphics term for rendering images is called “tiling,” (2) the mis-pronunciation of the name sounds like “textile,” which evokes an image of lines and bargello effects from weaving, and (3) this process of “tyl”-ing comes after \TeX .

The main design goals of *TEXtyl* were (1) the minimization of data required to typeset graphics on printers capable of setting only characters at arbitrary positions on a page, (2) the adherence to the DVI format of \TeX output for specifying the way to typeset a document in a device-independent manner, (3) a simple, portable interface that would not require a change to the functionality or design of \TeX , and (4) a user interface that is easy to use by people and programs alike.

TEXtyl was not intended to mimic nor replace sexy programs like *Ideal*[15], which know about constraints and alignment niceties. Rather, *TEXtyl* is designed to take care of medium-level details for \TeX for use on printing devices

not capable of directly accessing and loading full bitmaps produced by such systems as *PostScript*[1]. It is clear that DVI-format is a subset of *PostScript*, but a large community of users are not able to make use of such a super-set, and *TEXtyl* is intended to provide line-drawing capabilities for them.

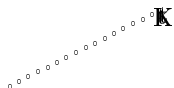
Probably the best way to read this manual is to read the first four (4) chapters, and skip right to the chapter entitled *Future Extensions*, for a good introduction to *TEXtyl*. The other chapters deal with details of the innards of *TEXtyl*, and describe the design in successively finer detail.

I would like to thank John Gourlay, and Clayton Elwell for their support, comments, reminders, and good ideas during this three-year labor, and the CGRG for not hassling me about this.

Chapter 1

The Basics

So, let's draw. I will assume that the reader has a reasonably good working knowledge of how to use $\text{T}_{\text{E}}\text{X}$ and/or $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$, and sort of understands Knuth's concept of "boxes," and their attributes of height, width, and current position. We need these concepts because our direct interface is with $\text{T}_{\text{E}}\text{X}$, in the form of a `.tex` text-file. To typeset the line-drawings, we ask $\text{T}_{\text{E}}\text{X}$ to place our drawing at the current position on the page. Throughout this manual, I will provide some examples of drawings to illustrate ideas. These figures are thumb-nail size, but should be sufficient to get the idea across without using up lots of space on a page. Let's look at a simple example: After typing `\input{textyl}` near the beginning of your file, but after any required preamble, we can obtain a trivial drawing like:



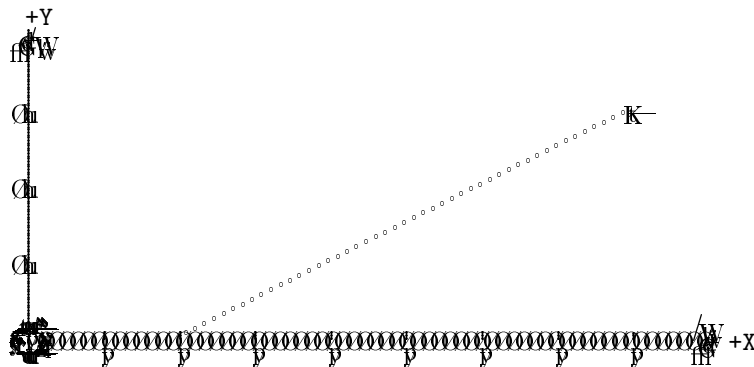
Simple, but we achieved this line by typing

```
...a trivial drawing like:\par
\noindent\hskip 2in\beginlyl{2cm}[1in]
\special{tyl line 4 20, 0; 80,30}
\endtyl\par
```

The `\beginlyl{2cm}[2in]` and `\endtyl` commands were to give us some space to "paste" the picture in (about 2 centimeters of vertical space, 1 inch

of horizontal space, and offset from the left-most margin by 2 inches),¹ and the `\special...` is our way of asking for the above drawn line. Here, we must use a “`\special`” to talk to T_EX at this level. Later, *T_EXtyl* will actually do the work involved in typesetting that line that T_EX recorded as being at this place on the page. The `\special` is a way to make a kind of note or memo that T_EX will write into the DVI file for some other program like *T_EXtyl* to work with, or choose to ignore, like most DVI filters do.


Let’s look at what `\special{tyl line 4 20, 0; 80,30}` is all about. First, `tyl` is a name-string asserting that this command should make sense to *T_EXtyl*. Next, `line` is the name of the graphic primitive to typeset: a line segment (we will get to the other primitives later). The rest of the “special” notes that the thickness of the line is 4, and to draw the line from a mark 20 printer’s-points to the right of the current position to a mark 80 points over and 30 points higher from that current position. Mathematically, we are using a **first-quadrant** cartesian coordinate-space whose origin is at the current position on the page, and we are drawing a line from (20, 0) to (80, 30) in integer units of printer’s-points in that space. It looks like this (magnified):



where we have emphasized the origin of the quadrant, and used tick-marks for every ten printer’s-points. In reality, someone preparing a figure would sketch something out on grid paper, and then use those coordinates for use in the `\special`. If he were really in luck, he could use a graphic editor that would take care of such numeric details, and maybe even output the `\special` strings for direct insertion into the `.tex` file. Maybe. For now, we will have to stick to the grid-paper method. The only things that we have

¹See also Appendix C

to remember are that the origin of our quadrant is placed at the position on the page where we invoke the `\special` of `TEX`, and that we may want to leave white space for our graphic to be drawn in. This last constraint is because `TEX` does not really know about the size of our drawing, it thinks that the `\special` is a “box” with zero height and width placed at the current position on the page. It is up to us to decide how much *real* space to tell `TEX` to leave for our drawing to be put into later by `TeXTyl`.

In the example above, and the following examples, we will put our `\special` strings within an environment that makes it easy to specify where the origin of our quadrant should be placed. For example, `\begintyl{4cm}[1in]` tells `TEX` to place our origin 4 centimeters down from where it was just sitting. A horizontal width is an optional second parameter to `\begintyl`, and is enclosed in square braces immediately following the vertical displacement parameter (with no spaces in-between). So `\begintyl{5cm}` would place our origin 5 centimeters below the current position, with no relative horizontal width. Other spacing requirements have to be done by hand (e.g., like typing `\hskip 1in` before the `\begintyl`), or by putting the whole `\begintyl ... \endtyl` group within a containing environment (e.g., `LATEX`'s `figure` environment). We could, if we wanted, leave no extra white-space and have the drawing extend into and over any text that `TEX` may typeset before *or* after we invoke a `\special{tyl ...}` command. We would, of course, start that tiling-environment with a `\begintyl{0pt}` command and finish with the `\endtyl` command, and it might do something like  this.

Once we are satisfied with our `.tex` file containing `\special` strings requesting line-drawings, we run the `.tex` file through `TEX` or `LATEX`, and (barring any fatal errors) end up with a `.dvi` file. We now run this `.dvi` file through `TeXTyl` which goes through the file and converts our `\special` strings left-over from `TEX` into commands to actually typeset the line-drawing. The whole process might look something like this:

```

prompt:tex
This is TeX, Version mumble...
**myfile.tex
    ... output from TEX
Output written on myfile.dvi ( n pages , m bytes).
prompt:textyl
This is TeXtyl, Version blah...
DVI-input File Name:myfile.dvi

```

```

DVI-output File Name
(different than input name)[default of myfile.tyl]: myfile2.dvi
... output from TEXtyl
Output written on myfile2.dvi
Log written on myfile2.tlog

```

So there you have it. All we have to do is give this new .dvi (or .tyl) file to a favorite DVI filter, and look at the document with the line-drawing results on that particular filter's output device (usually paper or a bitmapped screen). You should contact your local maintainer if *TEXtyl* or the filter cannot find the right fonts, and try again.

By the way, the .tlog file is useful for finding out about the "tyling" of the .dvi file. Besides noting any errors, the log file contains the approximate height and depth of each non-trivial figure. A portion of the .tlog file for this document might look like:

```

Figure #1 on page 18 is approx. 33 pts high and 47 pts wide

Figure #2 on page 18 is approx. 32 pts high and 51 pts wide

Figure #3 on page 18 is approx. 27 pts high and 45 pts wide

Figure #4 on page 18 is approx. 31 pts high and 82 pts wide
18] [19] [20] [

```

This information allows you to re-edit the parameters to `\begintyl` so that they can better approximate the actual sizes of the figures.

Let's try a slightly more complicated example: drawing a spline. *TEXtyl* understands drawing smooth spline curves through integer coordinate points (both positive and negative) on our grid. For example:

```

\begintyl{3cm}[1in]
\special{tyl spline m 4 K 7 5,10; 9,14; 15,7;22,13;
                    17,14;12,5;7,0;}
\endtyl\par

```

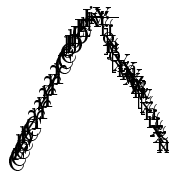
yields



Here, we are using units of millimeters for measurement (denoted by an `m` or `M`, a line thickness of 4, and we want to use a Catmull-Rom type spline (denoted by `k` or `K`—also the default type if none is specified). We specify that there are 7 control points to interpolate through, and then list those x, y pairs. Some things are optional for you (like the spline type, or the units of measure), but some are *not* (notably the line-thickness and the number of control points). As we go through these examples, I will point out the minimum that we need to specify a graphic primitive, and note other options or features that we might want to use. For a detailed look at the specifications of the primitives, please refer to the *User-Level Details* in chapter 3.

Let's try one more type of spline-primitive: the thick-n-thin spline. This is a primitive that lets us specify a spline of varying line thickness along the spline. An example invocation might be:

```
\special{TYL ttspline m 5 4,6; 14,25; 18,19; 21,15; 24,8;
                    10;    2;    6;    3;    8}
```



It looks similar to the `spline` primitive, but here we did not specify a single line thickness right after the `m` measure-marker. The `5` refers to the number of control-points as before, then is followed by the (x, y) coordinate pairs, and then the line-thicknesses. The first number (here, `10`) refers to the line thickness at the first control-point (here, $(4, 6)$); the next for the second control-point, etc. We did not need to align the thicknesses under the control-points, but it makes the correspondence more apparent. We did not specify the type of spline to use, so the Catmull-Rom interpolating spline was used by default. We could have inserted a `b` marker directly before the number of control-points, and gotten a spline using the B-spline basis. Using a `d` marker yields a spline with the Cardinal basis, and an `i` would indicate an Interpolating B-spline (which is a little different). We will look closer at these spline-types on page 14.

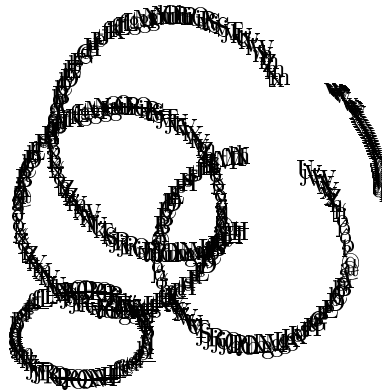
If you've been very observant, you may have noticed that the way we type in the `\special` strings (the letters and numbers within the `{ - }` pair) has been rather indiscriminant about upper vs. lower-case. That's okay. *TEXtyl* has only a few things to worry about, and so it will try to figure things out for you. It is pretty lenient about how you punctuate and separate ("delimit") keywords and markers.

We only have to remember

1. the relative ordering of the keywords and integers,
2. being careful to avoid mis-using the word markers² in the wrong places,
and
3. supplying enough parameters that are expected.

Other than its own macro-expansions, $\text{T}_{\text{E}}\text{X}$ really doesn't care about the “meaning” of the final strings that goes inside the curly braces of the `\special{...}` command-strings, only *T_EXt_l* does.

