SCHEME GRAPHICS REFERENCE MANUAL

Pee-Hong Chen, Wen-Ying Chi, Eric M. Ost
L. David Sabbagh and Geroge Springer

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 145

SCHEME GRAPHICS REFERENCE MANUAL

by

Pee-Hong Chen, Wen-Ying Chi, Eric M. Ost
L. David Sabbagh and George Springer

August, 1983

SCHEME GRAPHICS REFERENCE MANUAL

by

Pee-Hong Chen

Wen-Ying Chi

Eric M. Ost

L. David Sabbagh

George Springer

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, IN 47405

§1 Preface

Traditionally, tools used in computer graphics have included the well-known imperative style languages such as Basic, Fortran, and Pascal. An advantage in using these languages to program graphics applications is certainly in their popularity; most programmers use them. A disadvantage in using them, however, is that the functional nature of graphics is generally not reflected in the programs. The implication is that functional languages may be alternatives to such applications.

In this report, we present several sets of primitive functions essential to most graphics applications. The programming language Scheme is chosen as our underlying tool. Based upon this language, we develop a system of graphics primitives which will be described below. The file containing the source functions is "/usiu/lds/Sgraphics/sgp.s".

Scheme is an applicative order, block structured, tail recursive, lexically scoped dialect of Lisp. It was developed during the mid-70's at MIT. We use ⟦Scheme 311⟧([Fe83]), the Indiana University implementation, to realize our system. We start by presenting some basic and useful graphics primitives. A package of primitive linear algebra operators is then introduced. Finally, we demonstrate a few functions of turtle graphics which are applications of the first two sets.

§2 Basic Graphics Primitives

The basic graphics primitives can be categorized into five classes which specify, respectively,

— Screen attributes,

— Foreground and background colors,

— Cursor moves,

— Line drawings, and

— Circle and arc drawings.

In general, two coordinate systems exist simultaneously. We maintain a cartesian coordinate system and a polar coordinate system at the same time. The origin and the current position (henceforth abbreviated CP) common to both coordinate systems are of special importance to the primitives as well as most applications. Changes of cursor position can be done in one of the three ways: (1) absolute mode, (2) relative mode, and (3) offset mode. The last two are relative to the CP. The following graphics primitives are based upon the features of the GIGI Regis commands ([GIGI]) but can be implemented on other systems.

2.1 Screen Attributes

Screen attributes to be set up include graphics mode intitalization and termination, screen erase, writing features, *etc.* The following summarizes what is available.

▸ **`(initpl)`**

Initializes graphics mode. No arguments needed.

▸ **`(termpl)`**

Terminates graphics mode. No arguments needed.

▸ **`(topdisplay)`**

Sets the screen attribute so that the command line will be displayed on top of the screen in an interactive environment. No arguments needed.

▸ **`(botdisplay)`**

Sets the screen attribute so that the command line will be displayed on bottom of the screen in an interactive environment. No arguments needed.

▸ **`(nodisplay)`**

Sets the screen attribute so that the command line will not be displayed on the screen in an interactive environment. No arguments needed.

▸ **`(erase)`**

Erases the screen in graphics mode. No arguments needed.

▸ **`(clear)`**

Erases the screen at Scheme top level. No arguments needed.

▸ **`(pause`** *sec***`)`**

Pauses the screen operation for *sec* seconds.

▸ **`(invis)`**

Sets the screen attribute so that the cursor becomes invisible. This is the opposite of **`vis`**. No arguments needed.

▸ **`(vis)`**

Resets the screen attribute to normal (cursor visible). This is the opposite of **`invis`**. No arguments needed.

▸ **`(blink)`**

Sets the screen attribute to yield a blinking cursor which is the opposite of **`offblink`**. No arguments need.

▸ **`(!blink)`**

Returns the screen attribute of cursor blinking. Unlike **`blink`**, **`!blink`** causes no global side effects on screen attributes. However, it can be called by **`gprinc`** and result in some local side effects.