We'll briefly look at an example of one more graphic primitive, and then get on to more advanced topics. *T_EXt_l* has the ability to draw arcs and circles like:



²See the index for a list of markers

Arcs are drawn from an initial angle (measured from the horizontal (positive x-axis) in integer degrees) counter-clockwise to a final angle with the arc centered about the origin. If the initial angle is the same as the final angle, we get a full circle. A simple arc invocation might look like:

```
\special{tyl arc m 2 3, 48, 280}
```

which would draw an arc of line thickness 2, a radius of 3 millimeters, from an initial angle of 48 degrees until the final angle of 280 degrees centered about the origin. We could specify that the arc is to be centered elsewhere by using a special @ marker. So,

```
\special{tyl arc m 2 3, @ 7,7; 10 10 }
```

would draw an arc of line-thickness 2, radius of 3 mm as before, but centered at (7, 7) from the origin. Also, since the initial and final angles are the same in this example (10), we would get a full circle.

Chapter 2

Advanced Features

In the previous chapter, we saw basic examples of most of the graphic primitives available in *TeX*, how to invoke them, and some of the parameters to the primitives. We also saw examples of how we interface with \TeX using the `\special` command strings, and how the origin of our drawing quadrant is placed at the reference point of the “box” containing the `\special` (usually the lower-left corner of the box created with the `\begintyl` and `\endtyl` macros. (see the \TeX book[9] for a better explanation of “specials”). The rest of this chapter will look more at the parameters available for the primitives, and how we can combine primitives and manipulate them.

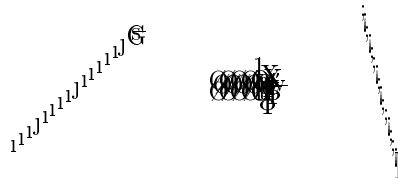
2.1 Transforms

The most interesting additional parameter for the primitives is a transform operation. It allows us to modify an already-existing definition of a primitive without our having to totally re-edit the `\special` string, nor having to explicitly re-compute the coordinates of the line/spline points. Thus we can take a simple definition similar as before, like:

```
\special{tyl line m 4 0,8; 39,44}
```

and change its size, or rotate it, or translate (shift) its position without completely changing the actual text of the specification of the control points (this is really useful when dealing with splines having many control points).

Let's look at an example, and then go into more detail about the transform. For example, to rotate the above definition about its center by 60 degrees counter-clockwise, all we have to specify is `\special{tyl line m T (100, 100, 0, 0, 60) 4 0,8; 39, 44}` which yields



as expected. We use the T marker to note the need for a transformation, and follow it by five numbers. The first two are for scaling, the next two for translation, and the last (here, 60) is for our rotation. In general, the format for specifying some transformation on the currently-specified points is:

`T <Sx> , <Sy> , <Tx> , <Ty> , <Rot> ,`

where $\langle Tx \rangle$ and $\langle Ty \rangle$ are translations of the object in the X or Y direction, respectively, by some signed distance according to the current units of $\langle measure \rangle$ (m still means millimeters in the above example). $\langle Rot \rangle$ is a signed integer angle (in degrees) by which to rotate the primitive counter-clockwise about its center. $\langle Sx \rangle$ and $\langle Sy \rangle$ are integer scale parameters representing the real values of the transform multiplied by 100. For example, if you wanted to scale the drawing in the X direction (relative to the drawing's center) by an additional 50% (i.e., scale by 1.5), $\langle Sx \rangle$ would be 150. Negative values are usable, too, so mirroring about the Y axis is achievable by scaling in the negative X direction, like $\langle Sx \rangle = -100$. To obtain any transform, *all* the transform parameters must be specified. A no-op transform would look kind of like `T (100, 100, 0, 0, 0)` in the middle of the `\special`.

2.2 Figures

The last major *TEXtyl* concept is the idea of combining several primitives into what we call a "figure." This figure can be manipulated either as a single entity or in parts by using the $\langle transform \rangle$ operations described in the previous section. These optional figure-level transformations will transform all the figure's primitives (which may have local transformations of their own). The result is a nested symbol definition that has concatenatable transformations.

This is useful for defining some symbol, created from various primitives, and dealing with it as a unit. For example, a basic logotype can be defined, and then moved about, scaled, or rotated as desired *without* having to change the original definition of the logotype or its parts from scratch. These commands are:

```
\special{tyl beginfigure [(transform)] }
```

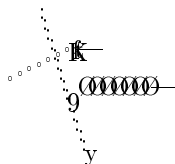
which opens a level of definition, and

```
\special{tyl endfigure}
```

which closes that level of definition. Any calls to a `\special{tyl ... primitive}` within a `beginfigure – endfigure` pair become part of that figure's definition. Note the definition of `beginfigure` allows the ability to apply a transformation at the outer level. A simple example might be:

```
\begintyl{2in}
\special{tyl beginfigure}
\special{tyl line m 2 5,10; 15,15 }
\special{tyl line m 4 10,20; 16,1 }
\special{tyl line m 8 15,10; 25,10 }
\special{tyl endfigure}
\endtyl
```

giving a figure of three lines:



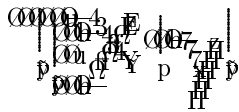
We can also define sub-figures within figures and apply transformations to those, too. The idea looks like:

```
\begintyl{down-distance}[optional-width]
\special{tyl beginfigure}
...
\special{tyl beginfigure} % Some sub-figure
... % of other primitives
\special{tyl endfigure}
...
\special{tyl endfigure}
```

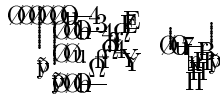
`\endtyl`

Where “...” means possible other invocations of primitives with `\special{tyl...}` strings. We are able to use spacing and tabbing to nicely indent our `\special` strings, since T_EX considers all that white space to be non-existent within the `\begintyl – \endtyl` environment.

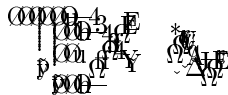
We’ll look at a basic figure, and apply some transforms to parts of it. For the basic figure, we have:



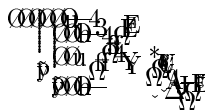
Now let’s scale a sub-figure by using a T transform at the `beginfigure` level; i.e., something like `\special{tyl beginfigure T 60 60 0 0 0}`:



We can rotate, and then also translate that same sub-figure and achieve the next two examples. First, a rotation about the sub-figure’s center:



and now also translated *after* the rotation:



Finally, we can take the whole figure, and perform a scale and a then a rotation on it (we evaluate scaling first, then rotations and finally translations):



2.3 Others

We'll finish off this section with a couple more primitives: a simple spline example, and the primitives for a music-typesetting project underway at Ohio State University[7]. The first example is not really that different from our first spline example, except that in this case, we have a “closed” spline: one whose first control-point will coincide with its last control-point. We can type something that looks like:

```
\special{tyl spline m 3 b 0 8 (10,9) (4,16) (8,23)
(11,21) (13,17) (20,16) (22,13) (19,8)}
```

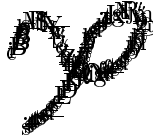
The 0 (letter “Oh”) marker denotes a closed spline. The default option is an “open” spline, and can be marked with a U instead of the 0 if you wish to be explicit. The difference is in the way that *TEXtyl* manipulates the control points when doing the interpolation. In this example, we used a B-spline, as marked with a b, and we listed *only* the unique coordinates to get this nice bean-shape:



TEXtyl provides three types of spline interpolation which I alluded to on page 6. The Catmull-Rom basis is probably the most intuitively useful spline for users. It produces a smooth curve that goes through the control points (which we can mark with dots):



The Cardinal basis is of the same family as the Catmull-Rom. It, too, interpolates through the control points, and is included for convenience and as an alternative to the Catmull-Rom. For example:



The B-spline is a little different in that the curve *approximates* the control points. Using the same control points as the above example, but using the B-spline basis, we see:



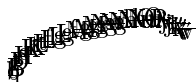
For the most part, you will probably want to draw curves through the control points that you specify, and so the Catmull-Rom basis is the convenient default. The B-spline type is provided for completeness for users who might have data using that basis.

There *are* times when the Catmull-Rom or Cardinal bases give flakey curves and the B-spline basis is not appropriate either. *TEXtyl* provides a fourth type of spline for these situations: an interpolating B-spline curve. It has the advantages of Catmull-Rom's interpolation, and the special flexibilities of the B-spline, but is computationally more expensive. See [14] or [3] for a complete explanation. Here is an example using the same control points as the above examples, but using the interpolating B-spline:



For the *MusiCopy* project, we have the ability to draw ties and slurs—the long arcs connecting groups of notes. For the most part, I assume that there is a program that is going to take care of the details of knowing where these graphic elements are to be placed, and will produce the correct `\special` string for inclusion in the `.tex` file, but I will give an example for completeness.

This graceful arc is produced by specifying a minimum and maximum thickness, and the five control points for the shape.



The call looked like:

```
\special{tyl tieslur m 2 8 5 (5,10)(7,13);[15,16];(24,17);  
28,15 }
```

where 2 and 8 are the min and max thicknesses at the endpoints and the middle, respectively. The following 5 is for the number of control points, and the five pairs of integers following that are the coordinates of the points. We use a special way of producing a smooth and gradual thin to thick to thin line-thickness, but will discuss it in a later chapter.

The last example is the most specialized for the music project: beams. We'll just look at an example of how they are called and appear, and leave the details for the next chapter.

The basic order of the parameters looks something like:

```
\special{tyl beam m 0 g 3,10;11,7 }
```

where 0 (zero) is the “staff-size,” *g* indicates a beam size for “grace” notes (*r* is the default for “regular” sized notes), and the last two pairs are the coordinates of the line-segment that the beam is to be centered over.

Chapter 3

User-Level Details

From the user's point of view using $\text{T}_{\text{E}}\text{X}$, commands that *T_EXtyl* interprets will look like the word “`tyl`”, followed by the name of the graphic to typeset, then a list of parameters all enclosed within the curly braces of a `\special{...}` command. This use of a `\special` could be produced by some other program (a graphic editor, for example), and the text strings merely inserted into the `.tex` text-file.

As far as the user is concerned, he is pasting in a picture at the current position on the page where he invokes the `\special`. The user should think of this picture as being a box just big enough to contain all the lines of his graphic image, and whose first-quadrant cartesian origin is the reference point that will be placed at the current position on the page. Thus, he might want to make sure that there is enough blank space around the current position on the page to contain the image before he tries to “paste it in.” Space can be created by using $\text{T}_{\text{E}}\text{X}$'s `\hskip` or `\vskip`, or $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$'s `\hspace` and `\vspace` commands, or hoping for the best with $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$'s `figure` environment surrounding the box created by `\begin tyl` and `\end tyl`.

The kinds of graphic primitives that *T_EXtyl* knows how to typeset are uniform-thickness line segments, uniform-thickness splines, uniform-thickness arcs, variable-thickness splines, music beams, and music ties/slurs. Line segments are merely a subset of splines, and so variable-thickness lines are available by specifying a spline of just two control points with a thickness at one end point, and another thickness at the other end point. The feature here is to have several primitives available, and allow for other graphic elements to be built using them.

We'll look at the basic syntax for the primitives, and then discuss what each parameter really means and how to specify what you want. The parameters for each of the graphic types are as follows: literals are in a **typewriter** face, any required parameters for the primitive are in $\langle \text{angle} \rangle$ brackets, and any *optional* parameters are within $[\text{square}]$ brackets. Please note that the relative ordering of the parameters *is* important, even if you do not use any or all optional parameters. Also, it does not matter if you use upper- or lower-case letters for the primitives or markers.

```
\special{tyl line [ $\langle measure \rangle$  , ] [ $\langle transform \rangle$  , ]  $\langle thick \rangle$  ,
                [ $\langle vector \rangle$  , ] [L  $\langle style \rangle$  , ]  $x_{left}$  ,  $y_{bottom}$  ,  $x_{right}$  ,  $y_{top}$  }
```

draws a line segment from the point (x_{left}, y_{bottom}) to the point (x_{right}, y_{top}) using a line of thickness $\langle thick \rangle$.

```
\special{tyl spline [ $\langle measure \rangle$  , ] [ $\langle transform \rangle$  , ]  $\langle thick \rangle$  ,
                  [ $\langle vector \rangle$  , ] [L  $\langle style \rangle$  , ] [ $\langle s-type \rangle$  , ] [ $\langle closure \rangle$  , ]
                  [X  $\langle dotsize \rangle$  , ]  $\langle numpts \rangle$  ,  $x_1$  ,  $y_1$  , ... ,  $x_{numpts}$  ,  $y_{numpts}$  }
```

draws a smooth spline curve through the integer points $x_1, y_1, \dots, x_{numpts}, y_{numpts}$ using a line of thickness $\langle thick \rangle$. If the optional **X** marker is specified, dots of diameter $\langle dotsize \rangle$ are placed to mark the position of the control points specified. This may be useful for checking your data.

```
\special{tyl ttspline [ $\langle measure \rangle$  , ] [ $\langle transform \rangle$  , ]
                    [ $\langle vector \rangle$  , ] [ $\langle s-type \rangle$  , ] [ $\langle closure \rangle$  , ] [L  $\langle style \rangle$  , ]
                    [X  $\langle dotsize \rangle$  , ]  $\langle numpts \rangle$  ,  $x_1$  ,  $y_1$  , ... ,  $x_{numpts}$  ,  $y_{numpts}$  ,
                     $thick_1$  , ... ,  $thick_{numpts}$  }
```

draws a smooth spline through the integer points x_1, y_1, \dots using a line of varying thickness defined by $thick_i$ at each control point (x_i, y_i) (ergo thick-thin splines). The **X** marker is also available for marking the positions of the specified control points.

```
\special{tyl arc [measure , ] [transform , ] thick ,
  [vector , ] [L style , ] radius , [C cent_x , cent_y , ]
  angle_1 , angle_2 }
```

draws an arc of radius $\langle arc\text{-}radius \rangle$ going counter-clockwise starting from $\langle angle_1 \rangle$ degrees from zero, and finishing at $\langle angle_2 \rangle$ degrees around from zero. If the center point is not specified with the **C** marker, the arc is assumed to be centered at (0, 0). Ellipses can be achieved by a simple scaling of a closed arc since scaling and rotational transformations are about the *center* of the primitive.

```
\special{tyl label [measure , ] face , x , y , " string " }
```

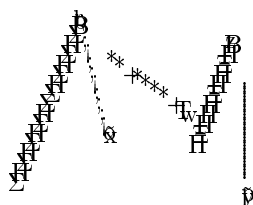
places a simple label at (x, y) using a font style of $\langle face \rangle$ (currently selectable by an integer: e.g., 1 is amtt10. See page 37 for a better list). $\langle String \rangle$ is a simple sequence of letters and spaces contained within matching double quotes ("). Here, the case of the letters of $\langle string \rangle$ is taken verbatim, and it is *not* interpreted; T_EX macros or typesetting commands are not usable here. If you want to typeset fancier labels, *T_EXtyl* is too late in the game.

The optional parameter $\langle measure \rangle$ is any of **S**, **P**, or **M** denoting that the integer points, like x_1, y_1, \dots are measured in scaled-points, printer's-points, or millimeters respectively. If none is specified, the measurement of printer's-points is assumed. Just to refresh how big each of those units is:

- there are 72.27 printer's points (*pp*) per inch
- and 65536 scaled-points (*sp*) per printer's point
- and 25.4 millimeters (*mm*) per inch

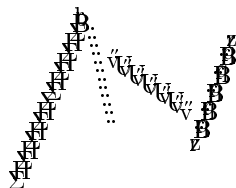
Also remember that the coordinates for the above control points (like x_{left} or x_i) are in *first quadrant* cartesian coordinate-space. It is useful not to have to remember some other coordinate systems, and makes things easier for the user and user programs to interact with *T_EXtyl*, which takes care of such details.

$\langle Thick \rangle$ is the pixel-thickness of the vector font to use. Currently the range is from about 1 to 12 pixels thick (we'll talk more about these sizes later). Here are some examples of lines having from 2 to 10 “pixels” in even thicknesses.



$\langle Vector \rangle$ refers to the *type* of vector to use, namely **C** for vectors that look as if they are drawn with a *circular* “pen,” **H** for a *horizontal* pen, and **V** for *vertical* pen appearance. A circular pen vector-type is the default. See also section 5.1 for examples and a better description.

$\langle Style \rangle$ is the line-style to use when drawing. Currently, *TEXtyl* is able to provide solid (style 0), dotted (style 1), dashed (2), and dot-dashed (3) line-styles. The solid line-style is the default. Here are some simple examples of the styles:



$\langle S-type \rangle$ is the kind of spline-basis to use when interpolating the spline through the control points. Currently, **B** indicates the *b*-spline basis (which interpolates a spline within the convex hull of the control points), **I** indicates an *interpolating* B-spline (goes through the points specified), **D** for the Cardinal basis, and **K** which indicates the Catmull-Rom basis (which also interpolates a spline *through* the control points). The Catmull basis is the default spline type in the current implementation.

$\langle Closure \rangle$ is used to indicate whether the spline is a closed curve (the endpoints overlap), or an open curve (the default). We use the iconography of **O** for a closed curve, and **U** for an open-ended curve.

Now we come to describe what *⟨transform⟩* is all about. *TEXtyl* allows the user to modify the coordinates of the control points without having to recalculate their new positions by himself. I tried to make the specification and modification of *TEXtyl*'s primitives as painless and intuitive as possible. We usually think more in terms of connecting dots and placing graphic elements, and not in terms of slopes, tangents and velocities which are messy at best, and get more cumbersome from there.

Geometric transforms like scaling (independently in the *X* and *Y* directions), rotating about the “center” of the graphic object (a primitive or a figure), and translating in the *X* and *Y* directions are available for most of the primitives. The format for specifying some transformation on the currently-specified points is:

T, *⟨Sx⟩*, *⟨Sy⟩*, *⟨Tx⟩*, *⟨Ty⟩*, *⟨Rot⟩*,

where *⟨Tx⟩* and *⟨Ty⟩* are translations of the object in the *X* and *Y* direction by some signed distance according to the current units of *⟨measure⟩*. *⟨Rot⟩* is a signed integer angle (in degrees) by which to rotate the primitive or figure counter-clockwise about its center. *⟨Sx⟩* and *⟨Sy⟩* are *integer* scale parameters representing the *floating-point* values of the transform multiplied by 100 and truncated. To obtain any transform, *all* the transform parameters must be specified.

Two other special types of graphics capabilities are meant to be utilized by a knowledgeable program, such as the music-typesetting system being written at the Ohio State University. These are:

\special{tyl tieslur} [*⟨measure⟩*,] *⟨minthick⟩*, *⟨maxthick⟩*,
⟨numpts⟩, *x*₁, *y*₁, . . . , *x*_{numpts}, *y*_{numpts}}

which draws a smooth spline curve through the points (*x*_{*i*}, *y*_{*i*}), and uses a built-in algorithm to figure out the correct thicknesses of the tie/slur between *⟨minthick⟩* and *⟨maxthick⟩*. These control points are also in first quadrant cartesian coordinates, and a circular-pen vector-type is used to draw the spline.

\special{tyl beam} [*⟨measure⟩*,] *⟨staffsize⟩*, [*⟨beamtype⟩*,]
*x*₁, *y*₁, *x*₂, *y*₂}

draws a music beam from (*x*₁, *y*₁) to (*x*₂, *y*₂) (also in first quadrant space) using a beam of *⟨beamtype⟩* **G** for a *grace*-note sized beam, or **R** (the default) for *regular* sized beams in the specified staff-size. For a description of staff sizes, see [12].

Finally, there are two more `\specials` that *TEXtyl* can understand. These are used to group any of the above graphic primitives together into a “symbol.” The commands are:

```
\special{tyl beginfigure [<measure> ,] [<transform> ,]
                        [W <width> , <height>] [F <width> , <height>] }
```

which opens a level of definition, and

```
\special{tyl endfigure}
```

which closes that level of definition. These are analagous to *push* and *pop* kinds of operations, with many levels of recursive definition available. The optional **F** marker specifies the final optimal width and height of the figure, in terms of units of *<measure>*. This is useful for fitting a figure to a specific size, so that you do not have to compute the scaling parameters by yourself. This ability is often desirable if the figure was created by some program that output the figure at a different scaling factor than you require on the page. The **W** marker indicates the width and height of the figure at the size that it was originally written, which we assume the figure-making program output along with the definition of the figure elements. For most practical purposes, you can let *TEXtyl* compute the size of the figure as it was originally created, and then let it fit the figure to the sizes requested by the **F** marker.

Besides the implicit transform parameters created by using the **F** and/or **W** markers, an explicit figure-level transformations can be specified using the *<transform>* capability using units of *<measure>* (or the default units of printer’s points). Also, any transforms applied at a level of definition are additive over the lower-level definitions and any transformations that they may have locally.

Chapter 4

When Things Go Wrong

Despite the best of examples and directions about careful typing of `\special` strings, there are times when *TeX*tyl will get upset and complain about some situation or input. Fortunately, *TeX*tyl can continue from most of them (like missing or incorrect parameters), and just report them in the `.tlog` log-file for you. An example run of *TeX*tyl might look like:

```
>textyl
This is TeXtyl, version ...
  DVI-input File Name: myfile.dvi
  DVI-output File Name :
(different than input name)[default of myfile.tyl]
  [1] [?2]
Output written on myfile.tyl
Log written on myfile.tlog
Some error(s) occurred. Please check Logfile for details.
Assume that the outputfile is incorrect
```

Rather than annoy you at the terminal with long error messages, *TeX*tyl puts a `?` mark to indicate that some error happened. Since it wrote the `.tyl` output file, *TeX*tyl must have recovered well enough to finish its job, but the output probably does not look right. The `.tlog` file might look like

```
This is TeXtyl, Version blah...
Reading File: myfile.dvi
[1] [
Error in fig# 1 on page 2
Arc Thickness not found. Setting to 1
2]
Output written on myfile.tyl
Log written on myfile.tlog
```

which was just a missing parameter to the `\special{tyl arc ...}` string. You will have to re-edit the file and insert the correct thickness. All the error messages that are written to the `.tlog` file try to pin down the error to the figure number on the error-containing page. These references to the position of the error come from the n^{th} figure on the page (the n count is relative to each page, and starts from 1), and the page number is from `TEX`'s `\count0` register that enumerates pages. Sometimes *TEXtyl* cannot always determine the exact figure on the page (e.g., if we forget to close a figure definition with an `endfigure`), and so estimates at least the bad page in question. For the most part, the error is a just missing parameter, or a bad parameter to a `\special` string being read by *TEXtyl*.

However, there are a larger number of errors that *TEXtyl* found “surprising” and so marks these errors on your screen as `?!`. Most of these internal errors are based on the functionality of `DVItype` concerning `TFM` information, and the structure and commands of the `DVI` file being read. *TEXtyl*, like `DVItype`, will complain, and possibly roll over and die gracefully.

The other “surprising” errors that *TEXtyl* might complain about are really internal to *TEXtyl*, and constitute a bug, or just a compile-time constant that is too small. These easily-fixed surprises are

- **too many dvistrings.** This page required more DVI bytes than usual. Have your local maintainer increase the size of the internal buffer used for storing a page’s DVI bytes.
- **too many spline segments required.** The spline you tried to set might be too large/long for the space allocated for a spline’s description. That size is also a compile-time constant. *TEXtyl* will reduce the number of control points that it interpolates so that you can at least get the output, and so that it does not die ungracefully.
- **figure definition not closed at end of page.** You forgot a `\special{tyl endfigure}` command in some figure definition on the page. *TEXtyl* will not typeset that figure, and will go on to the next page.