▸ **`(offblink)`**

Resets the cursor to normal (no blinking). No arguments needed.

▸ **`(!offblink)`**

Returns the screen attribute of cursor normal.

▸ **`(shade)`**

Initiates shading on the screen. The opposite is **`offshade`**.

▸ **`(!shade)`**

Returns the screen attribute of shading. Causes no global side effects unless intended by the invoker.

▸ **`(offshade)`**

Resets the screen to normal (no shading). The opposite is **shade**.

▸ **(!offshade)**

Resets the screen attribute of shading off. Causes no global side effects unless requested by the invoker.

▸ **(dash)**

Sets writing attribute to dashed lines. This is the opposite of **offdash**. The default writing multiple for spacing is 8 pixels. No arguments needed.

▸ **(offdash)**

Resets line drawing to normal (solid lines). This is the opposite of **dash**. No arguments needed.

▸ **(erasewrite)**

Erases existing images. This is the opposite of **overlaywrite**. No arguments needed.

▸ **(overlaywrite)**

Overlays new images on top of images already on the screen. No arguments needed.

▸ **(writemultiple** *n***)**

Sets writing offset multiple. The default offset multiple in GIGI is 1 which refers to 10 pixels.

2.2 Color Settings

▸ **(foreground** *color***)**

Sets the screen foreground color. In GIGI, *color* must be one of the digits between 0 and 7 which refer to *black, blue, red, magenta, green, cyan, yellow,* and *white,* respectively.

▸ **(background** *color***)**

Sets the screen background color. In GIGI, the options for *color* are the same as described above.

2.3 Text Settings

Text can be printed on the screen in different sizes and directions. We present the following text primitives.

▸ **(textsize** *s***)**

Sets text character size to s. In GIGI, s must satisfy $0 \leq s \leq 16$ and its default value is 1. With the exception of size 0, GIGI generates the character in an area of $9 \cdot s \times 10 \cdot s$ pixels.

▸ **(textdir** θ**)**

Sets the text character direction to θ degrees. In GIGI, θ must satisfy $-360° \leq \theta \leq 360°$ relative to the horizontal axis; its default value is 90° (no tilt).

▸ **(texthgt *h*)**

Sets text height to h. In GIGI, h must satisfy $1 \leq h \leq 16$ which gives the character height of $10 \cdot h$ pixels.

▸ **(text s_1 s_2 ... s_n)**

Prints the text strings s_1, s_2, ..., s_n on the screen starting from the CP. The CP is moved to the position of the last letter of the last string written. The number of arguments **text** takes is arbritrary.

2.4 Cursor Moves

The cursor can be moved to (1) an absolute position, (2) a position relative to the CP, or (3) a position whose distance and direction from the CP is designated by an offset code known to the system. Two coordinate systems are available, namely, cartesian coordinates and polar coordinates. We present the following primitives.

▸ **(push)**

Pushes the CP onto a system stack which can be recovered by **pop**. No arguments needed.

▸ **(pop)**

Pops the last CP from a system stack. No arguments needed.

▸ **(mid)**

Moves the cursor to the center of the screen. No arguments needed.

▸ **(hide)**

Moves the cursor to infinity relative to screen boundaries.

▸ **(moveabs2 *x* *y*)**

Moves the cursor to the point $[x,\ y]$ which becomes the new CP.

▸ **(moverel2 Δx Δy)**

Moves the cursor to the point $[x_{cur} + \Delta x,\ y_{cur} + \Delta y]$ which becomes the new CP, where $[x_{cur},\ y_{cur}]$ is the old CP.

▸ **(Pmoveabs2 r θ)**

Moves the cursor to the point $[r \cdot \cos\theta,\ r \cdot \sin\theta]$ which becomes the new CP.

▸ **(Pmoverel2 r θ)**

Moves the cursor to the point $[x_{cur} + r \cdot \cos\theta,\ y_{cur} + r \cdot \sin\theta]$ which becomes the new CP, where $[x_{cur},\ y_{cur}]$ is the old CP.

▸ **(moveabs *P*)**

Moves the cursor to the point P which becomes the new CP, where P is represented as a list $[x,\ y]$.

▸ **(moverel *P*)**

Moves the cursor to the point $[x_{cur} + x,\ y_{cur} + y]$ which becomes the new CP, where $[x_{cur},\ y_{cur}]$ is the old CP and $[x,\ y]$ are the coordinates for P.

▸ **(movedir** D**)**

> Moves the cursor to a point which has an offset of D relative to the CP. This point becomes the new CP. In GIGI, D must be one of the digits 0 through 7 which refer to the direction. East is 0, northeast is 1, north is 2, northwest is 3, west is 4, , southwest is 5, south is 6, and southeast is 7. The default moving distance is 10 pixels (which can be changed by **writemultiple**).

2.5 Line Drawing

Lines can be drawn either in the cartesian coordinate system or the polar coordinate system. We present the following primitives.

▸ **(lineabs2** x y**)**

> Draws a line from the CP to the point [x, y] which becomes the new CP.

▸ **(linerel2** Δx Δy**)**

> Draws a line from, [x_{cur}, y_{cur}], the CP, to the point [$x_{cur} + \Delta x$, $y_{cur} + \Delta y$] which becomes the new CP.

▸ **(Plineabs2** r θ**)**

> Draws a line from the CP to the point [$r \cdot \cos\theta$, $r \cdot \sin\theta$] which becomes the new CP.

▸ **(Plinerel2** r θ**)**

> Draws a line from [x_{cur}, y_{cur}], the CP, to the point [$x_{cur} + r \cdot \cos\theta$, $y_{cur} + r \cdot \sin\theta$] which becomes the new CP.

▸ **(lineabs** P**)**

> Draws a line from the CP to the point P which becomes the new CP, where P is represented as a 2D list [x, y].

▸ **(linerel** P**)**

> Draws a line from [x_{cur}, y_{cur}], the CP, to the point [$x_{cur}+ x$, $y_{cur}+ y$] which becomes the new CP, where [x, y] is the representation of P.

▸ **(linedir** D**)**

> Draws a line from the CP to a point which has an offset D relative to the CP. This point becomes the new CP. In GIGI, the options of D are described in **movedir**.

▸ **(Sline2** E**)**

> Draws a solid line between two endpoints designated by E which is a list of two points (P_1 P_2) and both points are a list of two coordinates. The CP is moved to P_2 after the drawing.

▸ **(Dline2** E**)**

> Draws a dashed line between two endpoints designated by E which is a list of two points (P_1 P_2) and both points are a list of two coordinates. The CP is moved to P_2 after the drawing.

▸ **(line2** E F**)**

Draws a solid or dashed line accoring to the boolean value of F. If F is true, `(Sline2` E`)` is invoked otherwise `(Dline2` E`)` is invoked. Refer to `Sline2` and `Dline2` for other details.

2.6 Point Plotting

Point plotting is rarely used. However, two primitives are provided here.

▸ `(pointabs2` x y`)`

Plots a point at `[`x`,` y`]` which becomes the new CP.

▸ `(pointrel2` Δx Δy`)`

Plots a point at `[`$x_{cur}+\Delta x$`,` $y_{cur}+\Delta y$`]` which becomes the new CP, where `[`x_{cur}`,` y_{cur}`]` is the old CP.

2.7 Circle Drawing

Circles and arcs can be drawn in both coordinate systems. We have:

▸ `(circabs2` x y r`)`

Draws a circle with center at `[`x`,` y`]` and radius r. The CP is moved to `[`x`,` y`]`.

▸ `(circrel2` r`)`

Draws a circle with center at the CP and radius r. The CP remains the same.

▸ `(Carc` Δx Δy ϕ`)`

Draws an arc of length ϕ. The arc's center is at `[`x_{cur}`,` y_{cur}`]`, the CP, and its starting point is `[`$x_{cur}+\Delta x$`,` $y_{cur}+\Delta y$`]`. If $\phi > 0$, the arc goes counterclockwise, otherwise it goes clockwise. The CP is unchanged.