There are also some down-deep errors that could potentially (although doubtfully) show up. These are probably some error in the vector fonts being referenced by *TeX*tyl.

- **min radius of vector font is zero.** The referenced vector font's TFM information reported that the font's radius size is smaller than it was designed to be. *TeX*tyl will make a temporary fix so that it can continue, but the output figure will be wrong. Have the local maintainer re-check the vector fonts.
- **bad dydx.** This is a rare problem. Somewhere, the code to typeset a diagonal line segment referenced an impossible dy/dx combination, and cannot find any vector character to fit. *TeX*tyl will continue, but you may see a tiny glitch in your figure.
- **dx is too big/small, dy is too big/small.** This is a problem. For some reason, a distance was referenced that is out of the range of the 32-bit representation of integers. This really should never happen.

For the most part, *TeX*tyl can recover from errors that it encounters, and give a brief summary in the `.tlog` file. Other errors have to do with TFM file information, and are outside the scope of *TeX*tyl to handle gracefully. The most serious “surprise” errors are bugs, and you should report them directly to me with a copy of the `.tex` file, or at least a detailed printout from DVItypex of the `.dvi` file processed by *TeX*tyl.

Chapter 5

Design Details

The top-level of *TEXtyl* is taken directly from the DVItypex[5] program, with a few modifications. Most of the diagnostic functionality of DVItypex is maintained, but without the user interface (*TEXtyl* assumes it is supposed to process all the pages). A “hook” was inserted into the program to handle the \TeX `\specials` that *TEXtyl* understands (hereafter, “special” will refer only to those commands that *TEXtyl* understands—other `\specials` are ignored, and simply output to the DVI file without modification). The DVItypex code was used mostly for its keeping track of the current DVI **h** and **v** position coordinates on a page, and handling TFM information.

Basically, as *TEXtyl* slurps in a DVI file, it processes “normal” commands in a normal way (i.e., just keeping track of positions and counters). When it reads a `\special` string, it tries to interpret the parameters and expand the special, producing commands to typeset the graphic using characters from special fonts.

The result is a transformed DVI file that now contains “normal” (i.e., non-special, ‘put-that-character-there’) dvi-commands as far as *TEXtyl* can tell, after taking care of new font definitions and doing any bookkeeping of counters and internal references.

5.1 Vectors

Several design decisions were made to give *TEXtyl* the ability to typeset the graphics of the complexity that we desire. This ability is intrinsically bound

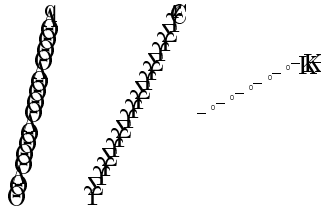
to the fonts to be used, and the way that we use them. For this project, I had to create a font of short, oriented line segments. This was to minimize the amount of information required to typeset a curve. One could address each pixel of the virtual device to draw the graphic (putting hundreds of periods from a font), but that is not device-independent, and the amount of data is huge in comparison to using line segments. These line segments can be connected together rather smoothly to give the appearance of a continuous line segment or curve.

Using this scheme, described by Nelson and Saxe at Carnegie-Mellon University[11], I am able to typeset the correct character corresponding to the desired length and angle. I modified their font those angles cover the first and fourth cartesian quadrant system, and whose vectors are from the origin to points clockwise along the perimeters of semisquares of decreasing radii. They use this scheme in order to use the longest vectors possible for a given distance (thus minimizing the number of characters to typeset), and also have shorter vectors for use in tightly curving lines. They use a semisquare (rather than a semicircle) for a quick line-tangent intersection test in their algorithm to decide the maximum length vector to use that has a satisfactory variance from the tangent to the line. I really must give correct thanks to Dario Giuse at C-MU[6] for first implementing the vector-font typesetting code that I subsequently modified for *TEXtyl*.

I modified the vector font for use with Metafont 79[10], and was able to describe it in a device-independent manner. Because of the limitations of Metafont 79's and \TeX 's representation of character heights and depths, I could not accurately describe the character dimensions in such a way that I could leave the low-level typesetting details to \TeX through its idea of TFM information. Thus, *TEXtyl* has to take care of the details of calculating the correct dimensions for each character of each vector font. This calculation is done, however, only as a particular font is first required, and the dimensions are computed on-the-fly using the group of parameters found in the TFM file that we have defined for our own use. After this initial parameter calculation, subsequent references to that font's TFM information is accessible via a caching scheme. The fonts contain such information as the measure of the maximum-length vector of the font, and the height and width of the pen used to draw the font (*all in scaled points*).

Current plans for the vector fonts are for sizes in the range from about 1 to 12 "pixels" for each of the three types of Metafont pens (circular, flat

horizontal, and flat vertical). The fonts are actually device-independent in size, as our “pixel” is measured in absolute units of 16384 (2^{14}) scaled-points. However, there is a step within Metafont 79 that insures that the vector size is an odd-number of physical pixels computed at the final output-resolution. This is to avoid some discretization errors and the accompanying aliasing artifacts. I expect the vectors drawn with circular pens (the “cvec” fonts) to be the most utilized. Their endpoints are rounded and are positioned to coincide so that the segments overlap slightly and meet up gracefully producing a smoother curve. Here are some examples of circular, horizontal, and vertical pens that can be used for different effects:



Beams

A similar scheme is used for the beam characters found within the music fonts. The beam characters are in two types (those for grace notes, and those for regular notes) for the staff sizes of music (numbered 0 down to 8). See also [12]. The font is also a set of short line-segments that will be connected to create a beam (some other program will have to know about the details of describing how to attach stems from the beams down to the notes). However, the beam characters are different from the vector fonts in that each staff size of music has one particular size of beam characters associated with it.

The beams are designed to appear as if they are drawn with a flat-edged pen held perpendicular to a ruler (at all orientations). When a vertical pen is drawn along a path, the actual cross sectional distance of the line segment decreases with the angle (i.e., it is thickest when drawn along a line at zero degrees from the horizontal, and thinnest as it is drawn along a line approaching ± 90 degrees). The way to maintain the appearance of consistent thickness of the line segments at all drawn angles is to actually *increase* the “height” of the vertical pen as a function of the angle from the horizontal. This function was geometrically derived to be the *secant* of the angle, which yielded in a scaling factor to apply to the original definition of the vertical

pen's height.

Another difference from the vector fonts is that the beam characters do not have to cover as large an angular range. The angles from the horizontal are less than 45 degrees on the average, and the lengths of the beam characters are no smaller than the width of a quarter note for that staff size. The current implementation defines the beam characters in four groups per staff size: regular beams in lengths of 1.0 and 1.5 quarter-note-lengths, and grace-note beams in lengths of 0.5 and 0.66 grace-note-lengths.

The scheme of typesetting is similar to that of the vectors, but an added constraint is resolved. Since the beam characters use a slightly smaller set of fixed angles from the horizontal, we have to choose the best beam character to use such that we use the maximum length beam character *and* try to obtain the least deviation from the angle requested to fit the beam font. The dimensions of these beam characters are computed only as each music font is required, and a caching scheme is maintained to avoid unnecessary re-loading of font information and subsequent definition in the DVI file.

5.2 Splines

A second decision that I made for the current implementation of this system was to use Catmull-Rom[4] splines as the default instead of B-splines. The reasons for this are two-fold: (1) Catmull-Rom splines are of the same family as B-splines, but are *guaranteed* to produce a smooth curve *through* the control points, whereas B-splines only approximate within the convex-hull described by the control points. Therefore, Catmull-Rom splines are a true interpolant; and (2) when dealing with the ties and slurs, the user wants the curve to go through the five points that he specifies (left, middle and right points). With B-splines, he would have to *estimate* the position of the middle control points in order to get the curve to be in the correct position. Catmull-Rom and Cardinal splines are just as easy to calculate, and usually achieve the effects that we need.

I have also experimented with operations on the B-splines to achieve true interpolation. This method was described by Barsky and Greenberg[3] describing how to re-compute B-spline control-points so that the resulting spline would interpolate through desired points (inverse interpolation). It actually does not require very hairy mathematics. Arcs and circles are im-

plemented using this spline primitive. The arc control-points are computed (or pre-computed in the case of circles, and ellipses which are just scaled circles) and then are used in the inversion procedure so that the resulting B-spline passes smoothly *through* those original points. We will later describe further this usefulness of implementing various graphic objects using lower-level primitives.

I also use the Catmull-Rom interpolation when computing the different thicknesses along both the tie/slurs and the variable-thickness splines (“ttsplines”). I need to interpolate from the current control-point’s thickness along the curve to the next control-point’s thickness, and can do this smoothly by using the interpolation methods available when computing the splines anyway.

Since we have the ability to draw splines of varying thickness, we can easily implement ties and slurs (those long arcs connecting groups of notes on a musical score), since they are splines of usually five control points, having the specified *minthick* thickness at the left and right, and *maxthick* thickness in the middle.

Chapter 6

The Implementation

6.1 Overview

TEXtyl is currently nearly 9000 lines of Pascal code (not WEB). It is a prototype, and *not* a product. Testing was simplified by the ability to pre-view the contents of the DVI file on Sun workstations using the DVISUN [8] program. It was written using Berkeley Unix¹ Pascal, but is not dependent on that operating system except for differences in I/O. There are also differences in efficiency/ease of reading from files—reads versus gets. I/O routines for both byte types (signed and unsigned) are written and available, and have been set up for easy porting to other machines.

The idea for handling DVI files is to read each byte into a buffer until we read an `eop` (end-of-page marker), when we will write that buffer out into a file, and start a fresh buffer with the next page. If we encounter a `\special` in our reading, we treat it as a macro-expansion—substituting *our* byte-string for the byte-string that represented the invoked special, and put this “tyled” string into the buffer for subsequent output. A nice feature of this slurping is the parser for the specials. It is simple and extensible enough to easily accommodate new *TEXtyl* primitives, and different or additional parameters.

Besides being careful about the expansion of the `\special` string, *TEXtyl* has to keep track of any newly-defined fonts for insertion into the `postamble` in the DVI file, and do some bookkeeping for `bop` backpointers, and file byte-

¹Unix is a trademark of Bell Laboratories

length.

Most of this chapter discusses the algorithms and data-structures of *TEXtyl* at various levels. We'll start with the high-level and user concepts, and work our way down into the details of outputting the actual bytes into the `.dvi` file.

6.2 Primitives

The user's interface to *TEXtyl* is through `.tex` text files. He types in the `\special` strings for `TEX`, and fills in the name of the primitive to typeset and any parameters to modify the attributes of the primitive. This `.tex` file is converted by `TEX` into a `.dvi` file which is then given to *TEXtyl*. For the most part, all of the graphic primitives in *TEXtyl* have common attributes of

- bounding-box,
- line thickness,
- pen-type (see section 5.1),
- line style

that describe basic appearances of an item. A “bounding-box” describes the minimum and maximum X and Y values of a minimum-sized orthogonal rectangle that encloses the primitive. I'll describe the primitives and point out their additional or modified attributes to those above. My plan was to provide a number of primitives that were conceptually built upon lower-level primitives, and share similar representations wherever possible. For example, a `tieslur` is clearly built upon the representation of a spline primitive. With this concept, I was able to represent the following primitives and effectively “draw” them by reducing to easier primitives, down to the vector-font level.

Line

The most basic primitive is the `line`, as all other non-figure primitives reduce to the same routines that produce line-segments. A line has attributes of bottom-left and upper-right endpoints, and is drawn from left to right in DVI-space, which is similar to fourth-quadrant cartesian space, but y -values are positive-increasing going down the y -axis. In section 6.5.1 I will describe how the line is “drawn” on the virtual page using the vector-font characters.

Spline

The spline primitives (`spline` and `ttspline`) have the attributes of a line, and in addition:

- spline type (see section 5.2),
- openness,
- a count, and a list of control-points.