▸ `(PCarc` r θ ϕ`)`

Draws an arc of length ϕ. The arc's center is at `[`x_{cur}`,` y_{cur}`]`, the CP, and its starting point is `[`$x_{cur} + r \cdot \cos\theta$`,` $y_{cur} + r \cdot \sin\theta$`]`. If $\phi > 0$, the arc goes counterclockwise, otherwise it goes clockwise. The CP is unchanged.

§3 Linear Algebra Primitives

The basic data types in linear algebra are *vectors* and *matrices*. In Scheme, we use the keywords **vector** and **matrix** to create data of these two types. They are both aliases for the system macro **list**. So

```
(define V (vector 1 2 3 4))
```

binds the vector `[1 2 3 4]` to the identifier V. Note that the number of arguments that `vector` takes is arbitrary. Similarly,

```
(define M (matrix (vector 1 2) (vector 3 4)))
```

declares M as the 2×2 matrix $\begin{pmatrix}1 & 2\\ 3 & 4\end{pmatrix}$. Notice that we declare a matrix by specifying its row vectors. The fact that all row vectors must be of the same dimension is an implicit protocol. Our system does not complain about inconsistent row vector dimensions in a matrix declaration.

We now examine the primitives of linear algebra. As a convention, operators dealing with vectors are prefixed with an asterisk (`*`) and operators for matrices are prefixed with an exclamation mark (`!`). Incidentally, the component accessing functions are prefixed with an and sign (`&`).

3.1 Component Accessing Primitives

▸ `(&comp` *OBJ* *i*`)`

Returns the i^{th} component of *OBJ*. For example, if *OBJ* is the vector `[`x_1 x_2 ... x_n`]`, then `(&comp` *OBJ* *i*`)` returns x_i if $1 \leq i \leq n$.

▸ `(&ext` *OBJ* *i*`)`

Returns a list containing the first i components of *OBJ*. For example, if *OBJ* is the vector `[`x_1 x_2 ... x_n`]`, then `(&ext` *OBJ* *i*`]` returns the vector `[`x_1 ... x_i`]` if $1 \leq i \leq n$.

3.2 Vector Manipulation Operators

In this section, let U be the vector `[`u_1 u_2 ... u_n`]` and let V be the vector `[`v_1 v_2 ... v_n`]`.

▸ `(*id+` *n*`)`

Returns the zero (additive identity) vector of dimension n. For example, `(*id+ 3)` returns the 3D zero vector `[0 0 0]`.

▸ `(*unth` *n* *i*`)`

Returns a unit vector of dimention n with 1 in the i^{th} place. For example, `(*unth 3 1)` returns the unit vector `[1 0 0]`.

▸ `(*dim` *U*`)`

Returns the dimension of U. That is, `(*dim` *U*`)` returns the scalar n.

▸ `(*length` *U*`)`

Returns the length of the vector U. So `(*length` *U*`)` gives

$$\sqrt{{u_1}^2 + {u_2}^2 + \ldots + {u_n}^2}.$$

▸ `(*tag` *U*`)`

Returns a function of one argument that when given a value h returns U appended with the vector `[`h`]`, i.e. `((*tag` *U*`)` *h*`)` returns the vector `[`u_1 u_2 ... u_n h`]`.

▸ `(*scp` *U*`)`

Returns a function of one argument which when given a scalar s returns a vector by multiplying each component of V by a factor of s. In other words, `((*scp` *U*`)` *s*`)` returns the vector `[`su_1 su_2 ... su_n`]`.

▸ **(*opp** U**)**

Returns the opposite of U. That is, **(*opp** U**)** returns the vector $[-u_1 \; -u_2 \; \ldots \; -u_n]$.