The spline primitives all use the same basic algorithm for interpolation of its control-points (“knots”) using one of three spline bases: the B-spline basis, the Cardinal basis, and the Catmull-Rom basis. The difference is mainly in the choice of basis matrix.

`Ttsplines` have an additional attribute that describes the thickness of the spline at each control point. *TEXtyl* uses the same method of spline interpolation for computing the varying thicknesses over the length of the spline as for computing the position of the line-segment positions over the spline.

Ties

A `tieslur` is just a subset of `ttsplines`. It has exactly 5 control-points, which is sufficient to describe even the longest slur. It needs two specified line thicknesses: (1) a minimum thickness value at the far end points, and (2) a maximum thickness in the middle of the slur. *TEXtyl* uses an internal algorithm to figure the other thicknesses of the control points. This then reduces to a `ttspline` of five control points for interpolation. The only other difference from a `ttspline` is in the “clamping” mechanism. Let me explain: optimally, we would like the smoothest possible transitions of line thicknesses along a varying thickness spline, but this is still subject to the aliasing artifacts (“stair-steps”, “jaggies”) of the printing device’s resolution (you can’t

have 3.3 pixels in black and white). We could have a vector font for every possible pixel-size of the printer, and thus be able to achieve as smooth a line as the printer is capable. This could easily require the printer to use dozens of different fonts per page (imagine a page of math, text, and a complex figure) which most printers' limited font memories cannot handle

We are faced with either not being able to print the page (sort of defeating the whole purpose), or to reduce the number of fonts required to set that particular figure. This reduction is called “clamping.” We make sure that a requested vector size is in a pre-selected set of fonts, and then modify that size if it is not in that set. Usually this involves just adding/subtracting a vector-pixel unit and iterating again through the test-modify loop until the size fits.

Ties and slurs use the same basic clamping mechanism as `ttsplines` (described above), but also have a built-in way to select and compute the thicknesses along the length of the spline. This means that *TEXtyl* decides the thicknesses for the tie/slur using its own algorithm based on the user specifying the maximum and minimum thicknesses.

Arcs

Arcs are described in terms of a radius, an initial and final angle, and an optional center coordinate. Internally, this is transformed into a uniform-thickness spline. Closed circles are a special case of arcs, and so we can pre-compute the control points of the closed spline to describe the circle, and then just scale to fit the desired radius and translate its center to the required coordinate. Ellipses, although not a named primitive, can be simulated effectively by specifying a closed arc (a.k.a circle) and transform it by scaling as desired in either X and/or Y , since circles are a special case of ellipses.

Open arcs are represented by an open spline whose control points were computed by sampling along the length of the arc. We do this sampling in 16 places using the parametric representation of the arc:

$$x = r \cos \theta, \quad y = r \sin \theta \quad \text{for } \alpha_1 \leq \theta \leq \alpha_2$$

and θ is a multiple of $\frac{1}{16}(\alpha_2 - \alpha_1)$. After we obtain these control points, we can interpolate over the knots and obtain the spline segments.

In reality, we have to treat arcs with a little more special care than simple splines. In computing open splines, we have to do some tricks with the list of control points. We introduce “phantom” control points which help the

endpoints look nicer. See [14] for a better explanation of phantoms. The problem with arcs is that the endpoints would look too “flat” if we used the normal phantom-ing method. What I do is to create two extra control points (one before the start of the actual arc, and one after) so that we preserve roundness by continuing around the arc with the same parameters to the representation. Then I *lie* to *TEXtyl*’s innards, saying, “these extra points are not really part of the arc to typeset. They are the phantoms, so don’t compute any for us.” After that, everything else works pretty much the same as regular open-splines in “tyl”-ing. It should be apparent that splines are the largest class of primitive types that *TEXtyl* deals with, since they are capable of simulating/representing most graphical objects that we usually work with.

Beams

The last basic primitive that *TEXtyl* offers is the `beam`. Beams are graphically equivalent to straight line-segments. The only difference is that they have a different font that they use. This font and the particular characters from it are determined by the beam’s two attributes: `staffsize` and `beam-type`. Staff size refers to the spacing of the staff lines of a music score, which also determines the size of the music symbols to use. It is analagous to defining the point size of a character given the interline spacing measure. See [12] for examples of each size, also the discussion of the beam characters on page 29. The `beam-type` attribute refers to the size of notes that the beam connects: regular or grace notes. More will be said about beams in section 6.5.2.

Labels

`Labels` are not really considered primitives, but are included here for convenience and completeness. They have no correspondence with any of the other graphic-primitives described above, but may be included in figure definitions. Labels have the attributes of starting position, font-style, and the string-body of the label. They are not transformable at the local level (but are affected by figure-level transforms), and are basically pretty simple in nature. `TEX` macros are not expanded within the string-body by *TEXtyl*; you should use `TEX` do to any smarter label-typesetting.

Currently, the built-in list of fonts selectable for use in placing labels is:

- `amtt10` (selectable by specifying face 1)
- `amb10` (face 2)

- amsl10 (3)
- amtt8 (4)
- and amsl8 (5)

The characters from these fonts are simply set, as *TeX* does not worry about kerning or ligatures, and spacing is done with information found in the parameter section of the `.tfm` file. This font mapping list is only temporary, and can be changed within the *TeX* source program.

6.3 Procedural Handling

After *TeX* picks apart the `\special` strings from the DVI file, and determines the parameters (both specified and defaulted) for the specified primitive, *TeX* makes a note about where in the file the start of the special occurs, and passes the data to a *handler*. A handler is a procedure that is specific to each primitive (e.g., there is a `linehandle`, and a `ttsplinehandle` procedure) that knows about the internal representation of each primitive, and how to “handle” the data passed to it. We’ll talk more about the internal representation in the next section.

The handlers’ primary responsibility is to take care of primitive-level geometric transformations, and then to decide what to do with the data, depending on the context of the primitive being dealt with. This context is determined by whether the primitive is contained in a figure, or is by itself. If the primitive is *not* contained in a figure (i.e., the `\special` string was not within a `beginfigure` – `endfigure` pair), the handler

1. computes the bounding box for the primitive,
2. transforms the coordinates of the primitive into DVI-space,
3. offsets them by the current position on the page,
4. outputs a DVI `push` command,
5. calls an appropriate procedure to actually typeset the primitive,
6. and finishes with a `pop` command.

We need to transform coordinates from our first-quadrant cartesian space into DVI-space because the lower-level procedures that actually typeset and output the vector characters assume that they are putting the characters onto a page of the DVI file.

If the primitive's definition is part of an enclosing figure definition, we delay output of this primitive until the figure is completely defined. The handler does any simple geometric transformations required, and then simply packs the data into another structure, and sort of "stacks" this new structure (let's call it an *item*) onto a list of items that are contained in this figure. Our reference to "figure" here could also be a sub-figure within an enclosing figure, but for now we'll just deal with primitives in terms of simple, one-level figures. When the figure definition is complete, the `figurehandle` procedure is called. We will get into figures more in the next section, but for now, we'll say that this handler

1. computes the bounding box of all the sub-items of this figure, taking care of necessary transformations,
2. outputs a `push` command,
3. unpacks each sub-item, and transforms its coordinates into DVI-space,
4. calls the item's particular primitive handler with a special flag that says to simply call the appropriate procedure for immediate typesetting without doing any further transforms,
5. and finishes with outputting a `pop` command to the DVI file.

The design of these handlers was to maintain the idea of "information-hiding" such that each handler (primitive- or figure-level) would know only about one level of abstraction below it.

6.4 Figures

Let's look more closely at what we called "items" in the previous section, and how we really represent figures. Up to this point, I have been rather ambiguous when referring to "primitives." In some cases, it meant *graphic-primitives* (like lines and splines); in other cases it meant *the things that*

we can make pictures with (like figures, sub-figures, and graphic-primitives). Well, now the ambiguity gets worse. I'll try to be careful about distinguishing “graphic-primitives” when I mean primitives that are reducible to vector characters, and use “primitives” when I mean graphic-primitives *and* figures. The reason for the difference is that since we have the ability to do recursive sub-figure definitions, a “figure” (as denoted by a `beginfigure`–`endfigure` pair) becomes a building block (just like graphic-primitives) for that enclosing figure’s definition. It is analagous to the programming language concept of $\langle block \rangle$, where (in BNF notation)

```

<figure> ::= <primitive> / <primlist>
<primlist> ::= <beginfigure> <primlist> <endfigure> /
               <primitive> <primlist> / <empty>

```

where $\langle primitive \rangle$ is what we previously referred to as an “item,” $\langle beginfigure \rangle$ is denoted by `\special{tyl beginfigure}`, and $\langle endfigure \rangle$ is denoted by `\special{tyl endfigure}`. Refer back to page 11 for an example of sub-figures.

Items

The “item” data structure contains the attributes for graphic-primitives, as outlined in section 6.2, and also contains attributes for a figure like:

- any figure-level transformation parameters,
- a “name,”
- and the list of items contained in this figure.

As you may surmise, this can yield a linked-list of linked-lists when built to represent a complex nested figure. In our previous discussion of handlers, one mode of their operation was to “pack and stack” the data into an item. Packing is easy enough to understand, but putting this structure in the correct place is tricky. When an item is created and filled with the appropriate parameters in a graphic-primitive’s handler, there is a notion of “context” which refers to the depth of recursion of sub-figures. A depth of 0 means that there is no higher-level figure, a depth of 1 means that any primitives encountered will belong to an outermost level figure, etc. When *TEXtyl* comes across a `beginfigure` special string, it changes the recursive-definition context, and names that figure with that depth number. After that,

any graphic-primitives that are encountered belong to that named figure in that context. An `endfigure` special will decrement the depth of recursion, and thus will change context back to the previous context. If this new context has a depth of 0, then we know we have closed the outermost level of figure definition, and thus we are ready to typeset the entire figure.

Let's look at how items are inserted onto our data structure so that context is maintained. *TeX*tyl uses a procedure called `pushItem` that takes an item and puts it on the correct list (or sub-list) according to the current figure context. Roughly, the algorithm is a simple insert, based on a key of depth-name.

```

Start from the front of the list of this figure
while the list is not empty
  Look at the item
    if it is a figure of the current context
      then we are done searching. do an insert
    otherwise there must be a sub-figure lower on the list
      look for the next item on the list that is a figure
      and start the next search from its list

```

I really simplified the explanation of the insertion scheme here, but the important idea is that of looking for the correctly labelled figure-list and inserting the item there.

Once the structure is built, and *TeX*tyl determines that it is ready to typeset the entire figure, it traverses the structure to compute and note the bounding boxes of the sub-figures (and their bounding boxes, too). We need to do this in case there are any figure-level scaling or rotations of graphic-primitives which are performed relative to the center of the bounding box of the primitives contained in the figure. As we traverse the sub-figures, we do any required sub-figure transforms, record the dimensions of the bounding box, and pop up a level. When our traversal has reached the top level, all the figure's primitives have been transformed according to the recursive definition's specifications (i.e., the transforms are additive as we descend in recursive depth). If there are any top-level transforms to be taken care of, we revisit the lists of primitives, performing the transforms. Finally, at this point we are ready to re-traverse the structure, grab each graphic-primitive, unpack it, transform it into DVI-space, and give control to its handler for immediate typesetting.

If you really want to get picky, the contents of a figure are not typeset and output until the last balancing `endfigure` is encountered. It is at the current position on the page where this `endfigure` is set that the origin of our figure is placed. We can get away with this condition because specials are viewed by T_EX as “boxes” of zero width, and so a consecutive series of `\specials` do not change the current position on the page relative to the first invocation of the list. If, however, the user types in normal T_EX-typesettable text in the middle of a list of `\specials` defining a figure, the current position will move, and our figure will be shifted down to the position where the final `endfigure` is invoked. This might make him unhappy. My advice is to think of the figure/picture as an atomic entity, whose contents is strictly a list of `\special` strings defining that figure.

6.5 The Fonts

The whole basis of this program lies in the ability to typeset graphics with characters. Thus we have two families of fonts: the vector fonts (originally designed at C-MU[11] by Bruce Lucas), and the beam fonts (part of the music fonts for the *MusiCopy* project).

TEXtyl converts the requests for line-drawings into commands to typeset these characters of line segments. See Appendix B for examples of these characters. Since Metafont 79 limits the TFM information to 16 heights and depths, we have to do the computations of the character dimensions ourselves. This is easy to do as *TEXtyl* has the correct dimensions for each character described in Pascal code in such a way that only a couple parameters from the `.tfm` file need to be accessed in order for the code to define the exact (internal) dimensions for a font. In essence, the basic description of the vector and beam fonts are hard-coded in *TEXtyl*, in a manner independent of the actual size of the pens used to draw them. This “on-the-fly” font calculation is done only as needed, and then the font information is cached and noted so that future requests for that font will not require us to re-compute these dimensions.

Within the modified DVI file, *TEXtyl* defines these new fonts and begins numbering from 300, as a sort of “signature.” So if you look at this new DVI file with DVItypex, you can tell the parts that are typeset figures by noting the places in the listing that reference a font numbered above 300.

6.5.1 Vector Setting

The vector fonts themselves, as described in section 5.1, are typeset using the method described by Nelson[11], and I'll describe how $T_{E}X_{t\text{y}l}$ does it. In section 6.3, we made mention that a graphic-primitive's handler call an "appropriate procedure to actually typeset the primitive." This typesetting procedure is actually the "hook" for a programmer to get right to the typesetting code. The procedures look like $Ty1x$, where x is a graphic primitive (e.g., $Ty1Spline$, $Ty1Arc$, etc.), and take as parameters the information equivalent to the unpacked representation of an item. The coordinates of any points used are in DVI-space, and in units of scaled-points. Refer to the code for the formal parameter list of each $Ty1x$ procedure. These hooks call procedures to "lay" line-segments on the page, after doing some work of clamping, and opening the correct font files for the requested line-thickness (thicknesses in the case of $ttsplines$). These procedures, in turn, call $layline$, a procedure that determines the best way to typeset the line-segment. $Layline$ checks to see if the line is strictly horizontal or vertical so that it can reduce the line to typesetting a $T_{E}X$ rule, and then just add endpoints for smooth overlap. However, if the line is more diagonal, we have to go through more work.

Diagonal lines, and certain cases where we think using $T_{E}X$ rules inappropriate, require potentially using the full set of vector characters. The $diagonal$ procedure intersects the line-segment with the origin of the semi-square of the vector font (see also page 28). In essence, it implements a greedy method by scaling down the radius of the semi-square to find the longest possible radius of the vector characters such that the far point of the line-segment lies just outside that radius.

Then $diagonal$ determines the dy and dx from the current position to that intersection point on the line.² We use the mapping function similar to [11] to determine character code corresponding to these dy and dx values:

$$code = 160 + dy + dx - 9 * \max(dx, -dy) \quad \text{for } dy < 0, dx \geq 0$$

$$code = 160 + dy - dx - 7 * \max(dx, dy) \quad \text{for } dy \geq 0, dx \geq 0$$

but this scheme assumes that our font has at least 160 characters (i.e., 256), but most fonts have only 128 characters. We have to re-map this discontinuous set of codes to fit with a range of 0 . . . 127, and in doing so, we have to

²Thus dx and $abs(dy)$ are integer powers of 2 between 0 and 16 inclusive.

lie about two of the characters by pretending that the two longest vertical vectors do not exist, but each is to be replaced by *two* vectors of half that length.

Once we have determined the vector character, all we have to do is move to the current-position, set the character, increment our position, and finish setting the rest of this line-segment. The thickness of the vector theoretically does not matter, as the path followed by the vector character is invariant over the thickness (at least to the resolution of the printer).

6.5.2 Beam Setting

The way that *TEXtyl* typesets beam is similar to typesetting line-segments. The `Ty1Beam` procedure takes care of determining the correct beam character to use. Recall from section 5.1 that there are two lengths for each type of quarternote (i.e., regular notes and grace notes). When *TEXtyl* tries to typeset the beam, it uses the beam character whose length is longest *and* whose angle is closest to that required.

Given a music font of beam characters with their lengths, heights, depths, and angles from horizontal computed, we try to set the whole beam at once by the following method:

1. Compute the dy and dx of the beam, as well as its length and actual angle from horizontal
2. compute the fractional number of characters needed to typeset the beam for both sizes of beam characters (this is $\frac{\text{length}_{\text{beam}}}{\text{length}_{\text{beamchar}}}$)
3. try to bracket a pair of beam characters whose angles are nearest the actual angle of the beam
4. use the beam character that has the smallest angular deviation from that of the actual beam
5. typeset the characters along the line from the left endpoint of the beam to as close to the right endpoint as possible. Then typeset the last character such that the right-hand-side of the character coincides with the right endpoint of the beam.

We only need to select one character to use for typesetting as the slope of the beam is constant over its length, and the character used has a “slope” (angle from horizontal) that is as close as possible to the beam.

This juggling of “best-length” versus “best-slope” is only necessary because of the need to keep the set of characters to a reasonable size, and also because the angles required for beams is not as broad as for the requirements of the more-general vector line-segments.

6.6 Buffering

As *TEXtyl* reads in bytes from the DVI file for subsequent interpretation, it copies them into an internal buffer. At a simple level, this buffer is used to simply copy the DVI bytes used on a DVI page (delimited by `bop` and `eop` bytes). Thus, *TEXtyl* would act simply as a mechanism that copies its input to its output, verbatim. At the level that *we* are interested in, this buffer will hold all the bytes up to, and including a `tyl` special string. On page 38, we made mention of a “note about where the start of the special occurred” in the DVI file. When we read a `special`, we mark the position of the beginning byte of the special, as found in our buffer. If we find that the special is “tyl”-able, we back up the current place in the buffer to the start of the special, and proceed with the tiling. *TEXtyl* will output typesetting command bytes, and effectively over-write the previous `special` string in the buffer, achieving a kind of in-line macro-expansion of the special.

A tricky part of this macro-expansion approach is taking care of defining new fonts that were used during tiling on a page. As *TEXtyl* “tyl”s a figure, it records the used font names (here, only vector or beam fonts) in a list of “fonts-to-be-defined.” At the end of a page, we formally define these as yet undefined fonts within the DVI file, and then mark them as having been DVI-defined to avoid re-definition before the postamble. Even this scheme needs cleverness since we do not know all the fonts used to typeset any primitive until the end of the page, but we have to define the font within the DVI file before the first actual use of the font to typeset those graphic primitives.

The way we get around this “Catch-22”-ish situation is to have an internal flag (a “font-flag”) inserted into the buffer before the first commands to typeset a graphic primitive. Since the DVI format allows us to define fonts between any two DVI commands in a file, we can define them mid-stream.

As *TEXtyl* parses a graphic primitive’s `special` string, it backs up in the buffer to the start of the special, outputs the font-flag and then continues to insert the tiling commands into the buffer until the next special, or the end of page. At the end of a page, all the tyl-able `specials` have been expanded, we have buffer full of DVI commands and font-flags for this page, and a list of fonts ready for initial definition in the DVI file. We are now ready to write the buffer to the actual file. As *TEXtyl* writes each byte of the buffer to the disk file, it first looks at the byte. If it is “normal” (i.e., 0 . . . 255), it converts the byte to the proper system-dependent format (if necessary) and outputs it to the disk file. However, if the byte is our font-flag, it treats this flag as a macro, too, and writes out to the file the formal DVI definitions of the new fonts. *TEXtyl* does this expansion only once per page, and ignores any other font-flags for the rest of the page (well, it actually outputs them as `nops`). After these new fonts have been defined, *TEXtyl* continues to output each byte of the page buffer, and when done, starts processing on the next DVI page for tiling.

6.7 Odds ’n Sods

This section contains down-deep information in *TEXtyl* that really does not fit nicely anywhere else.

As I discussed before, *TEXtyl* is built on top of the DVItypex program[5] and was modified in following ways:

- The `specialcases` and `skippages` procedures were modified to call *TEXtyl*’s `mainhandlespecials` procedure to handle any `\special` strings, instead of throwing those bytes out.
- The `readpostamble` procedure was changed so that we can insert all the font definitions of the new fonts that we used during any tiling. *TEXtyl* simply copies to the internal buffer any font definitions already in the postamble, until it reaches a `postpost`. Here *TEXtyl* backs up one byte (over the `postpost`), outputs all its new fonts, and then inserts the new `postpost`.
- The `main` loop was modified so that at each end-of-page, we write out the internal buffer, clear it, and reset any internal counters for the

page (like figure-depth context, figure-number, and counts of fonts-to-be-defined).

Other Notes

I have implemented an adaptive sub-division method for determining an optimal number of vector characters with which to typeset a splines. The method uses a simple quadrature sub-division on each span of the spline until a linear-distance criterion is met. If we let the number of sub-divisions be N , we can determine the near-optimum number of intervals with which to actually sample the spline when we do the interpolation as 2^N . This is “optimal” in that we use the fewest vectors for long spans, and do not under-sample shorter, tighter spans, and achieve a spline that is very smooth.

The names of the primitives within the `\special` strings need be unique only to the first three letters. Thus `\special{tyl beg}` is equivalent to `\special{tyl beginfigure}`. This can make input easier and shorter.

Also, we can be pretty lenient about how we delimit our keywords and makers. In most of the examples, commas or spaces were used to separate the markers and integers. In actuality, we can use almost anything for keyword and marker delimiting *other* than:

- other alphabetic letters
- other marker characters (like @ and ")
- %, ~, \ and those other reserved characters which might upset T_EX, as it tries to expand macros inside the body of the `special` string before outputting it to the DVI file.

For example, `\special{tyl line *()(!::4... 3** 4&&12[5]}` is lexically equivalent to the nicely typed `\special{tyl line m 4 (3,4)(12,5)}` or the tersely typed `\special{tyl lin m 4 3 4 12 5}`.

Another “signature” of *T_EXtyl* is whenever we output the commands to typeset a line-drawing, we precede the `push` command with *two nops*. This easily identifies the sections that have been “tyl”-ed.

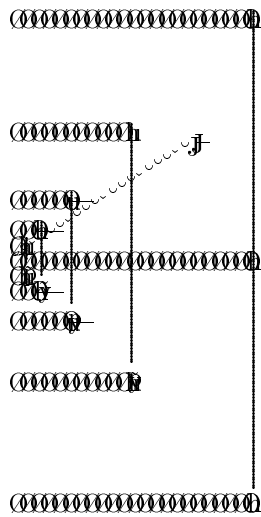


Figure 6.1: Semi-squares of the Vector Fonts

Chapter 7

Future Extensions

Given a little time, one could extend the current set of graphic primitives in *TEXtyl*, such as:

- simpler-appearing macros,
- regular polygons (all easily built upon currently-existing primitives),
- oriented arrows
- line-pattern filled figures,
- a more general method of defining and instancing symbol definitions.

I am still experimenting with two methods of adequately computing the smoothest interpolation of thicknesses along the length of the slur. When the thickness for a particular segment of the curve have been determined, we check to see if that thickness is in a subset of vector-font sizes. If so, we can use it as-is; otherwise, we have to clamp that thickness to fit in the subset. The only problem is that achieving a super-smooth interpolation along the curve's thicknesses might require a large number of vector fonts to be loaded—sometimes stressing the printing device's capabilities. I still have to experiment to find a reasonable subset of these sizes that can be used and still achieve adequate results.