▸ **(*unit** U**)**

Returns a unit vector in the direction of U. More precisely, **(*unit** U**)** returns the vector $[\frac{u_1}{|U|} \; \frac{u_2}{|U|} \; \ldots \; \frac{u_n}{|U|}]$, where

$$|U| = \sqrt{{u_1}^2 + {u_2}^2 + \ldots + {u_n}^2}, \text{ the length of } U.$$

▸ **(*add** U V**)**

Returns the sum vector of U and V, $U + V$. That is, **(*add** U V**)** returns the vector $[u_1 + v_1 \; u_2 + v_2 \; \ldots \; u_n + v_n]$.

▸ **(*diff** U V**)**

Returns the difference vector of U and V, $U - V$. In other words, **(*diff** U V**)** returns the vector $[u_1 - v_1 \; u_2 - v_2 \; \ldots \; u_n - v_n]$.

▸ **(*edgev** E**)**

Returns a vector from E's starting point to its ending point, where E is a list of two vectors (or points). Suppose E is bound to the list (U V), then **(*edgev** E**)** returns the vector $V - U$, $[v_1 - u_1 \; v_2 - u_2 \; \ldots \; v_n - u_n]$.

▸ **(*dot** U V**)**

Returns the dot (inner) product of U and V, $U \cdot V$. Thus **(*dot** U V**)** returns the scalar $u_1 v_1 + u_2 v_2 + \ldots + u_n v_n$.

▸ **(*cross** U V**)**

Returns the cross (outer) product of U and V, $U \times V$ (instead of $V \times U$). In this implementation, it is restricted to 3D. Let $W = U_3 \times V_3 = [w_1 \; w_2 \; w_3]$. We have $w_i =$ the determinant of the 2×2 matrix $\begin{pmatrix} u_{1+i \bmod 3} & u_{1+(i+1) \bmod 3} \\ v_{1+i \bmod 3} & v_{1+(i+1) \bmod 3} \end{pmatrix}$, for $1 \leq i \leq 3$.

3.3 Matrix Manipulation Primitives

In this section, let

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1j} \\ a_{21} & a_{22} & \ldots & a_{2j} \\ \vdots & \vdots & & \vdots \\ a_{i1} & a_{i2} & \ldots & a_{ij} \end{pmatrix} \text{ be an } i \times j \text{ matrix and let}$$

$$B = \begin{pmatrix} b_{11} & b_{12} & \ldots & b_{1l} \\ b_{21} & b_{22} & \ldots & b_{2l} \\ \vdots & \vdots & & \vdots \\ b_{k1} & b_{k2} & \ldots & b_{kl} \end{pmatrix} \text{ be a } k \times l \text{ matrix.}$$

We present the following primitives.

▸ **(!id+** D**)**

Returns the zero (additive identity) matrix of dimension D, where D is a list of two values, (r c) which refer to the number of rows and the number of columns, respectively.

▶ **(!id*** n**)**

Returns the $n \times n$ identity matrix.

▶ **(!dim** A**)**

Returns the dimension of A as a list, that is **(!dim** A**)** gives the list **(**i j**)**.

▶ **(!opp** A**)**

Returns the additive inverse of A. In other words, **(!opp** A**)** returns the matrix

$$\begin{pmatrix} -a_{11} & -a_{12} & \dots & -a_{1j} \\ -a_{21} & -a_{22} & \dots & -a_{2j} \\ \vdots & \vdots & & \vdots \\ -a_{i1} & -a_{i2} & \dots & -a_{ij} \end{pmatrix}.$$

▶ **(!xpose** A**)**

Returns the transpose of A. That is, **(!xpose** A**)** gives the matrix

$$\begin{pmatrix} a_{11} & a_{21} & \dots & a_{i1} \\ a_{12} & a_{22} & \dots & a_{i2} \\ \vdots & \vdots & & \vdots \\ a_{1j} & a_{2j} & \dots & a_{ij} \end{pmatrix}$$

which is a $j \times i$ matrix whose n^{th} row is A's n^{th} column.

▶ **(!det2** A**)**

Returns the determinant of A. In this implementation, it is restricted to 2D. So the determinant of the A_2 is $a_{11}a_{22} - a_{21}a_{12}$.