I wrote this program with the intention of ease of extensibility for additional “primitives” to be built on top of the functionality provided by *TEXtyl*. I have also interfaced *TEXtyl* with Dario Giuse's graphic drawing program

DP that he wrote at C-MU[6]. It is smart enough to represent its graphics primitives in files as a list of text-strings, and is now able to output the `\special` strings, specifying the width and height of the created figure. Converting the files of the MacDraw program to yield ASCII strings is potentially possible, too. In either case, such strings are be easily converted to `\special` command strings for *TeX* to work on.

Appendix A

T_EXtyl Summary

Let's summarize the abilities and requirements of *T_EXtyl*. First of all, we have to put an `\input{textyl}` near the start of the document (at least before our first invocation of a `\special{tyl ...}` command). This loads the macros that we need to create space for our figure, and to set up the right environment for *T_EXtyl*.

Once the macros have been loaded, when we want to set a figure (or even a single *T_EXtyl* primitive), we must place the primitives within an environment started by `\begintyl{<vertsize>}[<horizsize>]` and finished by `\endtyl`. Here, the required parameter, `<vertsize>`, determines the vertical size of the box that *T_EXtyl* will create for our figure to be placed in. If we do not need extra space, we put in `0pt` or some similar zero-length dimension. The optional parameter, `<horizsize>`, will add a non-zero width to the box that `\begintyl` creates for us. It is *crucial* that there is *no* space between the `}` and `[` characters of the `\begintyl` invocation if we decide to use the optional horizontal width specification. Otherwise, things may go awry for this figure. Also, we are responsible for any offsets that we may need before the figure is placed. We can achieve this through `\hskip` and `\vskip` calls in T_EX, or `\hspace` and `\vspace` calls in L^AT_EX.

The types of primitives that we may place within the `\begintyl` and `\endtyl` pair are summarized here, without their surrounding `\special{tyl - }`, nor any extra delimiting punctuation (like commas).

`beginfigure` [*<meas>*] [*<xfm>*] [**W** *<wid>* *<ht>*] [**F** *<wid>* *<ht>*] *which opens the environment*
`endfigure` *which closes the environment*

```

line [meas] [xfm] thick [vect] [L style] x1 yb xr yt
spline [meas] [xfm] thick [vect] [L style] [spltype]
    [closure] [X thick] numpts the x-y points
ttspline [meas] [xfm] [vect] [L style] [spltype]
    [closure] [X thick] numpts points thicknesses
arc [meas] [xfm] thick [vect] [L style] radius
    [C centx centy] startAng stopAng
label [meas] face x y " string "
tieslur [meas] minthick maxthick numpts points
beam [meas] staffsize [beamtype] x1 y1 x2 y2

```

where:

meas specifies units of length for this primitive. One of **S**, **P**, or **M** indicates scaled-points, printers-points, and millimeters, respectively. Defaults to printers-points.

xfm specifies a transformation to be applied. It is of the form $T \langle sx \rangle \langle sy \rangle \langle tx \rangle \langle ty \rangle \langle rot \rangle$, where *sx* and *sy* are for scaling, *tx* and *ty* are for translation, and *rot* is for counter-clockwise rotation (in degrees).

thick is the integer thickness of the line. Usually between 1 and 12.

vect is the type of vector font to use. One of **C**, **H**, or **V** indicates circular, horizontal-flat, or vertical-flat pen type. Defaults to circular pen.

style is the line-style to use. One of 0, 1, 2, or 3 indicates a solid, dotted, dashed, or dot-dashed line-style. Defaults to solid line.

spltype is the kind of spline to use through the control points. One of **B**, **D**, **K**, or **I** indicates the B-spline, Cardinal, Catmull-Rom, or Interpolating b-spline basis to use, respectively. Defaults to Catmull-Rom.

closure is one of **O** ("oh") or **U** which indicates that the spline is closed or open, respectively. Defaults to open.

numpts is the integer number of control points for this spline.

radius is the length of the radius in units specified by $\langle meas \rangle$.

centx and **centy** are the center-point coordinates of the arc/circle.

startAng and **stopAng** are the initial and final angles, respectively, for the arc as measured in degrees counter-clockwise from the positive x-axis. If they are the same number, a closed circle is indicated.

face refers to the style of typeface for the label. See page 37 for the list.

string is the string-body of the label to typeset at the specified position.

staffsize refers to the staff note-size for beams. It is between 0 and 8, inclusive.

beamtype is the type of beam to use. One of **G** or **R** indicates grace-note size, or regular size beams, respectively. Defaults to regular.

wid and **ht** refer to the width and height of a figure, in units specified by $\langle meas \rangle$. Mainly used in conjunction with **F** (“Fitting” size operator) and **W** (“Written” size marker).

Appendix B

Font Example

Example of cvec3

	'0	'1	'2	'3	'4	'5	'6	'7	
'00x	Γ	Δ	Θ	Λ	Ξ	Π	Σ	Υ	"0x
'01x	Φ	Ψ	Ω	ff	fi	fl	ffi	ffl	
'02x	ı	ı	`	'	˘	˙	-	°	"1x
'03x	˘	ß	æ	œ	ø	Æ	Œ	Ø	
'04x	˘	!	"	#	\$	%	&	'	"2x
'05x	()	*	+	,	-	.	/	
'06x	0	1	2	3	4	5	6	7	"3x
'07x	8	9	:	;	i	=	ı	?	
'10x	@	A	B	C	D	E	F	G	"4x
'11x	H	I	J	K	L	M	N	O	
'12x	P	Q	R	S	T	U	V	W	"5x
'13x	X	Y	Z	["]	^	·	
'14x	'	a	b	c	d	e	f	g	"6x
'15x	h	i	j	k	l	m	n	o	
'16x	p	q	r	s	t	u	v	w	"7x
'17x	x	y	z	-	—	"	~	¨	
	"8	"9	"A	"B	"C	"D	"E	"F	

Appendix C

Macros and Extended Examples

All of the thumbnail figures in this manual were created using the `\special{tyl...}` strings as described, and were placed within an environment delimited by `\begintyl` and `\endtyl`. This environment allows the user to specify a vertical and optional horizontal offset from the “current-position” on the page. This determines where the origin of the user’s coordinate system is to be placed. Also, this environment sets parameters so that the user can use tabbing and spacing to align the `\special` strings as he desires (e.g., to better show the nesting of figures and graphic-primitives). The macros used are defined below:

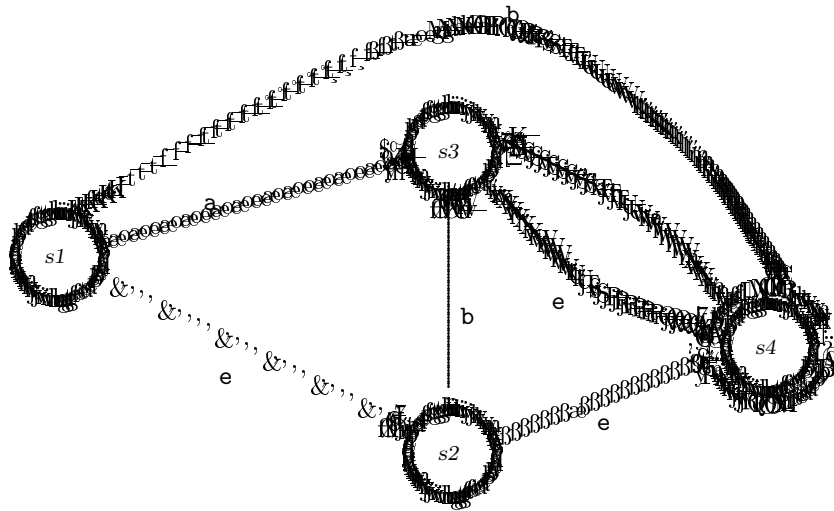
```
% This is textyl.tex
%
% macros for figures for Tyling
% specify :
% \begintyl{vert_dimen}[optional_horiz_dimen]
%   (with NO intervening space between the } and the [ chars)
% then the \special{tyl ...} strings
% then you must finish environment with \endtyl
%
\def\begintyl#1{\begingroup\endlinechar=-1\catcode '\^^I=9
\dimen120=#1
\ifhmode\toks0={\futurelet\tyltempb\begHtyl}
\let\endtyl=\endHtyl
\else\toks0={\futurelet\tyltempb\begVtyl}
\let\endtyl=\endVtyl\fi
\the\toks0}
```

```

\def\begHtyl{\ifx [\tyltempb \let\tylh=\tylHspace
\else\let\tylh=\tylHnospace\fi\tylh}
%
\def\tylHspace[#1]{\setbox0=\hbox to #1\bgroup
\box to \dimen120\bgroup\vss}
\def\tylHnospace{\setbox0=\hbox\bgroup
\box to \dimen120\bgroup\vss}
%
\def\begVtyl{\ifx [\tyltempb \let\tylv=\tylVspace
\else\let\tylv=\tylVnospace\fi\tylv}
%
\def\tylVspace[#1]{\setbox0=\vbox to \dimen120\bgroup\vss
\hbox to #1\bgroup}
\def\tylVnospace{\setbox0=\vbox to \dimen120\bgroup\vss
\hbox\bgroup}
%
\def\endHtyl{\egroup\hss\egroup\box0\endgroup}
\def\endVtyl{\hss\egroup\egroup\box0\endgroup}

```

Let's look at a non-trivial example of frequently-used figures. The original figure is from [2], page 325. I'll give the diagram, and then the annotated `\special` commands.



And the text to produce this is:

```
\begintyl{9cm}
\special{tyl begin /AHU=fig=9.7/}
  \special{tyl arc m 2 6 @ 10 46 10 10 /state=s1/}
  \special{tyl label m 5 9 46 "s1"}
  \special{tyl arc m 2 6 @ 62 20 10 10 /state=s2/}
  \special{tyl lab m 5 61 20 "s2"}
  \special{tyl arc m 2 6 @ 62 60 10 10 /state=s3/}
  \special{tyl lab m 5 61 60 "s3"}
  \special{tyl arc m 2 6 @ 104 34 10 10 /finalstate=s4/}
  \special{tyl arc m 2 8 @ 104 34 10 10}
  \special{tyl lab m 5 103 34 "s4"}
%now draw the connectors
  \special{tyl line m 3 16 48 56 59 /s1-s2/}
  \special{tyl lab m 1 30 53 "a"}
    \special{tyl line m 3 56 59 53 60}% and the arrow
    \special{tyl line m 3 56 59 54 57}
    \special{tyl line m 3 53 60 54 57}

  \special{tyl line m 3 15 43 56 23 /s1-s3/}
  \special{tyl lab m 1 32 30 "e"}
    \special{tyl line m 3 56 23 55 25}
    \special{tyl line m 3 56 23 53 23}
    \special{tyl line m 3 55 25 53 23}

  \special{tyl spline m 3 4 12 52; 62 76; 90,63; 105,44 /s1-s4/}
  \special{tyl lab m 1 70 78 "b"}
    \special{tyl line m 3 105,42 106 44}
    \special{tyl line m 3 105 42 104 44}
    \special{tyl line m 3 104 44 106 44}

  \special{tyl line m 3 62 26 62 54 /s3-s2/}
  \special{tyl lab m 1 64 38 "b"}
    \special{tyl line m 3 62 54 64 52}
    \special{tyl line m 3 62 54 60 52}
    \special{tyl line m 3 60 52 64 52}

  \special{tyl spline m 3 3 67 55; 80 42; 96 36 /s3-s4/}
  \special{tyl lab m 1 76 40 "e"}
    \special{tyl line m 3 96 36 95 38}
    \special{tyl line m 3 96 36 95 35}
    \special{tyl line m 3 95 35 95 38}
```

```
\special{tyl spline m 3 3 69 61; 86 52; 98 40 /s4-s3/}
\special{tyl lab m 1 89 51 "a"}
  \special{tyl line m 3 69 61 71 62}
  \special{tyl line m 3 69 61 70 59}
  \special{tyl line m 3 70 59 71 62}

\special{tyl line m 3 69 22 96 32 /s2-s4/}
\special{tyl lab m 1 82 24 "e"}
  \special{tyl line m 3 96 32, 94 33}
  \special{tyl line m 3 96 32 95 30}
  \special{tyl line m 3 94 33 95 30}
\special{tyl endfigure}
\endtyl\par
```

Bibliography

- [1] Adobe Systems Inc., *PostScript Language Reference Manual*. Addison-Wesley, Reading, MA, 1985.
- [2] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass. 1974.
- [3] Brian A. Barsky, and Donald P. Greenberg, “Determining a Set of B-spline Control Vertices to Generate an Interpolating Surface.” *Computer Graphics and Image Processing*. **14** 3, (Nov. 1980), pp.203–226.
- [4] Edwin E. Catmull, and Raphael J. Rom, “A Class of Local Interpolating Splines.” *Computer Aided Geometric Design*. Robert E. Barnhill and Richard F. Riesenfeld, eds. Academic Press, New York, NY, 1974, pp. 317–326.
- [5] David R. Fuchs, Howard Trickey, *The DVI type processor*. Program documentation, 1983–4.
- [6] Dario Giuse, *DP—Format of the drawing files*. Technical Report. Carnegie -Mellon University Robotics Institute, 1983.
- [7] John S. Gourlay, “A Language for Music Printing.” *Communications of the ACM* **29** 5, (May 1986), pp. 388–401.
- [8] Norm Hutchinson, et al. DVISUN Program. 1983.
- [9] Donald E. Knuth, *The T_EXbook*. Addison-Wesley, Reading, Mass. 1984.
- [10] Donald E. Knuth, *T_EX and METAFONT: New Directions in Typesetting*. Digital Press, Bedford, MA, 1979.

- [11] Bruce Nelson, and James Saxe, “Drawing Splines with a Vector Font.” Unpublished technical report. Carnegie-Mellon University, 1980.
- [12] Ted Ross, *The Art of Music Engraving and Processing*. Hansen Books, 1970.
- [13] Daniel Berkeley Updike, *Printing Types: Their History, Forms, and Use*. Harvard University Press, Cambridge, MA, 1937.
- [14] Sheng-Chuan Wu, John F. Abel, and Donald Greenberg, “An Interactive Computer Graphics Approach to Surface Representation.” *Communications of the ACM* **20** 10, (Oct. 1977), pp. 703–712.
- [15] Christopher J. Van Wyk, *A Language for Typesetting Graphics*. Ph.D. dissertation. Stanford University, 1980.
- [16] Hermann Zapf, *The Printing Salesman’s Herald*. Book 39. Champion Papers, New York, NY, 1978.

Index

- "-marker 19, 48
 - .dvi file 3
 - .tex text file 1, 3, 34
 - .tlog file 4, 23
 - .tyl file 23
 - ?! flag 24
 - ? flag 23
 - @-marker 8, 19
- A**
- arc 36
 - arc** 19
 - arcs 7
 - angles 8, 19
 - attributes 36
 - center 8
 - centering 8, 19
 - circles 8, 36
 - closed splines 36
 - control points 31, 36
 - ellipses 36
 - example 8
 - open 36
 - open spline 36
 - phantoming 37
 - radius 8
 - representation 31, 36
- B**
- B-marker 6, 14, 20
 - B-spline basis 6, 20
 - B-splines
 - interpolating-type 6
 - recomputing points 30
- beam** 22
- beam** 37
 - beam-types 22
 - beam font 29
 - constraints 30
 - design 29
 - difference from vector font 29
 - lengths 30
 - range of angles 30
 - relation to staff size 29
 - sizes 29
 - types 29
 - beams 16
 - attributes 37
 - default size 22
 - difference from lines 37
 - sizes 22
 - staff size 22
 - typesetting 45
 - beginfigure** 11, 22
 - begintyl** 1, 3, 4, 12, 57
 - bitmaps *vi*
 - blank space 17
 - bounding box 34, 38, 41
 - boxes 1, 3, 9
- C**
- C-marker 20

- capabilities *v*, *vi*, 17
 - Cardinal basis 6, 20
 - cartesian-space 2
 - Catmull-Rom basis 20
 - Catmull-Rom splines 5, 30
 - circles 7, 8, 36
 - control points 31, 36
 - relation to arcs 36
 - representation 31
 - circular pen 20
 - clamping 35, 43, 49
 - closed splines 14, 20
 - closure 20
 - combining primitives 9, 10
 - coordinates 2
 - origin 2
 - coordinate space system 19
 - current position 1, 2, 3, 17
 - curves 5
- D**
- D-marker 6, 20
 - defaulting values 5
 - defining fonts 46
 - delimiters 6, 48
 - design decisions 21, 27, 30
 - design goals *v*
 - DVI *v*
 - DVI-filters 4
 - DVI-space 35, 38, 43
 - DVI file 27
 - previewing 33
 - DVI format 46
 - DVI position coordinates 27
 - DVISUN 33
 - DVItype 27, 47
- E**
- ellipses 19, 31, 36
 - relation to circles 36
 - endfigure** 42
 - endfigure** 11, 22
 - endtyl** 1, 12, 57
 - errors
 - internal 24
 - extensions 49
- F**
- F-marker 22
 - figure definitions 22
 - figures 10
 - context 41
 - defining 11
 - example 12
 - examples 11
 - fitting to size 22
 - nested symbols 11
 - preparation 2
 - sub-figures 11
 - transforms 22
 - transforms 10, 22
 - fonts
 - computing dimensions of 42
 - defining 46
 - font flag 46
 - for labels 38
 - numbering 43
 - to be defined 46
- G**
- G-marker 16, 22
 - geometric transforms 21
 - Giuse, Dario 28, 49
 - grace-notes 16
 - graphic editors 17, 22
 - grid-origin 3

- H
- H-marker 20
 - handlers 38
 - design idea 39
 - figurehandle 39
 - functionality 38
 - linehandle 38
 - ttsplinehandle 38
 - handling specials 27
 - horizontal pen 20
- I
- I-marker 6, 20
 - Ideal *v*
 - input** 1
 - interpolating B-spline 6
 - basis 20
 - item 39, 40
 - inserting 41
 - packing 40
 - traversing 41
- J
- jaggies 36
- K
- K-marker 5, 20
 - keywords 6, 48
 - Knuth, Donald 1
- L
- L-marker 20
 - label** 37
 - label** 19
 - labels
 - face 19
 - font face-styles 38
 - font style 19
 - strings 19
 - L^AT_EX** 1, 3, 17, 51
 - figure** environment 17
 - layline** 43
 - leaving no space 3
 - leaving white space 3
 - lies 37, 44
 - line** 35
 - line** 18
 - line example 1
 - line segments 18, 35
 - line style 20, 34
 - dashed 20
 - dot-dashed 20
 - dotted 20
 - solid 20
 - line styles
 - examples 20
 - line thickness 2, 18, 20, 31, 43
 - example 20
 - local maintainer 4, 25
 - Lucas, Bruce 42
- M
- M-marker 5, 6, 10
 - M-measure 19
 - macro-expansion 33, 46
 - macros 1, 9, 57
 - see* **begintyl** 1
 - markers 6
 - measurement 19
 - default 19
 - default 22
 - units 2
 - Metafont 28, 42
 - limitations 28
 - millimeters 19
 - measurement 5
 - music *v*, 21
 - music font 29
 - MusiCopy *v*, 15, 42

- N Nelson, Bruce 28
 nested symbols 11
- O 0-marker 14, 20
 open spline 14, 20
 optional values 5
 origin 2
 origin of quadrant 17
 OSU 14
 output device 4, 49
 font memory 36
 oval
 see ellipse 19
- P P-measure 19
 parameters to *TEXtyl* 17
 Pascal 33
 pens 20
 phantom control points 37
 phantoming 37
 pixel thickness 20
 pop 39
 postamble 47
 PostScript *vi*
 preamble 1
 preparing a figure 2
 primitive 2
 primitives 17, 34
 attributes 34
 built-up 17
 combining 9, 10
 definition of 40
 invoking 9
 modifying attributes 34
 names 48
 parameters 9, 18
 reduction of 34
 scaling; rotating; translating 9
 printer's points 19
 measurement 2
 programmer's hook 43
 programs making figures 22
 push 39
- R R-marker 16, 22
 reference point 17
 rotation 9, 10
 using degrees measurement 10
 running *TEXtyl* 3
- S S-measure 19
 Saxe, James 28
 scaled points 19
 scaling 9
 slurs 15, 21, 30, 31
 see also ties 16
 space 1, 3, 51
 space macros 1
 special 38
special 2
 specials 2, 17, 27, 46, 57
 case insensitivity 6
 handling 33
 handling by *TEXtyl* 27
 punctuation 6
 typing in 6
spline 35
spline 18
 spline
 B-spline 6
 Cardinal 6
 Catmull-Rom 5
 control-points 5
 coordinates 5

- default 5
 - example 5
 - Interpolating B-spline 6
 - ttspline 6
 - spline curves 5
 - splines
 - attributes 35
 - B-spline 35
 - example 15
 - basis 20, 35
 - Cardinal 35
 - example 14
 - Catmull-Rom 35
 - example 14
 - clamping 35
 - closed 14
 - closure 20
 - default basis 20
 - default closure 20
 - default type 30
 - families 30
 - Interpolating B-spline
 - example 15
 - interpolation 30
 - inversion 30
 - open 14
 - types 20
 - typesetting 47
 - use in simulating 37
 - staff-size 16
 - staff lines 37
 - staff sizes 29, 37
 - Standard, Paul *i*
 - sub-figures 11
 - subdivision of splines 47
 - surprise errors 24
 - symbol
 - see* figure 22
- T**
- T-marker 10, 12
 - TEX *v*, 1, 3, 17, 34, 51
 - TEXbook 9
 - TEX commands 17, 51
 - TEXtyl *v*
 - .tlog log-file 23
 - error handling 23
 - error messages 24
 - error recovering 23
 - how to read manual *vi*
 - name *v*
 - parameters 17
 - serious problems 26
 - signatures 43, 48
 - simple problem 25
 - textyl.tex file 1
 - TFM information 27
 - Thick-n-thin spline 6
 - ties 15, 21, 30
 - and ttsplines 16
 - example 16
 - see also* slurs 16
 - tieslur** 21
 - tieslur 35
 - tieslur
 - attributes 35
 - clamping 35
 - example 16
 - implementation 31
 - pen type 21
 - relation to ttsplines 35
 - representation 34
 - thicknesses 16, 21
 - vector type 21
 - transforms

- figure-level 22
- transform** 21
- transformations
 - see* transforms 21
- transforms 9
 - additiveness 22, 42
 - concatenation 11
 - example 10
 - figure-level 41
 - mirroring 10
 - no-op 10
 - parameters 10, 21
 - format 21
 - requirement 21
 - rotating 9
 - scaling 9
 - sub-figures 11
 - top-level 42
 - translation 9
- transform to DVI space 39
- translating 9
- ttspline** 35
- ttspline 6, 31
 - clamping 35
 - defaults 6
 - example 6
 - line thicknesses 31
 - thicknesses 6, 18
- ttspline** 18
- tyling *v*, 33, 37, 46
- ty1** name-string 2
- typing in special strings 6

U

- U-marker 14, 20
- units of measure 19
- Unix 33
- user's view 17

- user intuition 21

V

- V-marker 20
- vector-types 20
 - default 20
- vector font 20, 28
 - computing dimensions 28
 - cvec 29
 - dimensions 28
 - mapping function 44
 - path invariance 45
 - pen-types 29
 - pens
 - example 29
 - pixelsize 29
 - range of angles 28
 - sizes 29
 - types 29
- vectors
 - typesetting with 28
- vertical pen 20

W

- W-marker 22

X

- X-marker 18

Z

- Zapf, Hermann *i*