▶ **(!add** A B**)**

Returns the matrix sum of A and B if $i = k$ and $j = l$. That means **(!add** A B**)** yields the matrix

$$A + B = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1j} + b_{1l} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2j} + b_{2l} \\ \vdots & \vdots & & \vdots \\ a_{i1} + b_{k1} & a_{i2} + b_{k2} & \dots & a_{ij} + b_{kl} \end{pmatrix}$$

if $i = k$ and $j = l$.

▶ **(!mult2** A B**)**

Returns the matrix product of A and B (instead of B and A) if $j = k$. More precisely, **(!mult2** A B**)** yields the matrix C. We have:

$$C = A \cdot B = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1l} \\ c_{21} & c_{22} & \dots & c_{2l} \\ \vdots & \vdots & & \vdots \\ c_{i1} & c_{i2} & \dots & c_{il} \end{pmatrix}$$

which is an $i \times l$ matrix if $j = k$, where $c_{mn} = \sum_{s=1}^{j} a_{ms}b_{sn}$ for $1 \le m \le i$ and $1 \le n \le l$.

▶ **(!mult** A_1 A_2 $\dots$ A_n**)**

Returns the matrix composite of $A_1, A_2, \ldots,$ and A_n. Let the dimension of A_i be $(x_i\ y_i)$. The composite matrix exists only if $y_i = x_{i+1}$ for $1 \leq i \leq n$. Note that **!mult** is a generalized form of **!mult2**.

§4 Turtle Graphics Primitives

Turtle graphics is at the center of the programming language Logo. We implement some of its basic primitives in Scheme. In this section, let V be a 2D vector $[x\ y]$ and let θ be any angle specified in degrees instead of in radians. The drawing *pen* can be *up* or *down*. We let Δ be a global identifier denoting the pen state. Also, two more registers maintaining the turtle position (Π) and the turtle heading (Θ) are global to the system. The position of system origin ($\Pi =$ [0, 0]) is at the center of the screen; the heading 0 is horizontal ($\Theta =$ [1, 0]).

▸ **(rotate** V θ**)**

Returns a vector which is the rotation of V by θ degrees. If **(rotate** V θ**)** yields the vector U, then

$$U = [x\ y] \cdot \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}.$$

▸ **(setheading** θ**)**

Sets Θ to $[\cos\theta,\ \sin\theta]$.

▸ **(setposition** V**)**

Sets Π to V and moves the turtle to V absolutely.

▸ **(pendown)** or **(pd)**

Sets Δ *down*.

▸ **(penup)** or **(pu)**

Sets Δ *up*.

▸ **(initturtle)**

Initializes turtle mode which yields $\Delta =$ *down*, $\Theta =$ [1, 0], and $\Pi =$ [0, 0].

▸ **(forward** d**)** or **(fd** d**)**

Moves the turtle to $\Pi + d\Theta$ which becomes the new Π. If Δ is *down*, a line will be drawn simultaneously from the old Π to the new Π.

▸ **(backward** d**)** or **(bk** d**)**

Moves the turtle to $\Pi - d\Theta$ which becomes the new Π. If Δ is *down*, a line will be drawn simultaneously from the old Π to the new Π.

▸ **(left** θ**)** or **(lt** θ**)**

Rotates Θ by θ degrees yielding a new Θ.

▸ **(right** θ**)** or **(rt** θ**)**

Rotates Θ by $-\theta$ degrees yielding a new Θ.

§5 References

[Fe83] Carol Fessenden, W. D. Clinger, D. P. Friedman, and C. T. Haynes, "⟦Scheme 311⟧version 4 reference manual", technical report #137, Department of Computer Science, Indiana University, February 1983.

[GIGI] *GIGI/ReGIS Handbook*, Digital Equipment Corporation, June 1981.

[Ab82] Harold Abelson, *LOGO for the Apple II*, BYTE/McGraw-Hill Book Company, Inc., 1982.

[AbSe] Harold Abelson and Andrew di Sessa, *Turtle Geometry*, MIT Press, 1980